



Intel® Math Kernel Library

Reference Manual

September 2007

Disclaimer and Legal Information

Document Number: 630813-027US

World Wide Web: <http://developer.intel.com>

Version	Version Information	Date
-001	Original Issue.	11/94
-002	Added functions crotg, zrotg. Documented versions of functions ?her2k, ?symm, ?syrk, and ?syr2k not previously described. Pagination revised.	5/95
-003	Changed the title; former title: "Intel BLAS Library for the Pentium® Processor Reference Manual." Added functions ?rotm, ?rotmg and updated Appendix C.	1/96
-004	Documents Intel® Math Kernel library (Intel® MKL) release 2.0 with the parallelism capability. Information on parallelism has been added in Chapter 1 and in section "BLAS Level 3 Routines" in Chapter 2.	11/96
-005	Two-dimensional FFTs have been added. C interface has been added to both one- and two-dimensional FFTs.	8/97
-006	Documents Intel Math Kernel Library release 2.1. Sparse BLAS section has been added in Chapter 2.	1/98
-007	Documents Intel Math Kernel Library release 3.0. Descriptions of LAPACK routines (Chapters 4 and 5) and CBLAS interface (Appendix C) have been added. Quick Reference has been excluded from the manual; MKL 3.0 Quick Reference is now available in HTML format.	1/99
-008	Documents Intel Math Kernel Library release 3.2. Description of FFT routines have been revised. In Chapters 4 and 5 NAG names for LAPACK routines have been excluded.	6/99
-009	New LAPACK routines for eigenvalue problems have been added in chapter 5.	11/99
-010	Documents Intel Math Kernel Library release 4.0. Chapter 6 describing the VML functions has been added.	06/00
-011	Documents Intel Math Kernel Library release 5.1. LAPACK section has been extended to include the full list of computational and driver routines.	04/01
-6001	Documents Intel Math Kernel Library release 6.0 beta. New DFT interface and Vector Statistical Library functions have been added.	07/02
-6002	Documents Intel Math Kernel Library 6.0 beta update. DFT functions description has been updated. CBLAS interface description was extended.	12/02
-6003	Documents Intel Math Kernel Library 6.0 gold. DFT functions have been updated. Auxiliary LAPACK routines' descriptions were added to the manual.	03/03
-6004	Documents Intel Math Kernel Library release 6.1.	07/03
-6005	Documents Intel Math Kernel Library release 7.0 beta. Includes ScaLAPACK and sparse solver descriptions.	11/03
-017	Documents Intel MKL and Intel® Cluster MKL release 7.0 gold. Auxiliary ScaLAPACK and alternative sparse solver interface were added.	04/04

Version	Version Information	Date
-018	Documents Intel MKL and Intel® Cluster MKL release 8.0 beta. Sparse BLAS and DFTI sections were extended. New functionality was added: Sparse BLAS, Cluster DFTI, iterative sparse solver, multiple-precision arithmetic, interval linear solver, and convolution/correlation. Fortran95 interface to LAPACK functions was added.	03/05
-019	Documents Intel MKL and Intel® Cluster MKL release 8.0 gold. Fortran95 interface to BLAS and Sparse BLAS functions has been added.	08/05
-020	Documents Intel MKL and Intel Cluster MKL release 8.0.2. PARDISO functionality description has been extended with indefinite symmetric matrices pivoting.	03/06
-021	Documents Intel MKL and Intel Cluster MKL release 8.1 gold. Chapter 13 on Trigonometric Transform functions has been added. Information on specific features of Fortran-95 implementation for LAPACK routines has been reflected in a new Appendix E and the relevant subsection of Chapter 3.	03/06
-022	Documents Intel MKL and Intel Cluster MKL release 9.0 beta. Statistical Functions, Fourier Transform Functions (Cluster DFT) descriptions have been updated. Description of new VML functions, RCI Sparse Solvers, and Poisson solver have been added. Chapter 13 has been renamed to PDE Support. Code examples have been expanded.	05/06
-023	Documents Intel MKL and Intel Cluster MKL release 9.0 gold. Complex Interval Solvers have been added. Description of the old deprecated FFT functions have been removed.	09/06
-024	Documents Intel MKL and Intel Cluster MKL release 9.1 beta. Optimization Solvers Routines and Support Functions chapters have been added. Chapters on Sparse Solvers and Partial Differential Equations Support have been extended. LAPACK chapters have been partially updated to reflect LAPACK 3.1 version.	01/07
-025	Documents Intel MKL and Intel Cluster MKL release 9.1 gold. BLACS chapter has been added. Chapters on BLAS and FFT have been extended. LAPACK chapters have been additionally updated to reflect LAPACK 3.1 version. Description of BSR format has been added to Appendix A. New FFT examples have been added to Appendix C.	03/07
-026	Documents Intel MKL and Intel Cluster MKL release 10.0 beta. BLACS, BLAS, Sparse Solver, VML, and FFT chapters have been extended. Description of new Threading Control functions has been added to Support Functions chapter.	07/07
-027	Documents Intel MKL release 10.0. BLAS, Sparse Solver, VML, and PDES chapters have been updated.	09/07

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel's Web Site. Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See http://www.intel.com/products/processor_number for details.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino Atom, Centrino Inside, Centrino logo, Core Inside, FlashFile, i960, InstantIP, Intel, Intel logo, Intel386, Intel486, IntelDX2, IntelDX4, IntelSX2, Intel Atom, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDCharm, Pentium, Pentium Inside, skool, Sound Mark, The Journey Inside, Viiv Inside, vPro Inside, VTune, Xeon, and Xeon Inside are trademarks of Intel Corporation in the U.S. and other countries.

* Other names and brands may be claimed as the property of others.

Copyright© 1994-2008, Intel Corporation. All rights reserved.

Portions© Copyright 2001 Hewlett-Packard Development Company, L.P.

Chapters 4 and 5 include derivative work portions that have been copyrighted:

© 1991, 1992, and 1998 by The Numerical Algorithms Group, Ltd.

Contents

Version Information.....	43
Legal Information.....	45

Chapter 1: Overview

About This Software.....	47
Technical Support.....	48
BLAS Routines.....	48
Sparse BLAS Routines.....	49
LAPACK Routines.....	49
ScaLAPACK Routines.....	50
Sparse Solver Routines.....	50
VML Functions.....	50
Statistical Functions.....	51
Fourier Transform Functions.....	51
Interval Solver Routines.....	51
Partial Differential Equations Support.....	51
Optimization Solver Routines.....	52
Support Functions.....	52
BLACS Routines.....	52
GMP Arithmetic Functions.....	52
Performance Enhancements.....	53
Parallelism.....	53
Platforms Supported.....	54
About This Manual.....	54

Audience for This Manual.....	55
Manual Organization.....	55
Notational Conventions.....	57

Chapter 2: BLAS and Sparse BLAS Routines

BLAS Routines.....	59
Routine Naming Conventions.....	59
Fortran 95 Interface Conventions.....	61
Matrix Storage Schemes.....	62
BLAS Level 1 Routines.....	63
?asum.....	64
?axpy.....	65
?copy.....	67
?dot.....	68
?sdot.....	70
?dotc.....	71
?dotu.....	73
?nrm2.....	74
?rot.....	75
?rotg.....	77
?rotm.....	78
?rotmg.....	81
?scal.....	83
?swap.....	85
i?amax.....	86
i?amin.....	87
dcabs1.....	89
BLAS Level 2 Routines.....	89
?gbmv.....	91
?gemv.....	95
?ger.....	98
?gerc.....	100

?geru.....	102
?hbm.....	104
?hem.....	107
?her.....	110
?her2.....	112
?hpm.....	114
?hpr.....	117
?hpr2.....	119
?sbm.....	121
?spm.....	125
?spr.....	127
?spr2.....	129
?sym.....	132
?syr.....	134
?syr2.....	136
?tbm.....	139
?tbs.....	142
?tpm.....	146
?tpsv.....	148
?trm.....	150
?trsv.....	153
BLAS Level 3 Routines.....	155
?gemm.....	156
?gemm3m.....	160
?hemm.....	164
?herk.....	167
?her2k.....	170
?symm.....	173
?syrk.....	177
?syr2k.....	180
?trmm.....	184
?trsm.....	187

Sparse BLAS Level 1 Routines.....	190
Vector Arguments.....	190
Naming Conventions.....	191
Routines and Data Types.....	191
BLAS Level 1 Routines That Can Work With Sparse Vectors.....	192
?axpyi.....	192
?doti.....	194
?dotci.....	195
?dotui.....	197
?gthr.....	198
?gthrz.....	200
?roti.....	201
?sctr.....	203
Sparse BLAS Level 2 and Level 3 Routines.....	204
Naming Conventions in Sparse BLAS Level 2 and Level 3.....	205
Sparse Matrix Storage Formats.....	206
Routines and Supported Operations.....	206
Interface Consideration.....	208
Sparse BLAS Level 2 and Level 3 Routines.....	215
mkl_?csrgemv.....	219
mkl_dbsgemv.....	221
mkl_dcoogemv.....	224
mkl_ddiagemv.....	226
mkl_dcsrsymv.....	228
mkl_dbrsymv.....	230
mkl_dcoosymv.....	232
mkl_ddiasymv.....	235
mkl_dcsrtrsv.....	237
mkl_dbstrsv.....	239
mkl_dcootrsv.....	242
mkl_ddiatrsv.....	244
mkl_cspblas_dcsrgemv.....	247

mkl_cspblas_dbsrgemv.....	249
mkl_cspblas_dcoogemv.....	251
mkl_cspblas_dcsrsymv.....	254
mkl_cspblas_dbsrsymv.....	256
mkl_cspblas_dcoosymv.....	258
mkl_cspblas_dcsrtrsv.....	261
mkl_cspblas_dbsrtrsv.....	263
mkl_cspblas_dcootrsv.....	266
mkl_dcsrcmv.....	268
mkl_dbsrmv.....	271
mkl_dcscmv.....	274
mkl_dcoomv.....	277
mkl_dcsrcsv.....	280
mkl_dbsrsv.....	283
mkl_dcscsv.....	286
mkl_dcoosv.....	289
mkl_dcsrcmm.....	292
mkl_dbsrmm.....	295
mkl_dcscmm.....	299
mkl_dcoomm.....	302
mkl_dcsrcsm.....	305
mkl_dcscsm.....	309
mkl_dcoosm.....	312
mkl_dbsrsm.....	315
mkl_ddiamv.....	318
mkl_dskymv.....	321
mkl_ddiasv.....	324
mkl_dskysv.....	326
mkl_ddiamm.....	329
mkl_dskymm.....	332
mkl_ddiasm.....	335
mkl_dskysm.....	337

Chapter 3: LAPACK Routines: Linear Equations

Routine Naming Conventions.....	342
Fortran 95 Interface Conventions.....	343
Intel® MKL Fortran 95 Interfaces for LAPACK Routines vs. Netlib Implementation.....	345
Matrix Storage Schemes.....	347
Mathematical Notation.....	347
Error Analysis.....	348
Computational Routines.....	349
Routines for Matrix Factorization.....	351
?getrf.....	352
?gbtrf.....	354
?gttrf.....	356
?potrf.....	358
?pptrf.....	360
?pbtrf.....	363
?pttrf.....	365
?sytrf.....	367
?hetrf.....	370
?sptrf.....	374
?hptrf.....	376
Routines for Solving Systems of Linear Equations.....	379
?getrs.....	379
?gbtrs.....	382
?gttrs.....	385
?potrs.....	388
?pptrs.....	390
?pbtrs.....	393
?pttrs.....	396
?sytrs.....	398
?hetrs.....	400

?sptrs.....	403
?hptrs.....	406
?trtrs.....	408
?tptrs.....	411
?tbtrs.....	414
Routines for Estimating the Condition Number.....	416
?gecon.....	417
?gbcon.....	419
?gtcon.....	422
?pocon.....	425
?ppcon.....	427
?pbcon.....	429
?ptcon.....	432
?sycon.....	434
?hecon.....	436
?spcon.....	438
?hpcon.....	440
?trcon.....	443
?tpcon.....	445
?tbcon.....	448
Refining the Solution and Estimating Its Error.....	450
?gerfs.....	451
?gbrfs.....	454
?gtrfs.....	457
?porfs.....	460
?pprfs.....	463
?pbrfs.....	466
?ptrfs.....	469
?syrfs.....	472
?herfs.....	475
?sprfs.....	478
?hprfs.....	481

?trrfs.....	484
?tprfs.....	487
?tbrfs.....	490
Routines for Matrix Inversion.....	493
?getri.....	493
?potri.....	496
?pptri.....	498
?sytri.....	500
?hetri.....	502
?sptri.....	504
?hptri.....	506
?trtri.....	508
?tptri.....	510
Routines for Matrix Equilibration.....	511
?geequ.....	512
?gbequ.....	514
?poequ.....	517
?ppequ.....	519
?pbequ.....	521
Driver Routines.....	524
?gesv.....	525
?gesvx.....	529
?gbsv.....	536
?gbsvx.....	538
?gtsv.....	545
?gtsvx.....	547
?posv.....	552
?posvx.....	554
?ppsv.....	560
?ppsvx.....	562
?pbsv.....	568
?pbsvx.....	570

?ptsv.....	576
?ptsvx.....	578
?sysv.....	582
?sysvx.....	585
?hesv.....	590
?hesvx.....	593
?spsv.....	598
?spsvx.....	601
?hpsv.....	605
?hpsvx.....	607

Chapter 4: LAPACK Routines: Least Squares and Eigenvalue

Problems

Routine Naming Conventions.....	614
Matrix Storage Schemes.....	616
Mathematical Notation.....	616
Computational Routines.....	617
Orthogonal Factorizations.....	617
?geqrf.....	619
?geqpf.....	622
?geqp3.....	625
?orgqr.....	629
?ormqr.....	632
?ungqr.....	635
?unmqr.....	637
?gelqf.....	640
?orglq.....	644
?ormlq.....	646
?unglq.....	649
?unmlq.....	652
?geqlf.....	655
?orgql.....	657

?ungql.....	660
?ormql.....	662
?unmql.....	665
?gerqf.....	668
?orgrq.....	671
?ungrq.....	673
?ormrq.....	676
?unmrq.....	679
?tzzrf.....	682
?ormrz.....	684
?unmrz.....	687
?ggqrf.....	691
?ggrqf.....	695
Singular Value Decomposition.....	699
?gebrd.....	702
?gbbbrd.....	706
?orgbr.....	709
?ormbr.....	713
?ungbr.....	716
?unmbr.....	720
?bdsqr.....	724
?bdsdc.....	728
Symmetric Eigenvalue Problems.....	731
?sytrd.....	736
?syldb.....	739
?herdb.....	742
?orgtr.....	745
?ormtr.....	747
?hetrd.....	750
?ungtr.....	753
?unmtr.....	755
?sptrd.....	758

?opgtr.....	760
?opmtr.....	762
?hptrd.....	765
?upgtr.....	767
?upmtr.....	769
?sbtrd.....	771
?hbtrd.....	774
?sterf.....	776
?steqr.....	778
?stemr.....	782
?stedc.....	787
?stegr.....	793
?pteqr.....	798
?stebz.....	802
?stein.....	806
?disna.....	809
Generalized Symmetric-Definite Eigenvalue Problems.....	811
?sygst.....	812
?hegst.....	814
?spgst.....	816
?hpgst.....	819
?sbgst.....	821
?hbgst.....	824
?pbstf.....	827
Nonsymmetric Eigenvalue Problems.....	829
?gehrd.....	834
?orghr.....	836
?ormhr.....	839
?unghr.....	843
?unmhr.....	846
?gebal.....	849
?gebak.....	853

?hseqr.....	855
?hsein.....	861
?trevc.....	867
?trsna.....	872
?trexc.....	877
?trsen.....	880
?trsyl.....	886
Generalized Nonsymmetric Eigenvalue Problems.....	889
?gghrd.....	890
?ggbal.....	894
?ggbak.....	896
?hgeqz.....	899
?tgevc.....	907
?tgexc.....	912
?tgsen.....	916
?tgsyl.....	923
?tgsna.....	928
Generalized Singular Value Decomposition.....	933
?ggsvp.....	934
?tgsja.....	939
Driver Routines.....	946
Linear Least Squares (LLS) Problems.....	947
?gels.....	947
?gelsy.....	951
?gelss.....	956
?gelsd.....	960
Generalized LLS Problems.....	964
?gglse.....	965
?ggglm.....	969
Symmetric Eigenproblems.....	972
?syev.....	973
?heev.....	976

?syevd.....	979
?heevd.....	983
?syevx.....	987
?heevx.....	992
?syevr.....	997
?heevr.....	1003
?spev.....	1010
?hpev.....	1012
?spevd.....	1014
?hpevd.....	1018
?spevx.....	1023
?hpevx.....	1027
?sbev.....	1032
?hbev.....	1034
?sbevd.....	1037
?hbevd.....	1041
?sbevx.....	1046
?hbevx.....	1050
?stev.....	1055
?stevd.....	1057
?stevx.....	1061
?stevr.....	1065
Nonsymmetric Eigenproblems.....	1071
?gees.....	1072
?geesx.....	1077
?geev.....	1084
?geevx.....	1089
Singular Value Decomposition.....	1096
?gesvd.....	1096
?gesdd.....	1102
?ggsvd.....	1107
Generalized Symmetric Definite Eigenproblems.....	1113

?sygv.....	1114
?hegv.....	1118
?sygvd.....	1121
?hegvd.....	1125
?sygvx.....	1130
?hegvx.....	1136
?spgv.....	1142
?hpgv.....	1145
?spgvd.....	1148
?hpgvd.....	1152
?spgvx.....	1157
?hpgvx.....	1162
?sbgv.....	1167
?hbgv.....	1170
?sbgvd.....	1173
?hbgvd.....	1177
?sbgvx.....	1182
?hbgvx.....	1187
Generalized Nonsymmetric Eigenproblems.....	1192
?gges.....	1193
?ggesx.....	1200
?ggev.....	1209
?ggevx.....	1215

Chapter 5: LAPACK Auxiliary and Utility Routines

Auxiliary Routines.....	1225
?lacgv.....	1240
?lacrm.....	1240
?lacrt.....	1242
?laesy.....	1243
?rot.....	1244
?spmv.....	1246

?spr.....	1248
?symv.....	1249
?syr.....	1251
i?max1.....	1253
?sum1.....	1254
?gbtf2.....	1255
?gebd2.....	1256
?gehd2.....	1259
?gelq2.....	1261
?geql2.....	1263
?geqr2.....	1264
?gerq2.....	1266
?gesc2.....	1268
?getc2.....	1269
?getf2.....	1271
?gtts2.....	1272
?isnan.....	1274
?laisnan.....	1274
?labrd.....	1275
?lacn2.....	1278
?lacon.....	1280
?lacpy.....	1281
?ladiv.....	1283
?lae2.....	1284
?laebz.....	1285
?laed0.....	1290
?laed1.....	1293
?laed2.....	1295
?laed3.....	1298
?laed4.....	1300
?laed5.....	1302
?laed6.....	1303

?laed7.....	1305
?laed8.....	1309
?laed9.....	1313
?laeda.....	1315
?laein.....	1317
?laev2.....	1320
?laexc.....	1322
?lag2.....	1323
?lags2.....	1326
?lagtf.....	1328
?lagtm.....	1330
?lagts.....	1332
?lagv2.....	1334
?lahqr.....	1336
?lahrd.....	1339
?lahr2.....	1342
?laic1.....	1345
?laln2.....	1347
?lals0.....	1350
?lalsa.....	1354
?lalsd.....	1358
?lamrg.....	1361
?laneg.....	1362
?langb.....	1363
?lange.....	1365
?langt.....	1366
?lanhs.....	1368
?lansb.....	1369
?lanhb.....	1371
?lansp.....	1373
?lanhp.....	1375
?lanst/?lanht.....	1376

?lansy.....	1378
?lanhe.....	1379
?lantb.....	1381
?lantp.....	1383
?lantr.....	1385
?lanv2.....	1387
?lapll.....	1388
?lapmt.....	1389
?lapy2.....	1390
?lapy3.....	1391
?laqgb.....	1392
?laqge.....	1394
?laqhb.....	1396
?laqp2.....	1398
?laqps.....	1400
?laqr0.....	1402
?laqr1.....	1406
?laqr2.....	1408
?laqr3.....	1412
?laqr4.....	1416
?laqr5.....	1420
?laqsb.....	1424
?laqsp.....	1426
?laqsy.....	1428
?laqtr.....	1429
?lar1v.....	1432
?lar2v.....	1435
?larf.....	1437
?larfb.....	1439
?larfg.....	1441
?larft.....	1443
?larfx.....	1446

?largv.....	1447
?larnv.....	1449
?larra.....	1450
?larrb.....	1452
?larrc.....	1454
?larrd.....	1456
?larre.....	1460
?larrf.....	1464
?larvj.....	1467
?larrk.....	1469
?larrr.....	1470
?larrv.....	1472
?lartg.....	1476
?lartv.....	1478
?laruv.....	1479
?larz.....	1480
?larzb.....	1482
?larzt.....	1485
?las2.....	1488
?lascl.....	1489
?lasd0.....	1491
?lasd1.....	1493
?lasd2.....	1496
?lasd3.....	1500
?lasd4.....	1503
?lasd5.....	1505
?lasd6.....	1506
?lasd7.....	1511
?lasd8.....	1515
?lasd9.....	1517
?lasda.....	1520
?lasdq.....	1524

?lasdt.....	1527
?laset.....	1528
?lasq1.....	1529
?lasq2.....	1530
?lasq3.....	1532
?lasq4.....	1533
?lasq5.....	1534
?lasq6.....	1536
?lasr.....	1537
?lasrt.....	1541
?lassq.....	1542
?lasv2.....	1544
?laswp.....	1545
?lasy2.....	1547
?lasyf.....	1549
?lahef.....	1551
?latbs.....	1554
?latdf.....	1556
?latps.....	1559
?latrd.....	1561
?latrs.....	1565
?latrz.....	1569
?lauu2.....	1571
?lauum.....	1573
?lazq3.....	1574
?lazq4.....	1577
?org2l/?ung2l.....	1579
?org2r/?ung2r.....	1580
?orgl2/?ungl2.....	1582
?orgr2/?ungr2.....	1584
?orm2l/?unm2l.....	1585
?orm2r/?unm2r.....	1588

?orml2/?unml2.....	1590
?ormr2/?unmr2.....	1593
?ormr3/?unmr3.....	1595
?pbt2.....	1598
?potf2.....	1600
?ptts2.....	1601
?rscl.....	1603
?sygs2/?hegs2.....	1604
?sytd2/?hetd2.....	1606
?sytf2.....	1608
?hetf2.....	1610
?tgex2.....	1612
?tgsy2.....	1615
?trti2.....	1619
clag2z.....	1620
dlag2s.....	1621
slag2d.....	1622
zlag2c.....	1623
Utility Functions and Routines.....	1624
ilaver.....	1625
ilaenv.....	1626
iparmq.....	1629
ieeeck.....	1631
lsame.....	1632
lsamen.....	1632
?labad.....	1633
?lamch.....	1634
?lamc1.....	1635
?lamc2.....	1636
?lamc3.....	1637
?lamc4.....	1637
?lamc5.....	1638

second/dsecnd.....	1639
xerbla.....	1640

Chapter 6: ScaLAPACK Routines

Overview.....	1641
Routine Naming Conventions.....	1642
Computational Routines.....	1644
Linear Equations.....	1644
Routines for Matrix Factorization.....	1645
p?getrf.....	1646
p?gbtrf.....	1648
p?dbtrf.....	1651
p?potrf.....	1653
p?pbtrf.....	1655
p?pttrf.....	1658
p?dttrf.....	1660
Routines for Solving Systems of Linear Equations.....	1663
p?getrs.....	1663
p?gbtrs.....	1665
p?potrs.....	1668
p?pbtrs.....	1670
p?pttrs.....	1673
p?dttrs.....	1676
p?dbtrs.....	1679
p?trtrs.....	1682
Routines for Estimating the Condition Number.....	1684
p?gecon.....	1685
p?pocon.....	1688
p?trcon.....	1691
Refining the Solution and Estimating Its Error.....	1694
p?gerfs.....	1695
p?porfs.....	1699

p?trrfs.....	1703
Routines for Matrix Inversion.....	1707
p?getri.....	1707
p?potri.....	1710
p?trtri.....	1712
Routines for Matrix Equilibration.....	1713
p?geequ.....	1714
p?poequ.....	1716
Orthogonal Factorizations.....	1719
p?geqrf.....	1719
p?geqpf.....	1722
p?orgqr.....	1725
p?ungqr.....	1727
p?ormqr.....	1729
p?unmqr.....	1733
p?gelqf.....	1736
p?orglq.....	1739
p?unglq.....	1741
p?ormlq.....	1743
p?unmlq.....	1747
p?geqlf.....	1750
p?orgql.....	1753
p?ungql.....	1755
p?ormql.....	1757
p?unmql.....	1761
p?gerqf.....	1764
p?orgrq.....	1767
p?ungrq.....	1769
p?ormrq.....	1772
p?unmrq.....	1775
p?tzzrf.....	1779
p?ormrz.....	1782

p?unmrz.....	1785
p?ggqrf.....	1789
p?ggrqf.....	1795
Symmetric Eigenproblems.....	1800
p?sytrd.....	1801
p?ormtr.....	1805
p?hetrd.....	1809
p?unmtr.....	1813
p?stebz.....	1817
p?stein.....	1821
Nonsymmetric Eigenvalue Problems.....	1826
p?gehrd.....	1826
p?ormhr.....	1830
p?unmhr.....	1834
p?lahqr.....	1837
Singular Value Decomposition.....	1840
p?gebrd.....	1840
p?ormbr.....	1845
p?unmbr.....	1850
Generalized Symmetric-Definite Eigen Problems.....	1855
p?sygst.....	1856
p?hegst.....	1858
Driver Routines.....	1861
p?gesv.....	1861
p?gesvx.....	1864
p?gbsv.....	1871
p?dbsv.....	1874
p?dtsv.....	1877
p?posv.....	1880
p?posvx.....	1882
p?pbsv.....	1889
p?ptsv.....	1892

p?gels.....	1895
p?syev.....	1900
p?syevx.....	1903
p?heevx.....	1911
p?gesvd.....	1920
p?sygvx.....	1925
p?hegvx.....	1934

Chapter 7: ScaLAPACK Auxiliary and Utility Routines

Auxiliary Routines.....	1945
p?lacgv.....	1951
p?max1.....	1952
?combamax1.....	1954
p?sum1.....	1955
p?dbtrsv.....	1956
p?dttrsv.....	1960
p?gebd2.....	1964
p?gehd2.....	1968
p?gelq2.....	1971
p?geql2.....	1974
p?geqr2.....	1977
p?gerq2.....	1980
p?getf2.....	1983
p?labrd.....	1985
p?lacon.....	1990
p?laconsb.....	1992
p?lcp2.....	1993
p?lcp3.....	1995
p?lacpy.....	1997
p?laevswp.....	1999
p?lahrd.....	2001
p?laiect.....	2004

p?lange.....	2005
p?lanhs.....	2008
p?lansy, p?lanhe.....	2010
p?lantr.....	2012
p?lapiv.....	2015
p?laqge.....	2019
p?laqsy.....	2021
p?lared1d.....	2024
p?lared2d.....	2025
p?larf.....	2027
p?larfb.....	2031
p?larfc.....	2035
p?larfg.....	2038
p?larft.....	2040
p?larz.....	2043
p?larzb.....	2047
p?larzc.....	2052
p?larzt.....	2056
p?lascl.....	2060
p?laset.....	2062
p?lasmsub.....	2064
p?lassq.....	2065
p?laswp.....	2068
p?latra.....	2070
p?latrd.....	2071
p?latrs.....	2075
p?latrz.....	2078
p?lauu2.....	2081
p?lauum.....	2083
p?lawil.....	2084
p?org2l/p?ung2l.....	2085
p?org2r/p?ung2r.....	2088

p?orgl2/p?ungl2.....	2091
p?orgr2/p?ungr2.....	2094
p?orm2l/p?unm2l.....	2097
p?orm2r/p?unm2r.....	2101
p?orml2/p?unml2.....	2106
p?ormr2/p?unmr2.....	2110
p?pbtrsv.....	2115
p?pttrsv.....	2119
p?potf2.....	2123
p?rscl.....	2125
p?sygs2/p?hegs2.....	2127
p?sytd2/p?hetd2.....	2130
p?trti2.....	2134
?lamsh.....	2136
?laref.....	2137
?lasorte.....	2140
?lasrt2.....	2141
?stein2.....	2142
?dbtf2.....	2145
?dbtrf.....	2147
?dttrf.....	2149
?dttrsv.....	2150
?pttrsv.....	2152
?steqr2.....	2154
Utility Functions and Routines.....	2156
p?labad.....	2156
p?lachkieee.....	2157
p?lamch.....	2158
p?lasnbt.....	2160
pxerbla.....	2161

Chapter 8: Sparse Solver Routines

PARDISO* - Parallel Direct Sparse Solver Interface.....	2163
pardiso.....	2164
Direct Sparse Solver (DSS) Interface Routines.....	2182
DSS Interface Description.....	2184
dss_create.....	2186
dss_define_structure.....	2187
dss_reorder.....	2189
dss_factor_real, dss_factor_complex.....	2190
dss_solve_real, dss_solve_complex.....	2191
dss_delete.....	2192
dss_statistics.....	2193
mkl_cvt_to_null_terminated_str.....	2196
Implementation Details.....	2197
Iterative Sparse Solvers based on Reverse Communication Interface	
(RCI ISS).....	2198
CG Interface Description.....	2203
FGMRES Interface Description.....	2209
dcg_init.....	2218
dcg_check.....	2219
dcg.....	2221
dcg_get.....	2223
dcgmrhs_init.....	2224
dcgmrhs_check.....	2225
dcgmrhs.....	2227
dcgmrhs_get.....	2229
dfgmres_init.....	2230
dfgmres_check.....	2231
dfgmres.....	2233
dfgmres_get.....	2235
Implementation Details.....	2237
Preconditioners based on Incomplete LU Factorization Technique...	2238
ILU0 and ILUT Preconditioners Interface Description.....	2241

dcscrilu0.....	2242
dcscrilut.....	2247
Calling Sparse Solver and Preconditioner Routines from C/C++.....	2252

Chapter 9: Vector Mathematical Functions

Data Types and Accuracy Modes.....	2255
Function Naming Conventions.....	2256
Functions Interface.....	2257
VML Mathematical Functions.....	2257
Pack Functions.....	2257
Unpack Functions.....	2257
Service Functions.....	2258
Input Parameters.....	2258
Output Parameters.....	2259
Vector Indexing Methods.....	2259
Error Diagnostics.....	2260
VML Mathematical Functions.....	2260
Arithmetic Functions.....	2263
Add.....	2263
Sub.....	2265
Sqr.....	2266
Mul.....	2268
MulByConj.....	2269
Conj.....	2271
Abs.....	2272
Power and Root Functions.....	2274
Inv.....	2274
Div.....	2275
Sqrt.....	2277
InvSqrt.....	2279
Cbrt.....	2280
InvCbrt.....	2281

Pow2o3.....	2282
Pow3o2.....	2283
Pow.....	2285
Powx.....	2287
Hypot.....	2289
Exponential and Logarithmic Functions.....	2291
Exp.....	2291
Expm1.....	2293
Ln.....	2294
Log10.....	2296
Log1p.....	2297
Trigonometric Functions.....	2299
Cos.....	2299
Sin.....	2301
SinCos.....	2303
CIS.....	2304
Tan.....	2305
Acos.....	2307
Asin.....	2309
Atan.....	2311
Atan2.....	2312
Hyperbolic Functions.....	2314
Cosh.....	2314
Sinh.....	2316
Tanh.....	2318
Acosh.....	2319
Asinh.....	2321
Atanh.....	2323
Special Functions.....	2324
Erf.....	2324
Erfc.....	2326
ErfInv.....	2328

Rounding Functions.....	2330
Floor.....	2330
Ceil.....	2331
Trunc.....	2333
Round.....	2334
NearbyInt.....	2335
Rint.....	2336
Modf.....	2338
VML Pack/Unpack Functions.....	2339
Pack.....	2339
Unpack.....	2342
VML Service Functions.....	2344
SetMode.....	2345
GetMode.....	2347
SetErrStatus.....	2349
GetErrStatus.....	2350
ClearErrStatus.....	2351
SetErrorCallBack.....	2351
GetErrorCallBack.....	2354
ClearErrorCallBack.....	2355

Chapter 10: Statistical Functions

Random Number Generators.....	2357
Conventions.....	2358
Mathematical Notation.....	2359
Naming Conventions.....	2360
Basic Generators.....	2365
BRNG Parameter Definition.....	2367
Random Streams.....	2368
Data Types.....	2368
Error Reporting.....	2369
Service Routines.....	2370

NewStream.....	2372
NewStreamEx.....	2373
iNewAbstractStream.....	2375
dNewAbstractStream.....	2377
sNewAbstractStream.....	2380
DeleteStream.....	2383
CopyStream.....	2384
CopyStreamState.....	2385
SaveStreamF.....	2386
LoadStreamF.....	2388
LeapfrogStream.....	2389
SkipAheadStream.....	2393
GetStreamStateBrng.....	2397
GetNumRegBrngs.....	2398
Distribution Generators.....	2399
Continuous Distributions.....	2401
Discrete Distributions.....	2441
Advanced Service Routines.....	2461
Data types.....	2462
RegisterBrng.....	2464
GetBrngProperties.....	2465
Formats for User-Designed Generators.....	2466
Convolution and Correlation.....	2470
Overview.....	2470
Naming Conventions.....	2471
Data Types.....	2472
Parameters.....	2473
Task Status and Error Reporting.....	2475
Task Constructors.....	2478
NewTask.....	2479
NewTask1D.....	2481
NewTaskX.....	2483

NewTaskX1D.....	2486
Task Editors.....	2489
SetMode.....	2490
SetInternalPrecision.....	2492
SetStart.....	2494
SetDecimation.....	2495
Task Execution Routines.....	2496
Exec.....	2498
Exec1D.....	2500
ExecX.....	2503
ExecX1D.....	2506
Task Destructors.....	2508
DeleteTask.....	2508
Task Copy.....	2510
CopyTask.....	2510
Usage Examples.....	2511
Mathematical Notation and Definitions.....	2515
Data Allocation.....	2516

Chapter 11: Fourier Transform Functions

FFT Functions.....	2522
Computing FFT.....	2523
FFT Interface.....	2523
Status Checking Functions.....	2525
ErrorClass.....	2525
ErrorMessage.....	2527
Descriptor Manipulation Functions.....	2528
CreateDescriptor.....	2529
CommitDescriptor.....	2531
CopyDescriptor.....	2532
FreeDescriptor.....	2534
FFT Computation Functions.....	2535

ComputeForward.....	2536
ComputeBackward.....	2539
Descriptor Configuration Functions.....	2541
SetValue.....	2542
GetValue.....	2544
Configuration Settings.....	2547
Precision of transform.....	2552
Forward domain of transform.....	2552
Transform dimension and lengths.....	2553
Number of transforms.....	2554
Scale.....	2554
Placement of result.....	2554
Packed formats.....	2554
Storage schemes.....	2559
Number of user threads.....	2570
Input and output distances.....	2571
Strides.....	2572
Ordering.....	2574
Transposition.....	2575
Cluster FFT Functions.....	2575
Computing Cluster FFT.....	2577
Distributing Data among Processes.....	2578
Cluster FFT Interface.....	2581
Descriptor Manipulation Functions.....	2582
CreateDescriptorDM.....	2582
CommitDescriptorDM.....	2584
FreeDescriptorDM.....	2586
FFT Computation Functions.....	2587
ComputeForwardDM.....	2587
ComputeBackwardDM.....	2591
Descriptor Configuration Functions.....	2594
SetValueDM.....	2594

GetValueDM.....	2597
Error Codes.....	2601

Chapter 12: Intel® Interval Linear Solvers

Routine Naming Conventions.....	2604
Routines for Fast Solution of Interval Systems.....	2605
?trtrs.....	2605
?gegas.....	2607
?gehss.....	2609
?gekws.....	2611
?gegss.....	2613
?gehbs.....	2615
Routines for Sharp Solution of Interval Systems.....	2616
?gepps.....	2616
?gepps.....	2619
Routines for Inverting Interval Matrices.....	2623
?trtri.....	2623
?geszi.....	2624
Routines for Checking Properties of Interval Matrices.....	2625
?gerbr.....	2625
?gesvr.....	2627
Auxiliary and Utility Routines.....	2630
?gemip.....	2630

Chapter 13: Partial Differential Equations Support

Trigonometric Transform Routines.....	2633
Transforms Implemented.....	2634
Sequence of Invoking TT Routines.....	2636
Interface Description.....	2639
TT Routines.....	2640
?_init_trig_transform.....	2640

?_commit_trig_transform.....	2641
?_forward_trig_transform.....	2644
?_backward_trig_transform.....	2646
free_trig_transform.....	2648
Common Parameters.....	2649
Implementation Details.....	2654
Poisson Library Routines	2657
Poisson Library Implemented.....	2658
Sequence of Invoking PL Routines.....	2666
Interface Description.....	2669
PL Routines for the Cartesian Solver.....	2670
?_init_Helmholtz_2D/?_init_Helmholtz_3D.....	2670
?_commit_Helmholtz_2D/?_commit_Helmholtz_3D....	2673
?_Helmholtz_2D/?_Helmholtz_3D.....	2679
free_Helmholtz_2D/free_Helmholtz_3D.....	2685
PL Routines for the Spherical Solver.....	2686
?_init_sph_p/?_init_sph_np.....	2686
?_commit_sph_p/?_commit_sph_np.....	2689
?_sph_p/?_sph_np.....	2691
free_sph_p/free_sph_np.....	2694
Common Parameters.....	2695
Implementation Details.....	2704
Calling PDE Support Routines from Fortran 90.....	2717

Chapter 14: Optimization Solver Routines

Organization and Implementation.....	2721
Nonlinear Least Squares Problem without Constraints.....	2723
dtrnlsp_init.....	2724
dtrnlsp_solve.....	2725
dtrnlsp_get.....	2727
dtrnlsp_delete.....	2729
Examples of dtrnlsp Usage.....	2730

Nonlinear Least Squares Problem with Linear (Bound) Constraints..	2743
dtrnlspbc_init.....	2744
dtrnlspbc_solve.....	2746
dtrnlspbc_get.....	2747
dtrnlspbc_delete.....	2749
Examples of dtrnlspbc Usage.....	2750
Jacobi Matrix Calculation Routines.....	2765
djacobi_init.....	2766
djacobi_solve.....	2767
djacobi_delete.....	2768
djacobi.....	2768
Examples of djacobi_solve Usage.....	2770
Examples of djacobi Usage.....	2777

Chapter 15: Support Functions

Version Information Functions.....	2783
MKLGetVersion.....	2783
MKLGetVersionString.....	2785
Threading Control Functions.....	2787
mkl_set_num_threads.....	2788
mkl_domain_set_num_threads.....	2788
mkl_set_dynamic.....	2789
mkl_get_max_threads.....	2790
mkl_domain_get_max_threads.....	2791
mkl_get_dynamic.....	2792
Error Handling Functions.....	2793
xerbla.....	2793
pxerbla.....	2793
Equality Test Functions.....	2794
lsame.....	2794
lsamen.....	2795
Timing Functions.....	2796

second/dsecnd.....	2796
getcpuclocks.....	2796
getcpufrequency.....	2797
setcpufrequency.....	2797
Memory Functions.....	2798
MKL_FreeBuffers.....	2798
MKL_MemStat.....	2800
MKL_malloc.....	2801
MKL_free.....	2802
Examples of MKL_malloc(), MKL_free(), MKL_MemStat() Usage.....	2803

Chapter 16: BLACS Routines

Matrix Shapes.....	2807
BLACS Combine Operations.....	2808
?gamx2d.....	2810
?gamn2d.....	2812
?gsum2d.....	2814
BLACS Point To Point Communication.....	2815
?gesd2d.....	2818
?trsd2d.....	2819
?gerv2d.....	2820
?trrv2d.....	2821
BLACS Broadcast Routines.....	2822
?gebs2d.....	2823
?trbs2d.....	2824
?gebr2d.....	2825
?trbr2d.....	2826
BLACS Support Routines.....	2827
Initialization Routines.....	2827
blacs_pinfo.....	2828
blacs_setup.....	2828

blacs_get.....	2830
blacs_set.....	2831
blacs_gridinit.....	2832
blacs_gridmap.....	2834
Destruction Routines.....	2835
blacs_freebuff.....	2836
blacs_gridexit.....	2836
blacs_abort.....	2837
blacs_exit.....	2837
Informational Routines.....	2838
blacs_gridinfo.....	2838
blacs_pnum.....	2839
blacs_pcoord.....	2840
Miscellaneous Routines.....	2840
blacs_barrier.....	2841
Examples of BLACS Routines Usage.....	2842

Appendix A: Linear Solvers Basics

Sparse Linear Systems.....	2861
Matrix Fundamentals.....	2862
Direct Method.....	2863
Sparse Matrix Storage Formats.....	2869
Interval Linear Systems.....	2883
Intervals.....	2883
Interval vectors and matrices.....	2885
Preconditioning.....	2890
Inverting interval matrices.....	2890

Appendix B: Routine and Function Arguments

Vector Arguments in BLAS.....	2893
Vector Arguments in VML.....	2894

Matrix Arguments.....	2895

Appendix C: Code Examples

BLAS Code Examples.....	2901
PARDISO Code Examples.....	2908
Examples for Sparse Symmetric Linear Systems.....	2909
Examples for Sparse Unsymmetric Linear Systems.....	2921
Direct Sparse Solver Code Examples.....	2935
Iterative Sparse Solver Code Examples.....	2949
Fourier Transform Functions Code Examples.....	2997
FFT Code Examples.....	2997
Examples of Using Multi-Threading for FFT Computation.....	3016
Examples for Cluster FFT Functions.....	3022
Auxiliary Data Transformations.....	3026
Interval Linear Solvers Code Examples.....	3029
PDE Support Code Examples.....	3041
Trigonometric Transforms Interface Code Examples.....	3041
Poisson Library Code Examples.....	3060
Preconditioner Code Examples.....	3091

Appendix D: CBLAS Interface to the BLAS

CBLAS Arguments.....	3159
Level 1 CBLAS.....	3160
Level 2 CBLAS.....	3164
Level 3 CBLAS.....	3172
Sparse CBLAS.....	3175

Appendix E: Specific Features of Fortran 95 Interfaces for LAPACK Routines

Interfaces Identical to Netlib.....	3180

Interfaces with Replaced Argument Names.....3183

Modified Netlib Interfaces.....3185

Interfaces Absent From Netlib.....3187

Interfaces of New Functionality.....3193

Appendix F: Optimization Solvers Basics

Nonlinear Least Squares Problem.....3195

Trust-Region Algorithm.....3196

Bibliography

Glossary

Index

Overview

The Intel® Math Kernel Library (Intel® MKL) provides Fortran routines and functions that perform a wide variety of operations on vectors and matrices including sparse matrices and interval matrices. The library also includes fast Fourier transform (FFT) functions, as well as vector mathematical and vector statistical functions with Fortran and C interfaces.

The versions of Intel MKL intended for Windows* and Linux* operating systems also include ScaLAPACK software and Cluster FFT software for solving respective computational problems on distributed-memory parallel computers.

The Intel MKL enhances performance of the application programs that use it because the library has been optimized for latest generations of Intel® processors.

This chapter introduces the Intel Math Kernel Library and provides information about the organization of this manual.

About This Software

The Intel® Math Kernel Library includes the following groups of routines:

- Basic Linear Algebra Subprograms (BLAS):
 - vector operations
 - matrix-vector operations
 - matrix-matrix operations
- Sparse BLAS Level 1, 2, and 3 (basic operations on sparse vectors and matrices)
- LAPACK routines for solving systems of linear equations
- LAPACK routines for solving least squares problems, eigenvalue and singular value problems, and Sylvester's equations
- Auxiliary and utility LAPACK routines
- ScaLAPACK computational, driver and auxiliary routines (only in Intel MKL for Linux* and Windows* operating systems)
- Direct and Iterative Sparse Solver routines
- Vector Mathematical Library (VML) functions for computing core mathematical functions on vector arguments (with Fortran and C interfaces)

- Vector Statistical Library (VSL) functions for generating vectors of pseudorandom numbers with different types of statistical distributions and for performing convolution and correlation computations
- General Fast Fourier Transform (FFT) Functions, providing fast computation of Discrete Fourier Transform via the FFT algorithms and having Fortran and C interfaces
- Cluster FFT functions (only in Intel MKL for Linux* and Windows* operating systems)
- Interval Solver routines for solving systems of interval linear equations
- Tools for solving partial differential equations - trigonometric transform routines and Poisson solver
- Optimization Solver routines for solving nonlinear least squares problems through the Trust-Region (TR) algorithms and computing Jacobi matrix by central differences
- Basic Linear Algebra Communication Subprograms (BLACS) that are used to support a linear algebra oriented message passing interface.
- GMP arithmetic functions

For specific issues on using the library, please refer to the *MKL Release Notes*.

Technical Support

Intel MKL provides a product web site that offers timely and comprehensive product information, including product features, white papers, and technical articles. For the latest information, check: <http://developer.intel.com/software/products/>

Intel also provides a support web site that contains a rich repository of self help information, including getting started tips, known product issues, product errata, license information, user forums, and more (visit <http://support.intel.com/support/performance/tools/libraries/mkl>).

Registering your product entitles you to one year of technical support and product updates through Intel® Premier Support. Intel Premier Support is an interactive issue management and communication web site providing these services:

- Submit issues and review their status.
- Download product updates anytime of the day.

To register your product, contact Intel, or seek product support, please visit: <http://developer.intel.com/software/products/support>.

BLAS Routines

The BLAS routines and functions are divided into the following groups according to the operations they perform:

- [BLAS Level 1 Routines and Functions](#) perform operations of both addition and reduction on vectors of data. Typical operations include scaling and dot products.
- [BLAS Level 2 Routines](#) perform matrix-vector operations, such as matrix-vector multiplication, rank-1 and rank-2 matrix updates, and solution of triangular systems.
- [BLAS Level 3 Routines](#) perform matrix-matrix operations, such as matrix-matrix multiplication, rank-k update, and solution of triangular systems.

Starting from release 8.0, Intel® MKL also supports Fortran 95 interface to the BLAS routines.

Sparse BLAS Routines

The [Sparse BLAS Level 1 Routines and Functions](#) and [Sparse BLAS Level 2 Routines and Level 3 routines](#) and functions operate on sparse vectors and matrices. These routines perform vector operations similar to the BLAS Level 1, 2, and 3 routines. The Sparse BLAS routines take advantage of vector and matrix sparsity: they allow you to store only non-zero elements of vectors and matrices. Intel MKL also supports Fortran 95 interface to Sparse BLAS routines.

LAPACK Routines

The Intel® Math Kernel Library fully supports LAPACK 3.1 set of computational, driver, auxiliary and utility routines.

The original versions of LAPACK from which that part of Intel MKL was derived can be obtained from <http://www.netlib.org/lapack/index.html>. The authors of LAPACK are E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen.

The LAPACK routines can be divided into the following groups according to the operations they perform:

- Routines for solving systems of linear equations, factoring and inverting matrices, and estimating condition numbers (see [Chapter 3](#)).
- Routines for solving least squares problems, eigenvalue and singular value problems, and Sylvester's equations (see [Chapter 4](#)).
- Auxiliary and utility routines used to perform certain subtasks, common low-level computation or related tasks (see [Chapter 5](#)).

Starting from release 8.0, Intel MKL also supports Fortran-95 interface to LAPACK computational and driver routines. This interface provides an opportunity for simplified calls of LAPACK routines with fewer required arguments.

ScaLAPACK Routines

The ScaLAPACK package (included only with the Intel® MKL versions for Linux* and Windows* operating systems, see [Chapter 6](#) and [Chapter 7](#)) runs on distributed-memory architectures and includes routines for solving systems of linear equations, solving linear least squares problems, eigenvalue and singular value problems, as well as performing a number of related computational tasks.

The original versions of ScaLAPACK from which that part of Intel MKL was derived can be obtained from <http://www.netlib.org/scalapack/index.html>. The authors of ScaLAPACK are L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley.

The Intel MKL version of ScaLAPACK is optimized for Intel® processors and uses MPICH version of MPI as well as Intel MPI.

Sparse Solver Routines

Direct sparse solver routines in Intel MKL (see [Chapter 8](#)) solve symmetric and symmetrically-structured sparse matrices with real or complex coefficients. For symmetric matrices, these Intel MKL subroutines can solve both positive-definite and indefinite systems. Intel MKL includes the PARDISO* sparse solver interface as well as an alternative set of user callable direct sparse solver routines.

If you use the sparse solver PARDISO* from Intel MKL, please cite:

O.Schenk and K.Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. J. of Future Generation Computer Systems, 20(3):475-487, 2004.

Intel MKL provides also an iterative sparse solver (see [Chapter 8](#)) that uses Sparse BLAS level 2 and 3 routines and works with different sparse data formats.

VML Functions

The Vector Mathematical Library (VML) functions (see [Chapter 9](#)) include a set of highly optimized implementations of certain computationally expensive core mathematical functions (power, trigonometric, exponential, hyperbolic, etc.) that operate on vectors of real and complex numbers.

Application programs that might significantly improve performance with VML include nonlinear programming software, integrals computation, and many others. VML provides interfaces both for Fortran and C languages.

Statistical Functions

The Vector Statistical Library (VSL) contains two sets of functions (see [Chapter 10](#)):

- The first set includes a collection of pseudo- and quasi-random number generator subroutines implementing basic continuous and discrete distributions. To provide best performance, the VSL subroutines use calls to highly optimized Basic Random Number Generators (BRNGs) and a library of vector mathematical functions.
- The second set includes a collection of routines that implement a wide variety of convolution and correlation operations.

Fourier Transform Functions

The Intel® MKL multidimensional Fast Fourier Transform (FFT) functions with mixed radix support (see [Chapter 11](#)) provide uniformity of discrete Fourier transform computation and combine functionality with ease of use. Both Fortran and C interface specification are given. There is also a cluster version of FFT functions, which runs on distributed-memory architectures and is provided only in Intel MKL versions for the Linux* and Windows* operating systems.

The FFT functions provide fast computation via the FFT algorithms not only for lengths that are powers of 2 but also for 3, 5, 7, 11, and other radices.

Interval Solver Routines

Interval Solver routines included into Intel® MKL (see [Chapter 12](#)) can be used to solve interval systems of linear equations and related problems.

Partial Differential Equations Support

Intel® MKL provides tools for solving Partial Differential Equations (PDE) (see [Chapter 13](#)). These tools are Trigonometric Transform interface routines and Poisson Library.

The Trigonometric Transform routines may be helpful to users who implement their own solvers similar to the solver that the Poisson Library provides. The users can improve performance of their solvers by using fast sine, cosine, and staggered cosine transforms implemented in the Trigonometric Transform interface.

The Poisson Library is designed for fast solving of simple Helmholtz, Poisson, and Laplace problems. The Trigonometric Transform interface, which underlies the solver, is based on the Intel MKL FFT interface (refer to [Chapter 11](#)), optimized for Intel® processors.

Optimization Solver Routines

Intel® MKL provides Optimization Solver routines (see [Chapter 14](#)) that can be used to solve nonlinear least squares problems with or without linear (bound) constraints through the Trust-Region (TR) algorithms and compute Jacobi matrix by central differences.

See also Appendix A [Optimization Solvers Basics](#) for description of the basic notions of optimization solvers, nonlinear least square problems, and the TR algorithm.

Support Functions

The Intel® MKL support functions (see [Chapter 15](#)) are used to support the operation of the Intel MKL software and provide basic information on the library and library operation, such as the current library version, timing, setting and measuring of CPU frequency, error handling, and memory allocation.

Starting from 10.0 version, the Intel MKL support functions provide additional threading control.

BLACS Routines

The Intel® Math Kernel Library implements routines from the BLACS (Basic Linear Algebra Communication Subprograms) package (see [Chapter 16](#)) that are used to support a linear algebra oriented message passing interface that may be implemented efficiently and uniformly across a large range of distributed memory platforms.

The original versions of BLACS from which that part of Intel MKL was derived can be obtained from <http://www.netlib.org/blacs/index.html>. The authors of BLACS are Jack Dongarra and R. Clint Whaley.

GMP Arithmetic Functions

Intel® MKL implementation of GMP arithmetic functions includes arbitrary precision arithmetic operations on integer numbers. The interfaces of such functions fully match the GNU Multiple Precision (GMP) Arithmetic Library.

Performance Enhancements

The Intel® Math Kernel Library has been optimized by exploiting both processor and system features and capabilities. Special care has been given to those routines that most profit from cache-management techniques. These especially include matrix-matrix operation routines such as `dgemm()`.

In addition, code optimization techniques have been applied to minimize dependencies of scheduling integer and floating-point units on the results within the processor.

The major optimization techniques used throughout the library include:

- Loop unrolling to minimize loop management costs
- Blocking of data to improve data reuse opportunities
- Copying to reduce chances of data eviction from cache
- Data prefetching to help hide memory latency
- Multiple simultaneous operations (for example, dot products in `dgemm`) to eliminate stalls due to arithmetic unit pipelines
- Use of hardware features such as the SIMD arithmetic units, where appropriate

These are techniques from which the arithmetic code benefits the most.

Parallelism

In addition to the performance enhancements discussed above, Intel® MKL offers performance gains through parallelism provided by the symmetric multiprocessing performance (SMP) feature. You can obtain improvements from SMP in the following ways:

- One way is based on user-managed threads in the program and further distribution of the operations over the threads based on data decomposition, domain decomposition, control decomposition, or some other parallelizing technique. Each thread can use any of the Intel MKL functions because the library has been designed to be thread-safe.
- Another method is to use the FFT and BLAS level 3 routines. They have been parallelized and require no alterations of your application to gain the performance enhancements of multiprocessing. Performance using multiple processors on the level 3 BLAS shows excellent scaling. Since the threads are called and managed within the library, the application does not need to be recompiled thread-safe (see also [Fortran 95 Interface Conventions](#) in Chapter 2).

- Yet another method is to use *tuned LAPACK routines*. Currently these include the single- and double precision flavors of routines for *QR* factorization of general matrices, triangular factorization of general and symmetric positive-definite matrices, solving systems of equations with such matrices, as well as solving symmetric eigenvalue problems.

For instructions on setting the number of available processors for the BLAS level 3 and LAPACK routines, see *Intel® MKL User's Guide*.

Platforms Supported

The Intel® Math Kernel Library includes Fortran routines and functions optimized for Intel® processor-based computers running operating systems that support multiprocessing. In addition to the Fortran interface, Intel MKL includes a C-language interface for the Discrete Fourier transform functions, as well as for the Vector Mathematical Library and Vector Statistical Library functions. For hardware and software requirements to use Intel MKL, see *Intel® MKL Release Notes*.

About This Manual

This manual describes the routines and functions of Intel® MKL. Each reference section describes a routine group typically consisting of routines used with four basic data types: single-precision real, double-precision real, single-precision complex, and double-precision complex.

Each routine group is introduced by its name, a short description of its purpose, and the calling sequence, or syntax, for each type of data with which each routine of the group is used. The following sections are also included:

Description	Describes the operation performed by routines of the group based on one or more equations. The data types of the arguments are defined in general terms for the group.
Input Parameters	Defines the data type for each parameter on entry, for example: a REAL for saxpy DOUBLE PRECISION for daxpy
Output Parameters	Lists resultant parameters on exit.

The following terms are used in the manual in reference to various operating systems:

Windows* OS	This term refers to information that is valid on all supported Microsoft Windows* operating systems.
Linux* OS	This term refers to information that is valid on all supported Linux* operating systems.

Mac OS* X This term refers to information that is valid on Intel®-based systems running the Mac OS* X operating system.

Audience for This Manual

The manual addresses programmers proficient in computational mathematics and assumes a working knowledge of the principles and vocabulary of linear algebra, mathematical statistics, and Fourier transforms.

Manual Organization

The manual contains the following chapters and appendixes:

Chapter 1	Overview . Introduces the Intel Math Kernel Library software, provides information on manual organization, and explains notational conventions.
Chapter 2	BLAS and Sparse BLAS Routines . Provides descriptions of BLAS and Sparse BLAS functions and routines.
Chapter 3	LAPACK Routines: Linear Equations . Provides descriptions of LAPACK routines for solving systems of linear equations and performing a number of related computational tasks: triangular factorization, matrix inversion, estimating the condition number of matrices.
Chapter 4	LAPACK Routines: Least Squares and Eigenvalue Problems . Provides descriptions of LAPACK routines for solving least squares problems, standard and generalized eigenvalue problems, singular value problems, and Sylvester's equations.
Chapter 5	LAPACK Auxiliary and Utility Routines . Describes auxiliary and utility LAPACK routines that perform certain subtasks or common low-level computation.
Chapter 6	ScaLAPACK Routines . Describes ScaLAPACK computational and driver routines (software included only with Intel MKL for Linux* and Windows* operating systems).
Chapter 7	ScaLAPACK Auxiliary and Utility Routines . Describes ScaLAPACK auxiliary routines (software included only with Intel MKL for Linux* and Windows* operating systems).
Chapter 8	Sparse Solver Routines . Describes direct sparse solver routines that solve symmetric and symmetrically-structured sparse matrices. Also describes the iterative sparse solver routines.
Chapter 9	Vector Mathematical Functions . Provides descriptions of VML functions for computing elementary mathematical functions on vector arguments.

Chapter 10	Statistical Functions . Provides descriptions of VSL functions for generating vectors of pseudorandom numbers and for performing convolution and correlation operations.
Chapter 11	Fourier Transform Functions . Describes multidimensional functions for computing fast Fourier transform and cluster FFT functions (software included only with Intel MKL for the Linux* and Windows* operating systems).
Chapter 12	Interval Linear Solvers . Describes routines that can be used to solve interval systems of linear equations and related problems.
Chapter 13	Partial Differential Equations Support . Describes Trigonometric Transform interface routines and Poisson Library implemented for solving Partial Differential Equations (PDE).
Chapter 14	Optimization Solver Routines . Describes routines for solving optimization problems. These routines solve nonlinear least squares problems through the Trust-Region (TR) algorithms and compute Jacobi matrix by central differences.
Chapter 15	Support Functions . Describes functions that are used to support the operation of Intel MKL software, such as status information functions, timing and error handling functions.
Chapter 16	BLACS Routines . Describes Basic Linear Algebra Communication Subprograms that are used to support a linear algebra oriented message passing interface.
Appendix A	Linear Solvers Basics . Briefly describes the basic definitions and approaches used in linear algebra for solving systems of linear equations. Describes sparse data storage formats, as well as basic concepts of interval arithmetic.
Appendix B	Routine and Function Arguments . Describes the major arguments of the BLAS routines and VML functions: vector and matrix arguments.
Appendix C	Code Examples . Provides code examples of calling various Intel MKL functions and routines (BLAS, PARDISO, Direct and Iterative Sparse Solver, FFT, Cluster FFT, Interval Linear Solvers, Trigonometric Transforms, and Poisson Library).
Appendix D	CBLAS Interface to the BLAS . Provides the C interface to the BLAS.
Appendix E	Specific Features of Fortran 95 Interfaces for LAPACK Routines . Provides the features of Intel MKL Fortran 95 interfaces for LAPACK routines in comparison with Netlib implementation.
Appendix F	Optimization Solvers Basics . Briefly describes the basic notions of optimization solvers, nonlinear least square problem, and Trust Region algorithm.

The manual also includes a [Bibliography](#), [Glossary](#) and an [Index](#).

Notational Conventions

This manual uses the following notational conventions:

- Routine name shorthand (for example, `?ungqr` instead of `cungqr/zungqr`).
- Font conventions used for distinction between the text and the code.

Routine Name Shorthand

For shorthand, names with a question mark “?” in them represent groups of routines with similar functionality. The question mark is used to indicate any or all possible varieties of a function; for example:

`?swap` Refers to all four data types of the vector-vector `?swap` routine: `sswap`, `dswap`, `cswap`, and `zswap`.

Font Conventions

The following font conventions are used:

<code>UPPERCASE COURIER</code>	Data type used in the description of input and output parameters for Fortran interface. For example, <code>CHARACTER*1</code> .
<code>lowercase courier</code>	Code examples: <code>a(k+i,j) = matrix(i,j)</code> and data types for C interface, for example, <code>const float*</code>
<code>lowercase courier</code> <code>mixed with UpperCase</code> <code>courier</code>	Function names for C interface, for example, <code>vmlSetMode</code>
<i>lowercase courier</i> <i>italic</i>	Variables in arguments and parameters description. For example, <i>incx</i> .
*	Used as a multiplication symbol in code examples and equations and where required by the Fortran syntax.

BLAS and Sparse BLAS Routines

2

This chapter contains descriptions of the BLAS and Sparse BLAS routines of the Intel® Math Kernel Library. The routine descriptions are arranged in several sections according to the BLAS level of operation:

- [BLAS Level 1 Routines](#) (vector-vector operations)
- [BLAS Level 2 Routines](#) (matrix-vector operations)
- [BLAS Level 3 Routines](#) (matrix-matrix operations)
- [Sparse BLAS Level 1 Routines](#) (vector-vector operations).
- [Sparse BLAS Level 2 and Level 3 Routines](#) (matrix-vector and matrix-matrix operations)

Each section presents the routine and function group descriptions in alphabetical order by routine or function group name; for example, the `?asum` group, the `?axpy` group. The question mark in the group name corresponds to different character codes indicating the data type (`s`, `d`, `c`, and `z` or their combination); see [Routine Naming Conventions](#).

When BLAS or Sparse BLAS routines encounter an error, they call the error reporting routine `xerbla`.

In BLAS Level 1 groups `i?amax` and `i?amin`, an “i” is placed before the character code and corresponds to the index of an element in the vector. These groups are placed in the end of the BLAS Level 1 section.

BLAS Routines

Routine Naming Conventions

BLAS routine names have the following structure:

`<character> <name> <mod> ( )`

The `<character>` field indicates the data type:

<code>s</code>	real, single precision
<code>c</code>	complex, single precision
<code>d</code>	real, double precision
<code>z</code>	complex, double precision

Some routines and functions can have combined character codes, such as `sc` or `dz`.

For example, the function `scasum` uses a complex input array and returns a real value.

The `<name>` field, in BLAS level 1, indicates the operation type. For example, the BLAS level 1 routines `?dot`, `?rot`, `?swap` compute a vector dot product, vector rotation, and vector swap, respectively.

In BLAS level 2 and 3, `<name>` reflects the matrix argument type:

<code>ge</code>	general matrix
<code>gb</code>	general band matrix
<code>sy</code>	symmetric matrix
<code>sp</code>	symmetric matrix (packed storage)
<code>sb</code>	symmetric band matrix
<code>he</code>	Hermitian matrix
<code>hp</code>	Hermitian matrix (packed storage)
<code>hb</code>	Hermitian band matrix
<code>tr</code>	triangular matrix
<code>tp</code>	triangular matrix (packed storage)
<code>tb</code>	triangular band matrix.

The `<mod>` field, if present, provides additional details of the operation. BLAS level 1 names can have the following characters in the `<mod>` field:

<code>c</code>	conjugated vector
<code>u</code>	unconjugated vector
<code>g</code>	Givens rotation.

BLAS level 2 names can have the following characters in the `<mod>` field:

<code>mv</code>	matrix-vector product
<code>sv</code>	solving a system of linear equations with matrix-vector operations
<code>r</code>	rank-1 update of a matrix
<code>r2</code>	rank-2 update of a matrix.

BLAS level 3 names can have the following characters in the `<mod>` field:

<code>mm</code>	matrix-matrix product
<code>sm</code>	solving a system of linear equations with matrix-matrix operations
<code>rk</code>	rank- k update of a matrix
<code>r2k</code>	rank- $2k$ update of a matrix.

The examples below illustrate how to interpret BLAS routine names:

<code>ddot</code>	<code><d> <dot></code> : double-precision real vector-vector dot product

<code>cdotc</code>	<code><c> <dot> <c></code> : complex vector-vector dot product, conjugated
<code>scasum</code>	<code><sc> <asum></code> : sum of magnitudes of vector elements, single precision real output and single precision complex input
<code>cdotu</code>	<code><c> <dot> <u></code> : vector-vector dot product, unconjugated, complex
<code>sgemv</code>	<code><s> <ge> <mv></code> : matrix-vector product, general matrix, single precision
<code>ztrmm</code>	<code><z> <tr> <mm></code> : matrix-matrix product, triangular matrix, double-precision complex.

Sparse BLAS level 1 naming conventions are similar to those of BLAS level 1. For more information, see [“Naming Conventions”](#).

Fortran 95 Interface Conventions

Fortran 95 interface to BLAS and Sparse BLAS Level 1 routines is implemented through wrappers that call respective FORTRAN 77 routines. This interface uses such features of Fortran 95 as assumed-shape arrays and optional arguments to provide simplified calls to BLAS and Sparse BLAS Level 1 routines with fewer parameters.



NOTE. For BLAS, Intel MKL offers two types of Fortran 95 interfaces:

- using `mkl_blas.fi` only through `include 'mkl_blas_subroutine.fi'` statement. Such interfaces allow you to make use of the original LAPACK routines with all their arguments
 - using `mkl_blas.f90` that includes improved interfaces described below. This file is used to generate the module files `mkl95_blas.mod` and `mkl95_precision.mod`. See also section “Fortran 95 interfaces and wrappers to LAPACK and BLAS” of *Intel® MKL User's Guide* for details. The module files are used to process the FORTRAN use clauses referencing the LAPACK interface: `use mkl95_blas` and `use mkl95_precision`.
-

The main conventions used in Fortran 95 interface are as follows:

- The names of parameters used in Fortran 95 interface are typically the same as those used for the respective generic (FORTRAN 77) interface. However, certain changes of some parameter names are made to reduce their number in the whole library. For BLAS and Sparse BLAS Level 1 routines only one generic parameter name `ap` is changed to `a` in the Fortran 95 interface.

Note that these changes of the formal parameter names have no impact on program semantics and follow the conventions of names unification.

- Some input parameters such as array dimensions are not required in Fortran 95 and are skipped from the calling sequence. Array dimensions are reconstructed from the user data that must exactly follow the required array shape.
- A parameter can be skipped if its value is completely defined by the presence or absence of another parameter in the calling sequence, and the restored value is the only meaningful value for the skipped parameter.
- Parameters specifying the increment values *incx* and *incy* are skipped. In most cases their values are equal to 1. In Fortran 95 an increment with different value can be directly established in the corresponding parameter.
- Some generic parameters are declared as optional in Fortran 95 interface and may or may not be present in the calling sequence. A parameter can be declared optional if it satisfies one of the following conditions:
 1. It can take only a few possible values. The default value of such parameter typically is the first value in the list; all exceptions to this rule are explicitly stated in the routine description.
 2. It has a natural default value.

Optional parameters are given in square brackets in Fortran 95 call syntax.

The particular rules used for reconstructing the values of omitted optional parameters are specific for each routine and are detailed in the respective "Fortran 95 Notes" subsection at the end of routine specification section. If this subsection is omitted, the Fortran 95 interface for the given routine does not differ from the corresponding FORTRAN 77 interface.

Note that this interface is not implemented in the current version of Sparse BLAS Level 2 and Level 3 routines. Fortran 95 interface for each of these routines is given in the "Interfaces - Fortran 95" subsection at the end of the respective routine specification section.

Matrix Storage Schemes

Matrix arguments of BLAS routines can use the following storage schemes:

- *Full storage*: a matrix *A* is stored in a two-dimensional array *a*, with the matrix element a_{ij} stored in the array element $a(i, j)$.
- *Packed storage* scheme allows you to store symmetric, Hermitian, or triangular matrices more compactly: the upper or lower triangle of the matrix is packed by columns in a one-dimensional array.
- *Band storage*: a band matrix is stored compactly in a two-dimensional array: columns of the matrix are stored in the corresponding columns of the array, and *diagonals* of the matrix are stored in rows of the array.

For more information on matrix storage schemes, see “[Matrix Arguments](#)” in Appendix B.

BLAS Level 1 Routines

BLAS Level 1 includes routines and functions, which perform vector-vector operations. [Table 2-1](#) lists the BLAS Level 1 routine and function groups and the data types associated with them.

Table 2-1 BLAS Level 1 Routine Groups and Their Data Types

Routine or Function Group	Data Types	Description
?asum	s, d, sc, dz	Sum of vector magnitudes (functions)
?axpy	s, d, c, z	Scalar-vector product (routines)
?copy	s, d, c, z	Copy vector (routines)
?dot	s, d	Dot product (functions)
?sdot	sd, d	Dot product with extended precision (functions)
?dotc	c, z	Dot product conjugated (functions)
?dotu	c, z	Dot product unconjugated (functions)
?nrm2	s, d, sc, dz	Vector 2-norm (Euclidean norm) (functions)
?rot	s, d, cs, zd	Plane rotation of points (routines)
?rotg	s, d, c, z	Givens rotation of points (routines)
?rotm	s, d	Modified plane rotation of points
?rotmg	s, d	Givens modified plane rotation of points
?scal	s, d, c, z, cs, zd	Vector scaling (routines)
?swap	s, d, c, z	Vector-vector swap (routines)
i?amax	s, d, c, z	Index of the maximum absolute value element of a vector (functions)

Routine or Function Group	Data Types	Description
<code>i?amin</code>	s, d, c, z	Index of the minimum absolute value element of a vector (functions)
<code>dcabs1</code>	d	Absolute value of a double precision complex number

?asum

Computes the sum of magnitudes of the vector elements.

Syntax

FORTRAN 77:

```
res = sasum(n, x, incx)
res = scasum(n, x, incx)
res = dasum(n, x, incx)
res = dzasum(n, x, incx)
```

Fortran 95:

```
res = asum(x)
```

Description

The function `?asum` function computes the sum of the magnitudes of elements of a real vector, or the sum of magnitudes of the real and imaginary parts of elements of a complex vector:

$$res = |Re\ x(1)| + |Im\ x(1)| + |Re\ x(2)| + |Im\ x(2)| + \dots + |Re\ x(n)| + |Im\ x(n)|,$$

where x is a vector of order n .

Input Parameters

n	INTEGER. Specifies the number of elements in vector x .
x	REAL for <code>sasum</code> DOUBLE PRECISION for <code>dasum</code> COMPLEX for <code>scasum</code> DOUBLE COMPLEX for <code>dzasum</code>

incx Array, DIMENSION at least $(1 + (n-1) * \text{abs}(incx))$.
 INTEGER. Specifies the increment for indexing vector *x*.

Output Parameters

res REAL for sasum
 DOUBLE PRECISION for dasum
 REAL for scasum
 DOUBLE PRECISION for dzasum
 Contains the sum of magnitudes of real and imaginary parts
 of all elements of the vector.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `asum` interface are the following:

x Holds the array of size(*n*).

?axpy

Computes a vector-scalar product and adds the result to a vector.

Syntax

FORTRAN 77:

```
call saxpy(n, a, x, incx, y, incy)
call daxpy(n, a, x, incx, y, incy)
call caxpy(n, a, x, incx, y, incy)
call zaxpy(n, a, x, incx, y, incy)
```

Fortran 95:

```
call axpy(x, y [,a])
```

Description

The `?axpy` routines perform a vector-vector operation defined as

$$y := a * x + y$$

where:

a is a scalar

x and y are vectors of order n .

Input Parameters

n	INTEGER. Specifies the number of elements in vectors x and y .
a	REAL for saxpy DOUBLE PRECISION for daxpy COMPLEX for caxpy DOUBLE COMPLEX for zaxpy Specifies the scalar a .
x	REAL for saxpy DOUBLE PRECISION for daxpy COMPLEX for caxpy DOUBLE COMPLEX for zaxpy Array, DIMENSION at least $(1 + (n-1) * \text{abs}(\text{incx}))$.
incx	INTEGER. Specifies the increment for the elements of x .
y	REAL for saxpy DOUBLE PRECISION for daxpy COMPLEX for caxpy DOUBLE COMPLEX for zaxpy Array, DIMENSION at least $(1 + (n-1) * \text{abs}(\text{incy}))$.
incy	INTEGER. Specifies the increment for the elements of y .

Output Parameters

y	Contains the updated vector y .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `axpy` interface are the following:

<i>x</i>	Holds the array of size (<i>n</i>).
<i>y</i>	Holds the array of size (<i>n</i>).
<i>a</i>	The default value is 1.

?copy

Copies vector to another vector.

Syntax

FORTRAN 77:

```
call scopy(n, x, incx, y, incy)
call dcopy(n, x, incx, y, incy)
call ccopy(n, x, incx, y, incy)
call zcopy(n, x, incx, y, incy)
```

Fortran 95:

```
call copy(x, y)
```

Description

The `?copy` routines perform a vector-vector operation defined as

$$y = x,$$

where *x* and *y* are vectors.

Input Parameters

<i>n</i>	INTEGER. Specifies the number of elements in vectors <i>x</i> and <i>y</i> .
<i>x</i>	REAL for <code>scopy</code> DOUBLE PRECISION for <code>dcopy</code>

	COMPLEX for ccopy
	DOUBLE COMPLEX for zcopy
	Array, DIMENSION at least $(1 + (n-1) * \text{abs}(incx))$.
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> .
<i>y</i>	REAL for scopy
	DOUBLE PRECISION for dcopy
	COMPLEX for ccopy
	DOUBLE COMPLEX for zcopy
	Array, DIMENSION at least $(1 + (n-1) * \text{abs}(incy))$.
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> .

Output Parameters

<i>y</i>	Contains a copy of the vector <i>x</i> if <i>n</i> is positive. Otherwise, parameters are unaltered.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `copy` interface are the following:

<i>x</i>	Holds the vector of length(<i>n</i>).
<i>y</i>	Holds the vector of length (<i>n</i>).

?dot

Computes a vector-vector dot product.

Syntax

FORTRAN 77:

```
res = sdot(n, x, incx, y, incy)
res = ddot(n, x, incx, y, incy)
```

Fortran 95:

```
res = dot(x, y)
```


Description

The `?dot` functions perform a vector-vector reduction operation defined as

$$\text{res} = \sum (x * y)$$

where x and y are vectors.

Input Parameters

n	INTEGER. Specifies the number of elements in vectors x and y .
x	REAL for <code>sdot</code> DOUBLE PRECISION for <code>ddot</code> Array, DIMENSION at least $(1 + (n-1) * \text{abs}(\text{incx}))$.
incx	INTEGER. Specifies the increment for the elements of x .
y	REAL for <code>sdot</code> DOUBLE PRECISION for <code>ddot</code> Array, DIMENSION at least $(1 + (n-1) * \text{abs}(\text{incy}))$.
incy	INTEGER. Specifies the increment for the elements of y .

Output Parameters

res	REAL for <code>sdot</code> DOUBLE PRECISION for <code>ddot</code> Contains the result of the dot product of x and y , if n is positive. Otherwise, res contains 0.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `dot` interface are the following:

x	Holds the vector of length(n).
y	Holds the vector of length (n).

?sdot

Computes a vector-vector dot product with extended precision.

Syntax

FORTRAN 77:

```
res = sdsdot(n, sb, sx, incx, sy, incy)
res = dsdot(n, sx, incx, sy, incy)
```

Fortran 95:

```
res = sdot(sx, sy)
res = sdot(sx, sy, sb)
```

Description

The ?sdot functions compute the inner product of two vectors with extended precision. Both functions use extended precision accumulation of the intermediate results, but the function sdsdot outputs the final result in single precision, whereas the function dsdot outputs the double precision result. The function sdsdot also adds scalar value *sb* to the inner product.

Input Parameters

<i>n</i>	INTEGER. Specifies the number of elements in the input vectors <i>sx</i> and <i>sy</i> .
<i>sb</i>	REAL. Single precision scalar to be added to inner product (for the function sdsdot only).
<i>sx, sy</i>	REAL. Arrays, DIMENSION at least $(1+(n-1)*abs(incx))$ and $(1+(n-1)*abs(incy))$, respectively. Contain the input single precision vectors.
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>sx</i> .
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>sy</i> .

Output Parameters

<i>res</i>	REAL for sdsdot DOUBLE PRECISION for dsdot

Contains the result of the dot product of sx and sy (with sb added for `sdsdot`), if n is positive. Otherwise, res contains sb for `sdsdot` and 0 for `dsdot`.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sdot` interface are the following:

sx	Holds the vector of length(n).
sy	Holds the vector of length(n).



NOTE. Note that scalar parameter sb is declared as a required parameter in Fortran 95 interface for the function `sdot` to distinguish between function flavors that output final result in different precision.

?dotc

Computes a dot product of a conjugated vector with another vector.

Syntax

FORTRAN 77:

```
res = cdotc(n, x, incx, y, incy )
```

```
res = zdotc(n, x, incx, y, incy )
```

Fortran 95:

```
res = dotc(x, y)
```

Description

The `?dotc` functions perform a vector-vector operation defined as

$$res = \sum (\text{conjg}(x) * y)$$

where x and y are n -element vectors.

Input Parameters

<i>n</i>	INTEGER. Specifies the number of elements in vectors <i>x</i> and <i>y</i> .
<i>x</i>	COMPLEX for <code>cdotc</code> DOUBLE COMPLEX for <code>zdotc</code> Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incx}))$.
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> .
<i>y</i>	COMPLEX for <code>cdotc</code> DOUBLE COMPLEX for <code>zdotc</code> Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incy}))$.
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> .

Output Parameters

<i>res</i>	COMPLEX for <code>cdotc</code> DOUBLE COMPLEX for <code>zdotc</code> Contains the result of the dot product of the conjugated <i>x</i> and unconjugated <i>y</i> , if <i>n</i> is positive. Otherwise, <i>res</i> contains 0.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `dotc` interface are the following:

<i>x</i>	Holds the vector of length(<i>n</i>).
<i>y</i>	Holds the vector of length (<i>n</i>).

?dotu

Computes a vector-vector dot product.

Syntax

FORTRAN 77:

```
res = cdotu(n, x, incx, y, incy )
```

```
res = zdotu(n, x, incx, y, incy )
```

Fortran 95:

```
res = dotu(x, y)
```

Description

The ?dotu functions perform a vector-vector reduction operation defined as

$$\text{res} = \sum (x^* y)$$

where x and y are n -element complex vectors.

Input Parameters

n	INTEGER. Specifies the number of elements in vectors x and y .
x	COMPLEX for cdotu DOUBLE COMPLEX for zdotu Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incx}))$.
incx	INTEGER. Specifies the increment for the elements of x .
y	COMPLEX for cdotu DOUBLE COMPLEX for zdotu Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incy}))$.
incy	INTEGER. Specifies the increment for the elements of y .

Output Parameters

res	COMPLEX for cdotu DOUBLE COMPLEX for zdotu Contains the result of the dot product of x and y , if n is positive. Otherwise, res contains 0.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `dotu` interface are the following:

x	Holds the vector of length(n).
y	Holds the vector of length (n).

?nrm2

Computes the Euclidean norm of a vector.

Syntax

FORTRAN 77:

```
res = snrm2(n, x, incx )
res = dnrm2(n, x, incx )
res = scnrm2(n, x, incx )
res = dznrm2(n, x, incx )
```

Fortran 95:

```
res = nrm2(x)
```

Description

The ?nrm2 functions perform a vector reduction operation defined as

$$res = ||x||,$$

where:

x is a vector

res is a value containing the Euclidean norm of the elements of x .

Input Parameters

n	INTEGER. Specifies the number of elements in vector x .
x	REAL for snrm2

DOUBLE PRECISION for dnorm2
 COMPLEX for scnorm2
 DOUBLE COMPLEX for dznorm2
Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incx}))$.
incx INTEGER. Specifies the increment for the elements of *x*.

Output Parameters

res REAL for snrm2
 DOUBLE PRECISION for dnorm2
 REAL for scnorm2
 DOUBLE PRECISION for dznorm2
 Contains the Euclidean norm of the vector *x*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `norm2` interface are the following:

x Holds the vector of length (n) .

?rot

Performs rotation of points in the plane.

Syntax

FORTRAN 77:

```
call srot( n, x, incx, y, incy, c, s )
call drot( n, x, incx, y, incy, c, s )
call csrot( n, x, incx, y, incy, c, s )
call zdrot( n, x, incx, y, incy, c, s )
```

Fortran 95:

```
call rot(x, y [,c] [,s])
```

Description

Given two complex vectors x and y , each vector element of these vectors is replaced as follows:

$$x(i) = c*x(i) + s*y(i)$$

$$y(i) = c*y(i) - s*x(i)$$

Input Parameters

n	INTEGER. Specifies the number of elements in vectors x and y .
x	REAL for srot DOUBLE PRECISION for drot COMPLEX for csrot DOUBLE COMPLEX for zdrot Array, DIMENSION at least $(1 + (n-1)*abs(incx))$.
$incx$	INTEGER. Specifies the increment for the elements of x .
y	REAL for srot DOUBLE PRECISION for drot COMPLEX for csrot DOUBLE COMPLEX for zdrot Array, DIMENSION at least $(1 + (n-1)*abs(incy))$.
$incy$	INTEGER. Specifies the increment for the elements of y .
c	REAL for srot DOUBLE PRECISION for drot REAL for csrot DOUBLE PRECISION for zdrot A scalar.
s	REAL for srot DOUBLE PRECISION for drot REAL for csrot DOUBLE PRECISION for zdrot A scalar.

Output Parameters

x	Each element is replaced by $c*x + s*y$.
y	Each element is replaced by $c*y - s*x$.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `rot` interface are the following:

<code>x</code>	Holds the vector of length(n).
<code>y</code>	Holds the vector of length (n).
<code>c</code>	The default value is 1.
<code>s</code>	The default value is 1.

?rotg

Computes the parameters for a Givens rotation.

Syntax

FORTRAN 77:

```
call srotg(a, b, c, s )
call drotg(a, b, c, s )
call crotg(a, b, c, s )
call zrotg(a, b, c, s )
```

Fortran 95:

```
call rotg(a, b, c, s)
```

Description

Given the cartesian coordinates (a, b) of a point p , these routines return the parameters a , b , c , and s associated with the Givens rotation that zeros the y -coordinate of the point.

See a more accurate LAPACK version [?lartg](#).

Input Parameters

<code>a</code>	REAL for srotg DOUBLE PRECISION for drotg COMPLEX for crotg

	DOUBLE COMPLEX for zrotg Provides the x -coordinate of the point p .
b	REAL for srotg DOUBLE PRECISION for drotg COMPLEX for crotg DOUBLE COMPLEX for zrotg Provides the y -coordinate of the point p .

Output Parameters

a	Contains the parameter r associated with the Givens rotation.
b	Contains the parameter z associated with the Givens rotation.
c	REAL for srotg DOUBLE PRECISION for drotg REAL for crotg DOUBLE PRECISION for zrotg Contains the parameter c associated with the Givens rotation.
s	REAL for srotg DOUBLE PRECISION for drotg COMPLEX for crotg DOUBLE COMPLEX for zrotg Contains the parameter s associated with the Givens rotation.

?rotm

Performs rotation of points in the modified plane.

Syntax

FORTRAN 77:

```
call srotm(n, x, incx, y, incy, param)
call drotm(n, x, incx, y, incy, param)
```

Fortran 95:

```
call rotm(x, y [,param])
```

Description

Given two complex vectors x and y , each vector element of these vectors is replaced as follows:

$$x(i) = H*x(i) + H*y(i)$$

$$y(i) = H*y(i) - H*x(i)$$

where:

H is a modified Givens transformation matrix whose values are stored in the $param(2)$ through $param(5)$ array. See discussion on the $param$ argument.

Input Parameters

n	INTEGER. Specifies the number of elements in vectors x and y .
x	REAL for srotm DOUBLE PRECISION for drotm Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incx}))$.
incx	INTEGER. Specifies the increment for the elements of x .
y	REAL for srotm DOUBLE PRECISION for drotm Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incy}))$.
incy	INTEGER. Specifies the increment for the elements of y .
$param$	REAL for srotm DOUBLE PRECISION for drotm Array, DIMENSION 5. The elements of the $param$ array are: $param(1)$ contains a switch, $flag$. $param(2-5)$ contain $h11$, $h21$, $h12$, and $h22$, respectively, the components of the array H . Depending on the values of $flag$, the components of H are set as follows:

$$flag = -1. : H = \begin{bmatrix} h11 & h12 \\ h21 & h22 \end{bmatrix}$$

$$flag = 0. : H = \begin{bmatrix} 1. & h12 \\ h21 & 1. \end{bmatrix}$$

$$flag = 1. : H = \begin{bmatrix} h11 & 1. \\ -1 & h22 \end{bmatrix}$$

$$flag = -2. : H = \begin{bmatrix} 1. & 0. \\ 0. & 1. \end{bmatrix}$$

In the last three cases, the matrix entries of 1., -1., and 0. are assumed based on the value of *flag* and are not required to be set in the *param* vector.

Output Parameters

<i>x</i>	Each element is replaced by $h11*x + h12*y$.
<i>y</i>	Each element is replaced by $h21*x + h22*y$.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `rotm` interface are the following:

<i>x</i>	Holds the vector of length(<i>n</i>).
<i>y</i>	Holds the vector of length (<i>n</i>).
<i>param</i>	The default value for <i>param</i> (1) is -2.

?rotmg

Computes parameters for a modified Givens rotation.

Syntax

FORTRAN 77:

```
call srotmg(d1, d2, x1, y1, param)
call drotmg(d1, d2, x1, y1, param)
```

Fortran 95:

```
call rotmg(x1, y1, param [, d1] [d2])
```

Description

Given cartesian coordinates $(x1, y1)$ of an input vector, these routines compute the components of a modified Givens transformation matrix H that zeros the y -component of the resulting vector:

$$\begin{bmatrix} x \\ 0 \end{bmatrix} = H \begin{bmatrix} \text{sqrt}(d_1)x_1 \\ \text{sqrt}(d_2)y_1 \end{bmatrix}$$

Input Parameters

$d1$	REAL for srotmg DOUBLE PRECISION for drotmg Provides the scaling factor for the x -coordinate of the input vector.
$d2$	REAL for srotmg DOUBLE PRECISION for drotmg Provides the scaling factor for the y -coordinate of the input vector.
$x1$	REAL for srotmg DOUBLE PRECISION for drotmg Provides the x -coordinate of the input vector.
$y1$	REAL for srotmg DOUBLE PRECISION for drotmg

Provides the y -coordinate of the input vector.

Output Parameters

<i>d1</i>	REAL for srotmg DOUBLE PRECISION for drotmg Provides the first diagonal element of the updated matrix.
<i>d2</i>	REAL for srotmg DOUBLE PRECISION for drotmg Provides the second diagonal element of the updated matrix.
<i>x1</i>	REAL for srotmg DOUBLE PRECISION for drotmg Provides the x-coordinate of the rotated vector before scaling.
<i>param</i>	REAL for srotmg DOUBLE PRECISION for drotmg Array, DIMENSION 5. The elements of the <i>param</i> array are: <i>param</i>(1) contains a switch, <i>flag</i>. <i>param</i>(2-5) contain <i>h11</i>, <i>h21</i>, <i>h12</i>, and <i>h22</i>, respectively, the components of the array <i>H</i>. Depending on the values of <i>flag</i>, the components of <i>H</i> are set as follows:

$$flag = -1. : H = \begin{bmatrix} h11 & h12 \\ h21 & h22 \end{bmatrix}$$

$$flag = 0. : H = \begin{bmatrix} 1. & h12 \\ h21 & 1. \end{bmatrix}$$

$$flag = 1. : H = \begin{bmatrix} h11 & 1. \\ -1 & h22 \end{bmatrix}$$

$$flag = -2. : H = \begin{bmatrix} 1. & 0. \\ 0. & 1. \end{bmatrix}$$

In the last three cases, the matrix entries of 1., -1., and 0. are assumed based on the value of *flag* and are not required to be set in the *param* vector.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `rotmg` interface are the following:

<i>d1</i>	The default value is 1.
<i>d2</i>	The default value is 1.

?scal

Computes a vector by a scalar product.

Syntax

FORTRAN 77:

```
call sscal(n, a, x, incx )
call dscal(n, a, x, incx )
call cscal(n, a, x, incx )
call zscal(n, a, x, incx )
call csscal(n, a, x, incx )
call zdscal(n, a, x, incx )
```

Fortran 95:

```
call scal(x, a)
```

Description

The `?scal` routines perform a vector-vector operation defined as

$$x = a * x$$

where:

a is a scalar, x is an n -element vector.

Input Parameters

n	INTEGER. Specifies the number of elements in vector x .
a	REAL for <code>sscal</code> and <code>csscal</code> DOUBLE PRECISION for <code>dscal</code> and <code>zdsal</code> COMPLEX for <code>cscal</code> DOUBLE COMPLEX for <code>zscal</code> Specifies the scalar a .
x	REAL for <code>sscal</code> DOUBLE PRECISION for <code>dscal</code> COMPLEX for <code>cscal</code> and <code>csscal</code> DOUBLE COMPLEX for <code>zscal</code> and <code>zdsal</code> Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incx}))$.
incx	INTEGER. Specifies the increment for the elements of x .

Output Parameters

x	Updated vector x .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `scal` interface are the following:

x	Holds the vector of length(n).

?swap

Swaps a vector with another vector.

Syntax

FORTRAN 77:

```
call sswap(n, x, incx, y, incy)
call dswap(n, x, incx, y, incy)
call cswap(n, x, incx, y, incy)
call zswap(n, x, incx, y, incy)
```

Fortran 95:

```
call swap(x, y)
```

Description

Given two vectors x and y , the ?swap routines return vectors y and x swapped, each replacing the other.

Input Parameters

n	INTEGER. Specifies the number of elements in vectors x and y .
x	REAL for sswap DOUBLE PRECISION for dswap COMPLEX for cswap DOUBLE COMPLEX for zswap Array, DIMENSION at least $(1 + (n-1)*abs(incx))$.
$incx$	INTEGER. Specifies the increment for the elements of x .
y	REAL for sswap DOUBLE PRECISION for dswap COMPLEX for cswap DOUBLE COMPLEX for zswap Array, DIMENSION at least $(1 + (n-1)*abs(incy))$.
$incy$	INTEGER. Specifies the increment for the elements of y .

Output Parameters

x	Contains the resultant vector x , that is, the input vector y .
y	Contains the resultant vector y , that is, the input vector x .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `swap` interface are the following:

x	Holds the vector of length(n).
y	Holds the vector of length(n).

i?amax

Finds the index of the element with maximum absolute value.

Syntax

FORTRAN 77:

```
index = isamax(n, x, incx )
index = idamax(n, x, incx )
index = icamax(n, x, incx )
index = izamax(n, x, incx )
```

Fortran 95:

```
index = iamax(x)
```

Description

Given a vector x , the `i?amax` functions return the position of the vector element $x(i)$ that has the largest absolute value for real flavors, or the largest sum $|\text{Re}(x(i))| + |\text{Im}(x(i))|$ for complex flavors.

If n is not positive, 0 is returned.

If more than one vector element is found with the same largest absolute value, the index of the first one encountered is returned.

Input Parameters

<i>n</i>	INTEGER. Specifies the order of the vector <i>x</i> .
<i>x</i>	REAL for <code>isamax</code> DOUBLE PRECISION for <code>idamax</code> COMPLEX for <code>icamax</code> DOUBLE COMPLEX for <code>izamax</code> Array, DIMENSION at least $(1 + (n-1) * \text{abs}(\text{incx}))$.
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> .

Output Parameters

<i>index</i>	INTEGER. Contains the position of vector element <i>x</i> that has the largest absolute value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `iamax` interface are the following:

<i>x</i>	Holds the vector of length (<i>n</i>).

i?amin

Finds the index of the element with the smallest absolute value.

Syntax

FORTRAN 77:

```
index = isamin(n, x, incx)
index = idamin(n, x, incx)
index = icamin(n, x, incx)
index = izamin(n, x, incx)
```

Fortran 95:

```
index = iamin(x)
```

Description

Given a vector x , the `i?amin` functions return the position of the vector element $x(i)$ that has the smallest absolute value for real flavors, or the smallest sum $|\text{Re}(x(i))| + |\text{Im}(x(i))|$ for complex flavors.

If n is not positive, 0 is returned.

If more than one vector element is found with the same smallest absolute value, the index of the first one encountered is returned.

Input Parameters

n	INTEGER. On entry, n specifies the order of the vector x .
x	REAL for <code>isamin</code> DOUBLE PRECISION for <code>idamin</code> COMPLEX for <code>icamin</code> DOUBLE COMPLEX for <code>izamin</code> Array, DIMENSION at least $(1+(n-1)*\text{abs}(\text{incx}))$.
incx	INTEGER. Specifies the increment for the elements of x .

Output Parameters

index	INTEGER. Contains the position of vector element x that has the smallest absolute value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `iamin` interface are the following:

x	Holds the vector of length (n) .

dcabs1

Computes absolute value of double complex number.

Syntax

FORTRAN 77:

```
res = dcabs1(z)
```

Fortran 95:

```
res = dcabs1(z)
```

Description

The `dcabs1` is an auxiliary routine for a few BLAS Level 1 routines. This function performs an operation defined as

$res = |\operatorname{Re}(z)| + |\operatorname{Im}(z)|$,

where z is a scalar and res is a value containing the absolute value of a double complex number z .

Input Parameters

z DOUBLE COMPLEX scalar.

Output Parameters

res DOUBLE PRECISION.
Contains the absolute value of a double complex number z .

BLAS Level 2 Routines

This section describes BLAS Level 2 routines, which perform matrix-vector operations. [Table 2-2](#) lists the BLAS Level 2 routine groups and the data types associated with them.

Table 2-2 BLAS Level 2 Routine Groups and Their Data Types

Routine Groups	Data Types	Description
?gbmv	s, d, c, z	Matrix-vector product using a general band matrix
?gemv	s, d, c, z	Matrix-vector product using a general matrix

Routine Groups	Data Types	Description
?ger	s, d	Rank-1 update of a general matrix
?gerc	c, z	Rank-1 update of a conjugated general matrix
?geru	c, z	Rank-1 update of a general matrix, unconjugated
?hbmV	c, z	Matrix-vector product using a Hermitian band matrix
?hemv	c, z	Matrix-vector product using a Hermitian matrix
?her	c, z	Rank-1 update of a Hermitian matrix
?her2	c, z	Rank-2 update of a Hermitian matrix
?hpmv	c, z	Matrix-vector product using a Hermitian packed matrix
?hpr	c, z	Rank-1 update of a Hermitian packed matrix
?hpr2	c, z	Rank-2 update of a Hermitian packed matrix
?sbmv	s, d	Matrix-vector product using symmetric band matrix
?spmv	s, d	Matrix-vector product using a symmetric packed matrix
?spr	s, d	Rank-1 update of a symmetric packed matrix
?spr2	s, d	Rank-2 update of a symmetric packed matrix
?symv	s, d	Matrix-vector product using a symmetric matrix
?syr	s, d	Rank-1 update of a symmetric matrix
?syr2	s, d	Rank-2 update of a symmetric matrix
?tbmv	s, d, c, z	Matrix-vector product using a triangular band matrix
?tbsv	s, d, c, z	Solution of a linear system of equations with a triangular band matrix

Routine Groups	Data Types	Description
?tpmv	s, d, c, z	Matrix-vector product using a triangular packed matrix
?tpsv	s, d, c, z	Solution of a linear system of equations with a triangular packed matrix
?trmv	s, d, c, z	Matrix-vector product using a triangular matrix
?trsv	s, d, c, z	Solution of a linear system of equations with a triangular matrix

?gbmv

Computes a matrix-vector product using a general band matrix

Syntax

FORTRAN 77:

```
call sgbmv(trans, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy )
call dgbmv(trans, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy )
call cgbmv(trans, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy )
call zgbmv(trans, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy )
```

Fortran 95:

```
call gbmv(a, x, y [,kl] [,m] [,alpha] [,beta] [,trans])
```

Description

The ?gbmv routines perform a matrix-vector operation defined as

```
y := alpha*A*x + beta*y
```

or

```
y := alpha*A'*x + beta*y,
```

or

```
y := alpha *conjg(A')*x + beta*y,
```

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*n* band matrix, with *kl* sub-diagonals and *ku* super-diagonals.

Input Parameters

<i>trans</i>	CHARACTER*1. Specifies the operation: If <i>trans</i>= 'N' or 'n', then $y := \alpha A * x + \beta y$ If <i>trans</i>= 'T' or 't', then $y := \alpha A' * x + \beta y$ If <i>trans</i>= 'C' or 'c', then $y := \alpha \text{conjg}(A') * x + \beta y$
<i>m</i>	INTEGER. Specifies the number of rows of the matrix <i>A</i>. The value of <i>m</i> must be at least zero.
<i>n</i>	INTEGER. Specifies the number of columns of the matrix <i>A</i>. The value of <i>n</i> must be at least zero.
<i>kl</i>	INTEGER. Specifies the number of sub-diagonals of the matrix <i>A</i>. The value of <i>kl</i> must satisfy $0 \leq kl$.
<i>ku</i>	INTEGER. Specifies the number of super-diagonals of the matrix <i>A</i>. The value of <i>ku</i> must satisfy $0 \leq ku$.
<i>alpha</i>	REAL for sgbmv DOUBLE PRECISION for dgbmv COMPLEX for cgbmv DOUBLE COMPLEX for zgbmv Specifies the scalar <i>alpha</i>.
<i>a</i>	REAL for sgbmv DOUBLE PRECISION for dgbmv COMPLEX for cgbmv DOUBLE COMPLEX for zgbmv Array, DIMENSION (<i>lda</i>, <i>n</i>). Before entry, the leading $(kl + ku + 1)$ by <i>n</i> part of the array <i>a</i> must contain the matrix of coefficients. This matrix must be supplied column-by-column, with the leading diagonal of the matrix in row $(ku + 1)$ of the array, the first super-diagonal starting at position 2 in row <i>ku</i>, the first

sub-diagonal starting at position 1 in row $(ku + 2)$, and so on. Elements in the array a that do not correspond to elements in the band matrix (such as the top left ku by ku triangle) are not referenced.

The following program segment transfers a band matrix from conventional full matrix storage to band storage:

```

do 20, j = 1, n
    k = ku + 1 - j
    do 10, i = max(1, j-ku), min(m, j+kl)
        a(k+i, j) = matrix(i,j)
    10 continue
20 continue

```

lda INTEGER. Specifies the first dimension of a as declared in the calling (sub)program. The value of lda must be at least $(kl + ku + 1)$.

x REAL for sgbmv
DOUBLE PRECISION for dgbmv
COMPLEX for cgbmv
DOUBLE COMPLEX for zgbmv
Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incx}))$ when $\text{trans} = 'N'$ or $'n'$, and at least $(1 + (m - 1) * \text{abs}(\text{incx}))$ otherwise. Before entry, the array x must contain the vector x .

incx INTEGER. Specifies the increment for the elements of x . incx must not be zero.

beta REAL for sgbmv
DOUBLE PRECISION for dgbmv
COMPLEX for cgbmv
DOUBLE COMPLEX for zgbmv
Specifies the scalar beta. When beta is equal to zero, then y need not be set on input.

y REAL for sgbmv
DOUBLE PRECISION for dgbmv
COMPLEX for cgbmv

DOUBLE COMPLEX for `zgbmv`

Array, DIMENSION at least $(1 + (m - 1) * \text{abs}(\text{incy}))$ when `trans = 'N'` or `'n'` and at least $(1 + (n - 1) * \text{abs}(\text{incy}))$ otherwise. Before entry, the incremented array `y` must contain the vector `y`.

`incy`

INTEGER. Specifies the increment for the elements of `y`. The value of `incy` must not be zero.

Output Parameters

`y`

Updated vector `y`.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gbmv` interface are the following:

`a`

Holds the array `a` of size $(kl+ku+1, n)$. Contains a banded matrix $m*n$ with `kl` lower diagonal and `ku` upper diagonal.

`x`

Holds the vector of length (rx) , where $rx = n$ if `trans = 'N'`, $rx = m$ otherwise.

`y`

Holds the vector of length (ry) , where $ry = m$ if `trans = 'N'`, $ry = n$ otherwise.

`trans`

Must be `'N'`, `'C'`, or `'T'`.
The default value is `'N'`.

`kl`

If omitted, assumed $kl = ku$, that is, the number of lower diagonals equals the number of the upper diagonals.

`ku`

Restored as $ku = lda - kl - 1$, where `lda` is the first dimension of matrix `A`.

`m`

If omitted, assumed $m = n$, that is, a square matrix.

`alpha`

The default value is 1.

`beta`

The default value is 1.

?gemv

Computes a matrix-vector product using a general matrix

Syntax

FORTRAN 77:

```
call sgemv( trans, m, n, alpha, a, lda, x, incx, beta, y, incy )
call dgemv( trans, m, n, alpha, a, lda, x, incx, beta, y, incy )
call cgemv( trans, m, n, alpha, a, lda, x, incx, beta, y, incy )
call zgemv( trans, m, n, alpha, a, lda, x, incx, beta, y, incy )
```

Fortran 95:

```
call gemv( a, x, y [,alpha][,beta] [,trans] )
```

Description

The ?gemv routines perform a matrix-vector operation defined as

$y := \alpha A x + \beta y,$

or

$y := \alpha A' x + \beta y,$

or

$y := \alpha \text{conjg}(A') x + \beta y,$

where:

α and β are scalars,

x and y are vectors,

A is an m -by- n matrix.

Input Parameters

trans CHARACTER*1. Specifies the operation:
 if *trans*= 'N' or 'n', then $y := \alpha A x + \beta y$;
 if *trans*= 'T' or 't', then $y := \alpha A' x + \beta y$;

	if <i>trans</i> = 'C' or 'c', then $y := \alpha * \text{conjg}(A') * x + \beta * y$.
<i>m</i>	INTEGER. Specifies the number of rows of the matrix <i>A</i> . The value of <i>m</i> must be at least zero.
<i>n</i>	INTEGER. Specifies the number of columns of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for sgemv DOUBLE PRECISION for dgemv COMPLEX for cgemv DOUBLE COMPLEX for zgemv Specifies the scalar <i>alpha</i> .
<i>a</i>	REAL for sgemv DOUBLE PRECISION for dgemv COMPLEX for cgemv DOUBLE COMPLEX for zgemv Array, DIMENSION (<i>lda</i> , <i>n</i>). Before entry, the leading <i>m</i> -by- <i>n</i> part of the array <i>a</i> must contain the matrix of coefficients.
<i>lda</i>	INTEGER. Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, m)$.
<i>x</i>	REAL for sgemv DOUBLE PRECISION for dgemv COMPLEX for cgemv DOUBLE COMPLEX for zgemv Array, DIMENSION at least $(1 + (n-1) * \text{abs}(\text{incx}))$ when <i>trans</i> = 'N' or 'n' and at least $(1 + (m - 1) * \text{abs}(\text{incx}))$ otherwise. Before entry, the incremented array <i>x</i> must contain the vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>beta</i>	REAL for sgemv DOUBLE PRECISION for dgemv COMPLEX for cgemv DOUBLE COMPLEX for zgemv

	Specifies the scalar <i>beta</i> . When <i>beta</i> is set to zero, then <i>y</i> need not be set on input.
<i>y</i>	REAL for sgemv DOUBLE PRECISION for dgemv COMPLEX for cgemv DOUBLE COMPLEX for zgemv Array, DIMENSION at least $(1 + (m - 1) * \text{abs}(\text{incy}))$ when <i>trans</i> = 'N' or 'n' and at least $(1 + (n - 1) * \text{abs}(\text{incy}))$ otherwise. Before entry with non-zero <i>beta</i> , the incremented array <i>y</i> must contain the vector <i>y</i> .
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.

Output Parameters

<i>y</i>	Updated vector <i>y</i> .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gemv` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (m,n) .
<i>x</i>	Holds the vector of length (rx) where $rx = n$ if <i>trans</i> = 'N', $rx = m$ otherwise.
<i>y</i>	Holds the vector of length (ry) where $ry = m$ if <i>trans</i> = 'N', $ry = n$ otherwise.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 1.

?ger

Performs a rank-1 update of a general matrix.

Syntax

FORTRAN 77:

```
call sger(m, n, alpha, x, incx, y, incy, a, lda )
call dger(m, n, alpha, x, incx, y, incy, a, lda )
```

Fortran 95:

```
call ger(a, x, y [,alpha])
```

Description

The ?ger routines perform a matrix-vector operation defined as

$$A := \alpha x y^T + A,$$

where:

α is a scalar,

x is an m -element vector,

y is an n -element vector,

A is an m -by- n matrix.

Input Parameters

m	INTEGER. Specifies the number of rows of the matrix A . The value of m must be at least zero.
n	INTEGER. Specifies the number of columns of the matrix A . The value of n must be at least zero.
α	REAL for sger DOUBLE PRECISION for dger Specifies the scalar α .
x	REAL for sger DOUBLE PRECISION for dger

	Array, DIMENSION at least $(1 + (m - 1) * \text{abs}(incx))$. Before entry, the incremented array x must contain the m -element vector x .
$incx$	INTEGER. Specifies the increment for the elements of x . The value of $incx$ must not be zero.
y	REAL for sger DOUBLE PRECISION for dger Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(incy))$. Before entry, the incremented array y must contain the n -element vector y .
$incy$	INTEGER. Specifies the increment for the elements of y . The value of $incy$ must not be zero.
a	REAL for sger DOUBLE PRECISION for dger Array, DIMENSION (lda, n) . Before entry, the leading m -by- n part of the array a must contain the matrix of coefficients.
lda	INTEGER. Specifies the first dimension of a as declared in the calling (sub)program. The value of lda must be at least $\max(1, m)$.

Output Parameters

a	Overwritten by the updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ger` interface are the following:

a	Holds the matrix A of size (m, n) .
x	Holds the vector of length (m) .
y	Holds the vector of length (n) .
$alpha$	The default value is 1.

?gerc

Performs a rank-1 update (conjugated) of a general matrix.

Syntax

FORTRAN 77:

```
call cgerc(m, n, alpha, x, incx, y, incy, a, lda )
call zgerc(m, n, alpha, x, incx, y, incy, a, lda )
```

Fortran 95:

```
call gerc(a, x, y [,alpha])
```

Description

The ?gerc routines perform a matrix-vector operation defined as

$$A := \alpha x \text{conjg}(y') + A,$$

where:

alpha is a scalar,

x is an *m*-element vector,

y is an *n*-element vector,

A is an *m*-by-*n* matrix.

Input Parameters

<i>m</i>	INTEGER. Specifies the number of rows of the matrix <i>A</i> . The value of <i>m</i> must be at least zero.
<i>n</i>	INTEGER. Specifies the number of columns of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	COMPLEX for cgerc DOUBLE COMPLEX for zgerc Specifies the scalar <i>alpha</i> .
<i>x</i>	COMPLEX for cgerc DOUBLE COMPLEX for zgerc

	Array, DIMENSION at least $(1 + (m - 1) * \text{abs}(incx))$. Before entry, the incremented array x must contain the m -element vector x .
$incx$	INTEGER. Specifies the increment for the elements of x . The value of $incx$ must not be zero.
y	COMPLEX for <code>cgerc</code> DOUBLE COMPLEX for <code>zgerc</code> Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(incy))$. Before entry, the incremented array y must contain the n -element vector y .
$incy$	INTEGER. Specifies the increment for the elements of y . The value of $incy$ must not be zero.
a	COMPLEX for <code>cgerc</code> DOUBLE COMPLEX for <code>zgerc</code> Array, DIMENSION (lda, n) . Before entry, the leading m -by- n part of the array a must contain the matrix of coefficients.
lda	INTEGER. Specifies the first dimension of a as declared in the calling (sub)program. The value of lda must be at least $\max(1, m)$.

Output Parameters

a	Overwritten by the updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gerc` interface are the following:

a	Holds the matrix A of size (m, n) .
x	Holds the vector of length (m) .
y	Holds the vector of length (n) .
$alpha$	The default value is 1.

?geru

Performs a rank-1 update (unconjugated) of a general matrix.

Syntax

FORTRAN 77:

```
call cgeru(m, n, alpha, x, incx, y, incy, a, lda )
call zgeru(m, n, alpha, x, incx, y, incy, a, lda )
```

Fortran 95:

```
call geru(a, x, y [,alpha])
```

Description

The ?geru routines perform a matrix-vector operation defined as

$$A := \alpha x y^T + A,$$

where:

alpha is a scalar,

x is an *m*-element vector,

y is an *n*-element vector,

A is an *m*-by-*n* matrix.

Input Parameters

<i>m</i>	INTEGER. Specifies the number of rows of the matrix <i>A</i> . The value of <i>m</i> must be at least zero.
<i>n</i>	INTEGER. Specifies the number of columns of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	COMPLEX for cgeru DOUBLE COMPLEX for zgeru Specifies the scalar <i>alpha</i> .
<i>x</i>	COMPLEX for cgeru DOUBLE COMPLEX for zgeru

	Array, DIMENSION at least $(1 + (m - 1) * \text{abs}(incx))$. Before entry, the incremented array x must contain the m -element vector x .
$incx$	INTEGER. Specifies the increment for the elements of x . The value of $incx$ must not be zero.
y	COMPLEX for cgeru DOUBLE COMPLEX for zgeru Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(incy))$. Before entry, the incremented array y must contain the n -element vector y .
$incy$	INTEGER. Specifies the increment for the elements of y . The value of $incy$ must not be zero.
a	COMPLEX for cgeru DOUBLE COMPLEX for zgeru Array, DIMENSION (lda, n) . Before entry, the leading m -by- n part of the array a must contain the matrix of coefficients.
lda	INTEGER. Specifies the first dimension of a as declared in the calling (sub)program. The value of lda must be at least $\max(1, m)$.

Output Parameters

a	Overwritten by the updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `geru` interface are the following:

a	Holds the matrix A of size (m, n) .
x	Holds the vector of length (m) .
y	Holds the vector of length (n) .
$alpha$	The default value is 1.

?hbmV

Computes a matrix-vector product using a Hermitian band matrix.

Syntax

FORTRAN 77:

```
call chbmV(uplo, n, k, alpha, a, lda, x, incx, beta, y, incy )
call zhbmV(uplo, n, k, alpha, a, lda, x, incx, beta, y, incy )
```

Fortran 95:

```
call hbmV(a, x, y [,uplo][,alpha] [,beta])
```

Description

The ?hbmV routines perform a matrix-vector operation defined as

$$y := \alpha A x + \beta y,$$

where:

alpha and *beta* are scalars,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* Hermitian band matrix, with *k* super-diagonals.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian band matrix <i>A</i> is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the matrix <i>A</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the matrix <i>A</i> is used.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>k</i>	INTEGER. Specifies the number of super-diagonals of the matrix <i>A</i> . The value of <i>k</i> must satisfy $0 \leq k$.
<i>alpha</i>	COMPLEX for chbmV

a

DOUBLE COMPLEX for zhbmv

Specifies the scalar *alpha*.

COMPLEX for chbmvm

DOUBLE COMPLEX for zhbmv

Array, DIMENSION (*lda*, *n*).

Before entry with *uplo* = 'U' or 'u', the leading (*k* + 1) by *n* part of the array *a* must contain the upper triangular band part of the Hermitian matrix. The matrix must be supplied column-by-column, with the leading diagonal of the matrix in row (*k* + 1) of the array, the first super-diagonal starting at position 2 in row *k*, and so on. The top left *k* by *k* triangle of the array *a* is not referenced. The following program segment transfers the upper triangular part of a Hermitian band matrix from conventional full matrix storage to band storage:

```

do 20, j = 1, n
    m = k + 1 - j
    do 10, i = max(1, j - k), j
        a(m + i, j) = matrix(i, j)
    10 continue
20 continue

```

Before entry with *uplo* = 'L' or 'l', the leading (*k* + 1) by *n* part of the array *a* must contain the lower triangular band part of the Hermitian matrix, supplied column-by-column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right *k* by *k* triangle of the array *a* is not referenced.

The following program segment transfers the lower triangular part of a Hermitian band matrix from conventional full matrix storage to band storage:

```
do 20, j = 1, n
    m = 1 - j
    do 10, i = j, min( n, j + k )
        a( m + i, j ) = matrix( i, j )
    10 continue
20 continue
```

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

<i>lda</i>	INTEGER. Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $(k + 1)$.
<i>x</i>	COMPLEX for chbmv DOUBLE COMPLEX for zhbmv Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>beta</i>	COMPLEX for chbmv DOUBLE COMPLEX for zhbmv Specifies the scalar <i>beta</i> .
<i>y</i>	COMPLEX for chbmv DOUBLE COMPLEX for zhbmv Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array <i>y</i> must contain the vector <i>y</i> .
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.

Output Parameters

<i>y</i>	Overwritten by the updated vector <i>y</i> .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hbmV` interface are the following:

<i>a</i>	Holds the array <i>a</i> of size $(k+1, n)$.
<i>x</i>	Holds the vector of length (n) .
<i>y</i>	Holds the vector of length (n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 1.

?hemv

Computes a matrix-vector product using a Hermitian matrix.

Syntax

FORTRAN 77:

```
call chemv(uplo, n, alpha, a, lda, x, incx, beta, y, incy )
call zhemv(uplo, n, alpha, a, lda, x, incx, beta, y, incy )
```

Fortran 95:

```
call hemv(a, x, y [,uplo][,alpha] [,beta])
```

Description

The `?hemv` routines perform a matrix-vector operation defined as

$$y := \alpha A x + \beta y,$$

where:

alpha and *beta* are scalars,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* Hermitian matrix.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>a</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangular of the array <i>a</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular of the array <i>a</i> is used.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i>. The value of <i>n</i> must be at least zero.
<i>alpha</i>	COMPLEX for chemv DOUBLE COMPLEX for zhemv Specifies the scalar <i>alpha</i>.
<i>a</i>	COMPLEX for chemv DOUBLE COMPLEX for zhemv Array, DIMENSION (<i>lda</i>, <i>n</i>). Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>n</i>-by-<i>n</i> upper triangular part of the array <i>a</i> must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of <i>a</i> is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', the leading <i>n</i>-by-<i>n</i> lower triangular part of the array <i>a</i> must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of <i>a</i> is not referenced. The imaginary parts of the diagonal elements need not be set and are assumed to be zero.
<i>lda</i>	INTEGER. Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, n)$.
<i>x</i>	COMPLEX for chemv DOUBLE COMPLEX for zhemv Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i>-element vector <i>x</i>.
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i>. The value of <i>incx</i> must not be zero.
<i>beta</i>	COMPLEX for chemv

	DOUBLE COMPLEX for <code>zhemv</code> Specifies the scalar <code>beta</code> . When <code>beta</code> is supplied as zero then <code>y</code> need not be set on input.
<code>y</code>	COMPLEX for <code>chemv</code> DOUBLE COMPLEX for <code>zhemv</code> Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array <code>y</code> must contain the n -element vector <code>y</code> .
<code>incy</code>	INTEGER. Specifies the increment for the elements of <code>y</code> . The value of <code>incy</code> must not be zero.

Output Parameters

<code>y</code>	Overwritten by the updated vector <code>y</code> .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hemv` interface are the following:

<code>a</code>	Holds the matrix <code>A</code> of size (n, n) .
<code>x</code>	Holds the vector of length (n) .
<code>y</code>	Holds the vector of length (n) .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.
<code>alpha</code>	The default value is 1.
<code>beta</code>	The default value is 1.

?her

Performs a rank-1 update of a Hermitian matrix.

Syntax

FORTRAN 77:

```
call cher(uplo, n, alpha, x, incx, a, lda )
call zher(uplo, n, alpha, x, incx, a, lda )
```

Fortran 95:

```
call her(a, x [,uplo] [, alpha])
```

Description

The ?her routines perform a matrix-vector operation defined as

$$A := \alpha x \text{conjg}(x') + A,$$

where:

alpha is a real scalar,

x is an *n*-element vector,

A is an *n*-by-*n* Hermitian matrix.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>a</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangular of the array <i>a</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular of the array <i>a</i> is used.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for cher DOUBLE PRECISION for zher Specifies the scalar <i>alpha</i> .
<i>x</i>	COMPLEX for cher DOUBLE COMPLEX for zher

	Array, dimension at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array x must contain the n -element vector x .
<i>incx</i>	INTEGER. Specifies the increment for the elements of x . The value of <i>incx</i> must not be zero.
<i>a</i>	COMPLEX for cher DOUBLE COMPLEX for zher Array, DIMENSION (<i>lda</i> , <i>n</i>). Before entry with <i>uplo</i> = 'U' or 'u', the leading n -by- n upper triangular part of the array a must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of a is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', the leading n -by- n lower triangular part of the array a must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of a is not referenced. The imaginary parts of the diagonal elements need not be set and are assumed to be zero.
<i>lda</i>	INTEGER. Specifies the first dimension of a as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, n)$.

Output Parameters

<i>a</i>	With <i>uplo</i> = 'U' or 'u', the upper triangular part of the array a is overwritten by the upper triangular part of the updated matrix. With <i>uplo</i> = 'L' or 'l', the lower triangular part of the array a is overwritten by the lower triangular part of the updated matrix. The imaginary parts of the diagonal elements are set to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `her` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (n,n) .
<i>x</i>	Holds the vector of length (n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>alpha</i>	The default value is 1.

?her2

Performs a rank-2 update of a Hermitian matrix.

Syntax

FORTRAN 77:

```
call cher2(uplo, n, alpha, x, incx, y, incy, a, lda )
call zher2(uplo, n, alpha, x, incx, y, incy, a, lda )
```

Fortran 95:

```
call her2(a, x, y [,uplo][,alpha])
```

Description

The ?her2 routines perform a matrix-vector operation defined as

$$A := \alpha * x * \text{conjg}(y') + \text{conjg}(\alpha) * y * \text{conjg}(x') + A,$$

where:

alpha is a scalar,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* Hermitian matrix.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>a</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangular of the array <i>a</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular of the array <i>a</i> is used.

<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	COMPLEX for <i>cher2</i> DOUBLE COMPLEX for <i>zher2</i> Specifies the scalar <i>alpha</i> .
<i>x</i>	COMPLEX for <i>cher2</i> DOUBLE COMPLEX for <i>zher2</i> Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>y</i>	COMPLEX for <i>cher2</i> DOUBLE COMPLEX for <i>zher2</i> Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array <i>y</i> must contain the <i>n</i> -element vector <i>y</i> .
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.
<i>a</i>	COMPLEX for <i>cher2</i> DOUBLE COMPLEX for <i>zher2</i> Array, DIMENSION (<i>lda</i> , <i>n</i>). Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>n</i> -by- <i>n</i> upper triangular part of the array <i>a</i> must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of <i>a</i> is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', the leading <i>n</i> -by- <i>n</i> lower triangular part of the array <i>a</i> must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of <i>a</i> is not referenced. The imaginary parts of the diagonal elements need not be set and are assumed to be zero.
<i>lda</i>	INTEGER. Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, n)$.

Output Parameters

a With *uplo* = 'U' or 'u', the upper triangular part of the array *a* is overwritten by the upper triangular part of the updated matrix.
 With *uplo* = 'L' or 'l', the lower triangular part of the array *a* is overwritten by the lower triangular part of the updated matrix.
 The imaginary parts of the diagonal elements are set to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `her2` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (n,n) .
<i>x</i>	Holds the vector of length (n) .
<i>y</i>	Holds the vector of length (n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>alpha</i>	The default value is 1.

?hpmv

Computes a matrix-vector product using a Hermitian packed matrix.

Syntax

FORTRAN 77:

```
call chpmv(uplo, n, alpha, ap, x, incx, beta, y, incy )
call zhpmv(uplo, n, alpha, ap, x, incx, beta, y, incy )
```

Fortran 95:

```
call hpmv(a, x, y [,uplo][,alpha] [,beta])
```

Description

The ?hpmv routines perform a matrix-vector operation defined as

$$y := \alpha A x + \beta y,$$

where:

α and β are scalars,

x and y are n -element vectors,

A is an n -by- n Hermitian matrix, supplied in packed form.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix A is supplied in the packed array ap. If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the matrix A is supplied in the packed array ap. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the matrix A is supplied in the packed array ap.
<i>n</i>	INTEGER. Specifies the order of the matrix A. The value of n must be at least zero.
<i>alpha</i>	COMPLEX for chpmv DOUBLE COMPLEX for zhpmv Specifies the scalar α.
<i>ap</i>	COMPLEX for chpmv DOUBLE COMPLEX for zhpmv Array, DIMENSION at least $((n*(n+1))/2)$. Before entry with <i>uplo</i> = 'U' or 'u', the array ap must contain the upper triangular part of the Hermitian matrix packed sequentially, column-by-column, so that $ap(1)$ contains $a(1, 1)$, $ap(2)$ and $ap(3)$ contain $a(1, 2)$ and $a(2, 2)$ respectively, and so on. Before entry with <i>uplo</i> = 'L' or 'l', the array ap must contain the lower triangular part of the Hermitian matrix packed sequentially, column-by-column, so that $ap(1)$ contains $a(1, 1)$, $ap(2)$ and $ap(3)$ contain $a(2, 1)$ and $a(3, 1)$ respectively, and so on.

	The imaginary parts of the diagonal elements need not be set and are assumed to be zero.
<i>x</i>	COMPLEX for <code>chpmv</code> DOUBLE PRECISION COMPLEX for <code>zhpmv</code> Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>beta</i>	COMPLEX for <code>chpmv</code> DOUBLE COMPLEX for <code>zhpmv</code> Specifies the scalar <i>beta</i> . When <i>beta</i> is equal to zero then <i>y</i> need not be set on input.
<i>y</i>	COMPLEX for <code>chpmv</code> DOUBLE COMPLEX for <code>zhpmv</code> Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array <i>y</i> must contain the <i>n</i> -element vector <i>y</i> .
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.

Output Parameters

<i>y</i>	Overwritten by the updated vector <i>y</i> .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hpmv` interface are the following:

<i>a</i>	Holds the array <i>a</i> of size $(n * (n + 1) / 2)$.
<i>x</i>	Holds the vector of length (<i>n</i>).
<i>y</i>	Holds the vector of length (<i>n</i>).
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>alpha</i>	The default value is 1.

beta

The default value is 1.

?hpr

Performs a rank-1 update of a Hermitian packed matrix.

Syntax

FORTRAN 77:

```
call chpr(uplo, n, alpha, x, incx, ap)
```

```
call zhpr(uplo, n, alpha, x, incx, ap)
```

Fortran 95:

```
call hpr(a, x [,uplo] [, alpha])
```

Description

The ?hpr routines perform a matrix-vector operation defined as

$$A := \alpha * x * \text{conjg}(x') + A,$$

where:

alpha is a real scalar,

x is an *n*-element vector,

A is an *n*-by-*n* Hermitian matrix, supplied in packed form.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i> . If <i>uplo</i> = 'U' or 'u', the upper triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i> . If <i>uplo</i> = 'L' or 'l', the low triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i> .
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for chpr

	DOUBLE PRECISION for zhpr Specifies the scalar <i>alpha</i> .
<i>x</i>	COMPLEX for chpr DOUBLE COMPLEX for zhpr Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . <i>incx</i> must not be zero.
<i>ap</i>	COMPLEX for chpr DOUBLE COMPLEX for zhpr Array, DIMENSION at least $((n * (n + 1)) / 2)$. Before entry with <i>uplo</i> = 'U' or 'u', the array <i>ap</i> must contain the upper triangular part of the Hermitian matrix packed sequentially, column-by-column, so that <i>ap</i> (1) contains <i>a</i> (1, 1), <i>ap</i> (2) and <i>ap</i> (3) contain <i>a</i> (1, 2) and <i>a</i> (2, 2) respectively, and so on. Before entry with <i>uplo</i> = 'L' or 'l', the array <i>ap</i> must contain the lower triangular part of the Hermitian matrix packed sequentially, column-by-column, so that <i>ap</i> (1) contains <i>a</i> (1, 1), <i>ap</i> (2) and <i>ap</i> (3) contain <i>a</i> (2, 1) and <i>a</i> (3, 1) respectively, and so on. The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

Output Parameters

<i>ap</i>	With <i>uplo</i> = 'U' or 'u', overwritten by the upper triangular part of the updated matrix. With <i>uplo</i> = 'L' or 'l', overwritten by the lower triangular part of the updated matrix. The imaginary parts of the diagonal elements are set to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hpr` interface are the following:

<code>a</code>	Holds the array <code>a</code> of size $(n*(n+1)/2)$.
<code>x</code>	Holds the vector of length (n) .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.
<code>alpha</code>	The default value is 1.

?hpr2

Performs a rank-2 update of a Hermitian packed matrix.

Syntax

FORTRAN 77:

```
call chpr2(uplo, n, alpha, x, incx, y, incy, ap )
call zhpr2(uplo, n, alpha, x, incx, y, incy, ap )
```

Fortran 95:

```
call hpr2(a, x, y [,uplo][,alpha])
```

Description

The `?hpr2` routines perform a matrix-vector operation defined as

$$A := \alpha * x * \text{conjg}(y') + \text{conjg}(\alpha) * y * \text{conjg}(x') + A,$$

where:

`alpha` is a scalar,

`x` and `y` are n -element vectors,

`A` is an n -by- n Hermitian matrix, supplied in packed form.

Input Parameters

`uplo` CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix `A` is supplied in the packed array `ap`.
If `uplo = 'U'` or `'u'`, then the upper triangular part of the matrix `A` is supplied in the packed array `ap`.

	If <i>uplo</i> = 'L' or 'l', then the low triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i>.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i>. The value of <i>n</i> must be at least zero.
<i>alpha</i>	COMPLEX for <i>chpr2</i> DOUBLE COMPLEX for <i>zhpr2</i> Specifies the scalar <i>alpha</i>.
<i>x</i>	COMPLEX for <i>chpr2</i> DOUBLE COMPLEX for <i>zhpr2</i> Array, dimension at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i>-element vector <i>x</i>.
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i>. The value of <i>incx</i> must not be zero.
<i>y</i>	COMPLEX for <i>chpr2</i> DOUBLE COMPLEX for <i>zhpr2</i> Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array <i>y</i> must contain the <i>n</i>-element vector <i>y</i>.
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i>. The value of <i>incy</i> must not be zero.
<i>ap</i>	COMPLEX for <i>chpr2</i> DOUBLE COMPLEX for <i>zhpr2</i> Array, DIMENSION at least $((n * (n + 1)) / 2)$. Before entry with <i>uplo</i> = 'U' or 'u', the array <i>ap</i> must contain the upper triangular part of the Hermitian matrix packed sequentially, column-by-column, so that <i>ap</i>(1) contains <i>a</i>(1,1), <i>ap</i>(2) and <i>ap</i>(3) contain <i>a</i>(1,2) and <i>a</i>(2,2) respectively, and so on. Before entry with <i>uplo</i> = 'L' or 'l', the array <i>ap</i> must contain the lower triangular part of the Hermitian matrix packed sequentially, column-by-column, so that <i>ap</i>(1) contains <i>a</i>(1,1), <i>ap</i>(2) and <i>ap</i>(3) contain <i>a</i>(2,1) and <i>a</i>(3,1) respectively, and so on. The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

Output Parameters

ap With *uplo* = 'U' or 'u', overwritten by the upper triangular part of the updated matrix.
 With *uplo* = 'L' or 'l', overwritten by the lower triangular part of the updated matrix.
 The imaginary parts of the diagonal elements need are set to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hpr2` interface are the following:

<i>a</i>	Holds the array <i>a</i> of size $(n * (n+1) / 2)$.
<i>x</i>	Holds the vector of length (n) .
<i>y</i>	Holds the vector of length (n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>alpha</i>	The default value is 1.

?sbmv

Computes a matrix-vector product using a symmetric band matrix.

Syntax

FORTRAN 77:

```
call ssbmv(uplo, n, k, alpha, a, lda, x, incx, beta, y, incy)
call dsbmv(uplo, n, k, alpha, a, lda, x, incx, beta, y, incy)
```

Fortran 95:

```
call sbmv(a, x, y [,uplo][,alpha] [,beta])
```

Description

The ?sbmv routines perform a matrix-vector operation defined as

$$y := \alpha * A * x + \beta * y,$$

where:

α and β are scalars,

x and y are n -element vectors,

A is an n -by- n symmetric band matrix, with k super-diagonals.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the band matrix A is used: if <i>uplo</i> = 'U' or 'u' - upper triangular part; if <i>uplo</i> = 'L' or 'l' - low triangular part.
<i>n</i>	INTEGER. Specifies the order of the matrix A . The value of n must be at least zero.
<i>k</i>	INTEGER. Specifies the number of super-diagonals of the matrix A . The value of k must satisfy $0 \leq k$.
<i>alpha</i>	REAL for ssbmv DOUBLE PRECISION for dsbmv Specifies the scalar α .
<i>a</i>	REAL for ssbmv DOUBLE PRECISION for dsbmv Array, DIMENSION (<i>lda</i> , n). Before entry with <i>uplo</i> = 'U' or 'u', the leading $(k + 1)$ by n part of the array a must contain the upper triangular band part of the symmetric matrix, supplied column-by-column, with the leading diagonal of the matrix in row $(k + 1)$ of the array, the first super-diagonal starting at position 2 in row k , and so on. The top left k by k triangle of the array a is not referenced.

The following program segment transfers the upper triangular part of a symmetric band matrix from conventional full matrix storage to band storage:

```

do 20, j = 1, n
    m = k + 1 - j
    do 10, i = max( 1, j - k ), j
        a( m + i, j ) = matrix( i, j )
    10 continue
20 continue

```

Before entry with `uplo = 'L' or 'l'`, the leading $(k + 1)$ by n part of the array `a` must contain the lower triangular band part of the symmetric matrix, supplied column-by-column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right k by k triangle of the array `a` is not referenced.

The following program segment transfers the lower triangular part of a symmetric band matrix from conventional full matrix storage to band storage:

```

do 20, j = 1, n
    m = 1 - j
    do 10, i = j, min( n, j + k )
        a( m + i, j ) = matrix( i, j )
    10 continue
20 continue

```

`lda`

INTEGER. Specifies the first dimension of `a` as declared in the calling (sub)program. The value of `lda` must be at least $(k + 1)$.

`x`

REAL for `ssbmv`

DOUBLE PRECISION for `dsbmv`

Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incx}))$.

Before entry, the incremented array `x` must contain the vector `x`.

<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>beta</i>	REAL for <i>ssbm</i> v DOUBLE PRECISION for <i>dsbm</i> v Specifies the scalar <i>beta</i> .
<i>y</i>	REAL for <i>ssbm</i> v DOUBLE PRECISION for <i>dsbm</i> v Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array <i>y</i> must contain the vector <i>y</i> .
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.

Output Parameters

<i>y</i>	Overwritten by the updated vector <i>y</i> .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *sbmv* interface are the following:

<i>a</i>	Holds the array <i>a</i> of size $(k+1, n)$.
<i>x</i>	Holds the vector of length (n) .
<i>y</i>	Holds the vector of length (n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 1.

?spmv

Computes a matrix-vector product using a symmetric packed matrix.

Syntax

FORTRAN 77:

```
call sspmv(uplo, n, alpha, ap, x, incx, beta, y, incy )
call dspmv(uplo, n, alpha, ap, x, incx, beta, y, incy )
```

Fortran 95:

```
call spmv(a, x, y [,uplo][,alpha] [,beta])
```

Description

The ?spmv routines perform a matrix-vector operation defined as

$$y := \alpha A x + \beta y,$$

where:

α and β are scalars,

x and y are n -element vectors,

A is an n -by- n symmetric matrix, supplied in packed form.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix A is supplied in the packed array ap. If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the matrix A is supplied in the packed array ap. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the matrix A is supplied in the packed array ap.
<i>n</i>	INTEGER. Specifies the order of the matrix a. The value of n must be at least zero.
<i>alpha</i>	REAL for sspmv DOUBLE PRECISION for dspmv Specifies the scalar α.

<i>ap</i>	REAL for <code>sspmv</code> DOUBLE PRECISION for <code>dspmv</code> Array, DIMENSION at least $(n*(n + 1))/2$. Before entry with <i>uplo</i> = 'U' or 'u', the array <i>ap</i> must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that <i>ap</i>(1) contains <i>a</i>(1,1), <i>ap</i>(2) and <i>ap</i>(3) contain <i>a</i>(1,2) and <i>a</i>(2, 2) respectively, and so on. Before entry with <i>uplo</i> = 'L' or 'l', the array <i>ap</i> must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that <i>ap</i>(1) contains <i>a</i>(1,1), <i>ap</i>(2) and <i>ap</i>(3) contain <i>a</i>(2,1) and <i>a</i>(3,1) respectively, and so on.
<i>x</i>	REAL for <code>sspmv</code> DOUBLE PRECISION for <code>dspmv</code> Array, DIMENSION at least $(1 + (n - 1)*abs(incx))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i>-element vector <i>x</i>.
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i>. The value of <i>incx</i> must not be zero.
<i>beta</i>	REAL for <code>sspmv</code> DOUBLE PRECISION for <code>dspmv</code> Specifies the scalar <i>beta</i>. When <i>beta</i> is supplied as zero, then <i>y</i> need not be set on input.
<i>y</i>	REAL for <code>sspmv</code> DOUBLE PRECISION for <code>dspmv</code> Array, DIMENSION at least $(1 + (n - 1)*abs(incy))$. Before entry, the incremented array <i>y</i> must contain the <i>n</i>-element vector <i>y</i>.
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i>. The value of <i>incy</i> must not be zero.

Output Parameters

<i>y</i>	Overwritten by the updated vector <i>y</i> .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `spmv` interface are the following:

<i>a</i>	Holds the array <i>a</i> of size $(n*(n+1)/2)$.
<i>x</i>	Holds the vector of length (<i>n</i>).
<i>y</i>	Holds the vector of length (<i>n</i>).
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 1.

?spr

Performs a rank-1 update of a symmetric packed matrix.

Syntax

FORTRAN 77:

```
call sspr(uplo, n, alpha, x, incx, ap )
call dspr(uplo, n, alpha, x, incx, ap )
```

Fortran 95:

```
call spr(a, x [,uplo] [, alpha])
```

Description

The `?spr` routines perform a matrix-vector operation defined as

$$a := \alpha * x * x' + A,$$

where:

alpha is a real scalar,

x is an *n*-element vector,

A is an *n*-by-*n* symmetric matrix, supplied in packed form.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i>. If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i>. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i>.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i>. The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for sspr DOUBLE PRECISION for dspr Specifies the scalar <i>alpha</i>.
<i>x</i>	REAL for sspr DOUBLE PRECISION for dspr Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i>-element vector <i>x</i>.
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i>. The value of <i>incx</i> must not be zero.
<i>ap</i>	REAL for sspr DOUBLE PRECISION for dspr Array, DIMENSION at least $((n * (n + 1)) / 2)$. Before entry with <i>uplo</i> = 'U' or 'u', the array <i>ap</i> must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that <i>ap</i>(1) contains <i>a</i>(1,1), <i>ap</i>(2) and <i>ap</i>(3) contain <i>a</i>(1,2) and <i>a</i>(2,2) respectively, and so on. Before entry with <i>uplo</i> = 'L' or 'l', the array <i>ap</i> must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that <i>ap</i>(1) contains <i>a</i>(1,1), <i>ap</i>(2) and <i>ap</i>(3) contain <i>a</i>(2,1) and <i>a</i>(3,1) respectively, and so on.

Output Parameters

ap With *uplo* = 'U' or 'u', overwritten by the upper triangular part of the updated matrix.
 With *uplo* = 'L' or 'l', overwritten by the lower triangular part of the updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *spr* interface are the following:

a Holds the array *a* of size $(n*(n+1)/2)$.
x Holds the vector of length (n) .
uplo Must be 'U' or 'L'. The default value is 'U'.
alpha The default value is 1.

?spr2

Performs a rank-2 update of a symmetric packed matrix.

Syntax

FORTRAN 77:

```
call sspr2(uplo, n, alpha, x, incx, y, incy, ap )
call dspr2(uplo, n, alpha, x, incx, y, incy, ap )
```

Fortran 95:

```
call spr2(a, x, y [,uplo][,alpha])
```

Description

The ?spr2 routines perform a matrix-vector operation defined as

$$A := \alpha x y' + \alpha y x' + A,$$

where:

alpha is a scalar,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* symmetric matrix, supplied in packed form.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i>. If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i>. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i>.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i>. The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for sspr2 DOUBLE PRECISION for dspr2 Specifies the scalar <i>alpha</i>.
<i>x</i>	REAL for sspr2 DOUBLE PRECISION for dspr2 Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i>-element vector <i>x</i>.
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i>. The value of <i>incx</i> must not be zero.
<i>y</i>	REAL for sspr2 DOUBLE PRECISION for dspr2 Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array <i>y</i> must contain the <i>n</i>-element vector <i>y</i>.
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i>. The value of <i>incy</i> must not be zero.
<i>ap</i>	REAL for sspr2 DOUBLE PRECISION for dspr2

Array, `DIMENSION` at least $((n * (n + 1)) / 2)$. Before entry with `uplo = 'U' or 'u'`, the array `ap` must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that `ap(1)` contains $a(1,1)$, `ap(2)` and `ap(3)` contain $a(1,2)$ and $a(2,2)$ respectively, and so on.

Before entry with `uplo = 'L' or 'l'`, the array `ap` must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that `ap(1)` contains $a(1,1)$, `ap(2)` and `ap(3)` contain $a(2,1)$ and $a(3,1)$ respectively, and so on.

Output Parameters

`ap`

With `uplo = 'U' or 'u'`, overwritten by the upper triangular part of the updated matrix.

With `uplo = 'L' or 'l'`, overwritten by the lower triangular part of the updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `spr2` interface are the following:

<code>a</code>	Holds the array <code>a</code> of size $(n * (n + 1)) / 2$.
<code>x</code>	Holds the vector of length (n) .
<code>y</code>	Holds the vector of length (n) .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.
<code>alpha</code>	The default value is 1.

?symv

Computes a matrix-vector product for a symmetric matrix.

Syntax

FORTRAN 77:

```
call ssymv(uplo, n, alpha, a, lda, x, incx, beta, y, incy )
call dsymv(uplo, n, alpha, a, lda, x, incx, beta, y, incy )
```

Fortran 95:

```
call symv(a, x, y [,uplo][,alpha] [,beta])
```

Description

The ?symv routines perform a matrix-vector operation defined as

$$y := \alpha A x + \beta y,$$

where:

α and β are scalars,

x and y are n -element vectors,

A is an n -by- n symmetric matrix.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>a</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the array <i>a</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the array <i>a</i> is used.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>a</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for ssymv DOUBLE PRECISION for dsymv Specifies the scalar <i>alpha</i> .
<i>a</i>	REAL for ssymv

	DOUBLE PRECISION for dsymv Array, DIMENSION (<i>lda</i>, <i>n</i>). Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>n</i>-by-<i>n</i> upper triangular part of the array <i>a</i> must contain the upper triangular part of the symmetric matrix <i>A</i> and the strictly lower triangular part of <i>a</i> is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', the leading <i>n</i>-by-<i>n</i> lower triangular part of the array <i>a</i> must contain the lower triangular part of the symmetric matrix <i>A</i> and the strictly upper triangular part of <i>a</i> is not referenced.
<i>lda</i>	INTEGER. Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, n)$.
<i>x</i>	REAL for ssymv DOUBLE PRECISION for dsymv Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i>-element vector <i>x</i>.
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i>. The value of <i>incx</i> must not be zero.
<i>beta</i>	REAL for ssymv DOUBLE PRECISION for dsymv Specifies the scalar <i>beta</i>. When <i>beta</i> is supplied as zero, then <i>y</i> need not be set on input.
<i>y</i>	REAL for ssymv DOUBLE PRECISION for dsymv Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array <i>y</i> must contain the <i>n</i>-element vector <i>y</i>.
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i>. The value of <i>incy</i> must not be zero.

Output Parameters

<i>y</i>	Overwritten by the updated vector <i>y</i> .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `symv` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (n,n) .
<i>x</i>	Holds the vector of length (n) .
<i>y</i>	Holds the vector of length (n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 1.

?syr

Performs a rank-1 update of a symmetric matrix.

Syntax

FORTRAN 77:

```
call ssyr(uplo, n, alpha, x, incx, a, lda )
call dsyr(uplo, n, alpha, x, incx, a, lda )
```

Fortran 95:

```
call syr(a, x [,uplo] [, alpha])
```

Description

The `?syr` routines perform a matrix-vector operation defined as

$$A := \alpha * x * x' + A,$$

where:

alpha is a real scalar,

x is an *n*-element vector,

A is an *n*-by-*n* symmetric matrix.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>a</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the array <i>a</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the array <i>a</i> is used.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i>. The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for ssyr DOUBLE PRECISION for dsyr Specifies the scalar <i>alpha</i>.
<i>x</i>	REAL for ssyr DOUBLE PRECISION for dsyr Array, DIMENSION at least $(1 + (n-1)*abs(incx))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i>-element vector <i>x</i>.
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i>. The value of <i>incx</i> must not be zero.
<i>a</i>	REAL for ssyr DOUBLE PRECISION for dsyr Array, DIMENSION (<i>lda</i>, <i>n</i>). Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>n</i>-by-<i>n</i> upper triangular part of the array <i>a</i> must contain the upper triangular part of the symmetric matrix <i>A</i> and the strictly lower triangular part of <i>a</i> is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', the leading <i>n</i>-by-<i>n</i> lower triangular part of the array <i>a</i> must contain the lower triangular part of the symmetric matrix <i>A</i> and the strictly upper triangular part of <i>a</i> is not referenced.
<i>lda</i>	INTEGER. Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, n)$.

Output Parameters

a With *uplo* = 'U' or 'u', the upper triangular part of the array *a* is overwritten by the upper triangular part of the updated matrix.
 With *uplo* = 'L' or 'l', the lower triangular part of the array *a* is overwritten by the lower triangular part of the updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `syr` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (n,n) .
<i>x</i>	Holds the vector of length (n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>alpha</i>	The default value is 1.

?syr2

Performs a rank-2 update of symmetric matrix.

Syntax

FORTRAN 77:

```
call ssyr2(uplo, n, alpha, x, incx, y, incy, a, lda )
call dsyr2(uplo, n, alpha, x, incx, y, incy, a, lda )
```

Fortran 95:

```
call syr2(a, x, y [,uplo][,alpha])
```

Description

The `?syr2` routines perform a matrix-vector operation defined as

$$A := \alpha x x^T + \alpha y y^T + A,$$

where:

α is a scalar,

x and y are n -element vectors,

A is an n -by- n symmetric matrix.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array a is used. If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the array a is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the array a is used.
<i>n</i>	INTEGER. Specifies the order of the matrix A. The value of n must be at least zero.
<i>alpha</i>	REAL for ssyr2 DOUBLE PRECISION for dsyr2 Specifies the scalar α.
<i>x</i>	REAL for ssyr2 DOUBLE PRECISION for dsyr2 Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array x must contain the n-element vector x.
<i>incx</i>	INTEGER. Specifies the increment for the elements of x. The value of <i>incx</i> must not be zero.
<i>y</i>	REAL for ssyr2 DOUBLE PRECISION for dsyr2 Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array y must contain the n-element vector y.
<i>incy</i>	INTEGER. Specifies the increment for the elements of y. The value of <i>incy</i> must not be zero.
<i>a</i>	REAL for ssyr2 DOUBLE PRECISION for dsyr2 Array, DIMENSION (lda, n).

Before entry with `uplo = 'U' or 'u'`, the leading n -by- n upper triangular part of the array `a` must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of `a` is not referenced.

Before entry with `uplo = 'L' or 'l'`, the leading n -by- n lower triangular part of the array `a` must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of `a` is not referenced.

`lda`

INTEGER. Specifies the first dimension of `a` as declared in the calling (sub)program. The value of `lda` must be at least $\max(1, n)$.

Output Parameters

`a`

With `uplo = 'U' or 'u'`, the upper triangular part of the array `a` is overwritten by the upper triangular part of the updated matrix.

With `uplo = 'L' or 'l'`, the lower triangular part of the array `a` is overwritten by the lower triangular part of the updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sy2` interface are the following:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>x</code>	Holds the vector x of length (n) .
<code>y</code>	Holds the vector y of length (n) .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.
<code>alpha</code>	The default value is 1.

?tbmv

Computes a matrix-vector product using a triangular band matrix.

Syntax

FORTRAN 77:

```
call stbmv(uplo, trans, diag, n, k, a, lda, x, incx )
call dtbmv(uplo, trans, diag, n, k, a, lda, x, incx )
call ctbmv(uplo, trans, diag, n, k, a, lda, x, incx )
call ztbmv(uplo, trans, diag, n, k, a, lda, x, incx )
```

Fortran 95:

```
call tbmv(a, x [,uplo] [, trans] [,diag])
```

Description

The ?tbmv routines perform one of the matrix-vector operations defined as

$x := A*x$, or $x := A'*x$, or $x := \text{conjg}(A')*x$,

where:

x is an n -element vector,

A is an n -by- n unit, or non-unit, upper or lower triangular band matrix, with $(k + 1)$ diagonals.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the matrix A is an upper or lower triangular matrix: if <i>uplo</i> = 'U' or 'u', then the matrix is upper triangular; if <i>uplo</i> = 'L' or 'l', then the matrix is low triangular.
<i>trans</i>	CHARACTER*1. Specifies the operation: if <i>trans</i> = 'N' or 'n', then $x := A*x$; if <i>trans</i> = 'T' or 't', then $x := A'*x$; if <i>trans</i> = 'C' or 'c', then $x := \text{conjg}(A')*x$.
<i>diag</i>	CHARACTER*1. Specifies whether the matrix A is unit triangular:

if *diag* = 'U' or 'u' then the matrix is unit triangular;
 if *diag* = 'N' or 'n', then the matrix is not unit triangular.

n INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

k INTEGER. On entry with *uplo* = 'U' or 'u', *k* specifies the number of super-diagonals of the matrix *A*. On entry with *uplo* = 'L' or 'l', *k* specifies the number of sub-diagonals of the matrix *a*.
 The value of *k* must satisfy $0 \leq k$.

a REAL for *stbmv*
 DOUBLE PRECISION for *dtbmv*
 COMPLEX for *ctbmv*
 DOUBLE COMPLEX for *ztbmv*
 Array, DIMENSION (*lda*, *n*).
 Before entry with *uplo* = 'U' or 'u', the leading (*k* + 1) by *n* part of the array *a* must contain the upper triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row (*k* + 1) of the array, the first super-diagonal starting at position 2 in row *k*, and so on. The top left *k* by *k* triangle of the array *a* is not referenced. The following program segment transfers an upper triangular band matrix from conventional full matrix storage to band storage:

```

do 20, j = 1, n
    m = k + 1 - j
do 10, i = max(1, j - k), j
    a(m + i, j) = matrix(i, j)
10 continue
20 continue
  
```

Before entry with *uplo* = 'L' or 'l', the leading (*k* + 1) by *n* part of the array *a* must contain the lower triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right *k* by *k*

triangle of the array *a* is not referenced. The following program segment transfers a lower triangular band matrix from conventional full matrix storage to band storage:

```

do 20, j = 1, n
  m = 1 - j
  do 10, i = j, min(n, j + k)
    a(m + i, j) = matrix (i, j)
  10 continue
20 continue

```

Note that when *diag* = 'U' or 'u', the elements of the array *a* corresponding to the diagonal elements of the matrix are not referenced, but are assumed to be unity.

lda INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least $(k + 1)$.

x REAL for stbmv
 DOUBLE PRECISION for dtbmv
 COMPLEX for ctbmv
 DOUBLE COMPLEX for ztbmv
 Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incx}))$.
 Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx INTEGER. Specifies the increment for the elements of *x*.
 The value of *incx* must not be zero.

Output Parameters

x Overwritten with the transformed vector *x*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tbmv` interface are the following:

<i>a</i>	Holds the array <i>a</i> of size $(k+1, n)$.
<i>x</i>	Holds the vector of length (n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.

?tbsv

Solves a system of linear equations whose coefficients are in a triangular band matrix.

Syntax

FORTRAN 77:

```
call stbsv(uplo, trans, diag, n, k, a, lda, x, incx )
call dtbsv(uplo, trans, diag, n, k, a, lda, x, incx )
call ctbsv(uplo, trans, diag, n, k, a, lda, x, incx )
call ztbsv(uplo, trans, diag, n, k, a, lda, x, incx )
```

Fortran 95:

```
call tbsv(a, x [,uplo] [, trans] [,diag])
```

Description

The ?tbsv routines solve one of the following systems of equations:

$A*x = b$, or $A'*x = b$, or $\text{conjg}(A')*x = b$,

where:

b and *x* are *n*-element vectors,

A is an *n*-by-*n* unit, or non-unit, upper or lower triangular band matrix, with $(k + 1)$ diagonals.

The routine does not test for singularity or near-singularity.

Such tests must be performed before calling this routine.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the matrix <i>A</i> is an upper or lower triangular matrix: if <i>uplo</i> = 'U' or 'u' the matrix is upper triangular; if <i>uplo</i> = 'L' or 'l', the matrix is low triangular.
<i>trans</i>	CHARACTER*1. Specifies the system of equations: if <i>trans</i> = 'N' or 'n', then $A*x = b$; if <i>trans</i> = 'T' or 't', then $A'*x = b$; if <i>trans</i> = 'C' or 'c', then $\text{conjg}(A)*x = b$.
<i>diag</i>	CHARACTER*1. Specifies whether the matrix <i>A</i> is unit triangular: if <i>diag</i> = 'U' or 'u' then the matrix is unit triangular; if <i>diag</i> = 'N' or 'n', then the matrix is not unit triangular.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i>. The value of <i>n</i> must be at least zero.
<i>k</i>	INTEGER. On entry with <i>uplo</i> = 'U' or 'u', <i>k</i> specifies the number of super-diagonals of the matrix <i>A</i>. On entry with <i>uplo</i> = 'L' or 'l', <i>k</i> specifies the number of sub-diagonals of the matrix <i>A</i>. The value of <i>k</i> must satisfy $0 \leq k$.
<i>a</i>	REAL for stbsv DOUBLE PRECISION for dtbsv COMPLEX for ctbsv DOUBLE COMPLEX for ztbsv Array, DIMENSION (<i>lda</i>, <i>n</i>). Before entry with <i>uplo</i> = 'U' or 'u', the leading $(k + 1)$ by <i>n</i> part of the array <i>a</i> must contain the upper triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row $(k + 1)$ of the array, the first super-diagonal starting at position 2 in row <i>k</i>, and so on. The top left <i>k</i> by <i>k</i> triangle of the array <i>a</i> is not referenced.

The following program segment transfers an upper triangular band matrix from conventional full matrix storage to band storage:

```

do 20, j = 1, n
    m = k + 1 - j
    do 10, i = max(1, j - k), j1
        a(m + i, j) = matrix (i, j)
    10 continue
20 continue

```

Before entry with *uplo* = 'L' or 'l', the leading $(k + 1)$ by n part of the array *a* must contain the lower triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right k by k triangle of the array *a* is not referenced.

The following program segment transfers a lower triangular band matrix from conventional full matrix storage to band storage:

```

do 20, j = 1, n
    m = 1 - j
    do 10, i = j, min(n, j + k)
        a(m + i, j) = matrix (i, j)
    10 continue
20 continue

```

When *diag* = 'U' or 'u', the elements of the array *a* corresponding to the diagonal elements of the matrix are not referenced, but are assumed to be unity.

lda

INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least $(k + 1)$.

x

REAL for stbsv

DOUBLE PRECISION for dtbsv

COMPLEX for `ctbsv`
 DOUBLE COMPLEX for `ztbsv`
 Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incx}))$.
 Before entry, the incremented array x must contain the
 n -element right-hand side vector b .
incx INTEGER. Specifies the increment for the elements of x .
 The value of *incx* must not be zero.

Output Parameters

x Overwritten with the solution vector x .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tbsv` interface are the following:

<i>a</i>	Holds the array a of size $(k+1, n)$.
x	Holds the vector of length (n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.

?tpmv

Computes a matrix-vector product using a triangular packed matrix.

Syntax

FORTRAN 77:

```
call stpmv(uplo, trans, diag, n, ap, x, incx )
call dtpmv(uplo, trans, diag, n, ap, x, incx )
call ctpmv(uplo, trans, diag, n, ap, x, incx )
call ztpmv(uplo, trans, diag, n, ap, x, incx )
```

Fortran 95:

```
call tpmv(a, x [,uplo] [, trans] [,diag])
```

Description

The ?tpmv routines perform one of the matrix-vector operations defined as

$x := A * x$, or $x := A' * x$, or $x := \text{conjg}(A') * x$,

where:

x is an n -element vector,

A is an n -by- n unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the matrix A is upper or lower triangular: if <i>uplo</i> = 'U' or 'u', then the matrix is upper triangular; if <i>uplo</i> = 'L' or 'l', then the matrix is low triangular.
<i>trans</i>	CHARACTER*1. Specifies the operation: if <i>trans</i> = 'N' or 'n', then $x := A * x$; if <i>trans</i> = 'T' or 't', then $x := A' * x$; if <i>trans</i> = 'C' or 'c', then $x := \text{conjg}(A') * x$.
<i>diag</i>	CHARACTER*1. Specifies whether the matrix A is unit triangular:

	if <i>diag</i> = 'U' or 'u' then the matrix is unit triangular; if <i>diag</i> = 'N' or 'n', then the matrix is not unit triangular.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i>. The value of <i>n</i> must be at least zero.
<i>ap</i>	REAL for <i>stpmv</i> DOUBLE PRECISION for <i>dtpmv</i> COMPLEX for <i>ctpmv</i> DOUBLE COMPLEX for <i>ztpmv</i> Array, DIMENSION at least $((n*(n+1))/2)$. Before entry with <i>uplo</i> = 'U' or 'u', the array <i>ap</i> must contain the upper triangular matrix packed sequentially, column-by-column, so that <i>ap</i>(1) contains <i>a</i>(1,1), <i>ap</i>(2) and <i>ap</i>(3) contain <i>a</i>(1,2) and <i>a</i>(2,2) respectively, and so on. Before entry with <i>uplo</i> = 'L' or 'l', the array <i>ap</i> must contain the lower triangular matrix packed sequentially, column-by-column, so that <i>ap</i>(1) contains <i>a</i>(1,1), <i>ap</i>(2) and <i>ap</i>(3) contain <i>a</i>(2,1) and <i>a</i>(3,1) respectively, and so on. When <i>diag</i> = 'U' or 'u', the diagonal elements of <i>a</i> are not referenced, but are assumed to be unity.
<i>x</i>	REAL for <i>stpmv</i> DOUBLE PRECISION for <i>dtpmv</i> COMPLEX for <i>ctpmv</i> DOUBLE COMPLEX for <i>ztpmv</i> Array, DIMENSION at least $(1 + (n-1)*abs(incx))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i>-element vector <i>x</i>.
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i>. The value of <i>incx</i> must not be zero.

Output Parameters

<i>x</i>	Overwritten with the transformed vector <i>x</i> .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tpmv` interface are the following:

<code>a</code>	Holds the array <code>a</code> of size $(n * (n+1) / 2)$.
<code>x</code>	Holds the vector of length (n) .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.
<code>trans</code>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<code>diag</code>	Must be 'N' or 'U'. The default value is 'N'.

?tpsv

Solves a system of linear equations whose coefficients are in a triangular packed matrix.

Syntax

FORTRAN 77:

```
call stpsv(uplo, trans, diag, n, ap, x, incx )
call dtpsv(uplo, trans, diag, n, ap, x, incx )
call ctpsv(uplo, trans, diag, n, ap, x, incx )
call ztpsv(uplo, trans, diag, n, ap, x, incx )
```

Fortran 95:

```
call tpsv(a, x [,uplo] [, trans] [,diag])
```

Description

The `?tpsv` routines solve one of the following systems of equations

$A * x = b$, or $A' * x = b$, or $\text{conjg}(A') * x = b$,

where:

b and x are n -element vectors,

A is an n -by- n unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

This routine does not test for singularity or near-singularity.

Such tests must be performed before calling this routine.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the matrix <i>A</i> is upper or lower triangular: if <i>uplo</i> = 'U' or 'u', then the matrix is upper triangular; if <i>uplo</i> = 'L' or 'l', then the matrix is low triangular.
<i>trans</i>	CHARACTER*1. Specifies the system of equations: if <i>trans</i> = 'N' or 'n', then $A*x = b$; if <i>trans</i> = 'T' or 't', then $A'*x = b$; if <i>trans</i> = 'C' or 'c', then $\text{conjg}(A')*x = b$.
<i>diag</i>	CHARACTER*1. Specifies whether the matrix <i>A</i> is unit triangular: if <i>diag</i> = 'U' or 'u' then the matrix is unit triangular; if <i>diag</i> = 'N' or 'n', then the matrix is not unit triangular.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i>. The value of <i>n</i> must be at least zero.
<i>ap</i>	REAL for stpsv DOUBLE PRECISION for dtpsv COMPLEX for ctpsv DOUBLE COMPLEX for ztpsv Array, DIMENSION at least $((n*(n+1))/2)$. Before entry with <i>uplo</i> = 'U' or 'u', the array <i>ap</i> must contain the upper triangular matrix packed sequentially, column-by-column, so that <i>ap</i>(1) contains $a(1, +1)$, <i>ap</i>(2) and <i>ap</i>(3) contain $a(1, 2)$ and $a(2, 2)$ respectively, and so on. Before entry with <i>uplo</i> = 'L' or 'l', the array <i>ap</i> must contain the lower triangular matrix packed sequentially, column-by-column, so that <i>ap</i>(1) contains $a(1, +1)$, <i>ap</i>(2) and <i>ap</i>(3) contain $a(2, +1)$ and $a(3, +1)$ respectively, and so on. When <i>diag</i> = 'U' or 'u', the diagonal elements of <i>a</i> are not referenced, but are assumed to be unity.
<i>x</i>	REAL for stpsv DOUBLE PRECISION for dtpsv COMPLEX for ctpsv DOUBLE COMPLEX for ztpsv

Array, `DIMENSION` at least $(1 + (n - 1) * \text{abs}(incx))$.
 Before entry, the incremented array `x` must contain the
 n -element right-hand side vector `b`.

`incx` INTEGER. Specifies the increment for the elements of `x`.
 The value of `incx` must not be zero.

Output Parameters

`x` Overwritten with the solution vector `x`.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tpsv` interface are the following:

<code>a</code>	Holds the array <code>a</code> of size $(n * (n + 1) / 2)$.
<code>x</code>	Holds the vector of length (n) .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.
<code>trans</code>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<code>diag</code>	Must be 'N' or 'U'. The default value is 'N'.

?trmv

Computes a matrix-vector product using a triangular matrix.

Syntax

FORTRAN 77:

```
call strmv(uplo, trans, diag, n, a, lda, x, incx )
call dtrmv(uplo, trans, diag, n, a, lda, x, incx )
call ctrmv(uplo, trans, diag, n, a, lda, x, incx )
call ztrmv(uplo, trans, diag, n, a, lda, x, incx )
```

Fortran 95:

```
call trmv(a, x [,uplo] [, trans] [,diag])
```

Description

The ?trmv routines perform one of the following matrix-vector operations defined as

$x := A*x$, or $x := A'*x$, or $x := \text{conjg}(A')*x$,

where:

x is an n -element vector,

A is an n -by- n unit, or non-unit, upper or lower triangular matrix.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the matrix A is upper or lower triangular: if <i>uplo</i> = 'U' or 'u', then the matrix is upper triangular; if <i>uplo</i> = 'L' or 'l', then the matrix is low triangular.
<i>trans</i>	CHARACTER*1. Specifies the operation: if <i>trans</i> = 'N' or 'n', then $x := A*x$; if <i>trans</i> = 'T' or 't', then $x := A'*x$; if <i>trans</i> = 'C' or 'c', then $x := \text{conjg}(A')*x$.
<i>diag</i>	CHARACTER*1. Specifies whether the matrix A is unit triangular: if <i>diag</i> = 'U' or 'u' then the matrix is unit triangular; if <i>diag</i> = 'N' or 'n', then the matrix is not unit triangular.
<i>n</i>	INTEGER. Specifies the order of the matrix A . The value of n must be at least zero.
<i>a</i>	REAL for strmv DOUBLE PRECISION for dtrmv COMPLEX for ctrmv DOUBLE COMPLEX for ztrmv Array, DIMENSION (<i>lda</i> , n). Before entry with <i>uplo</i> = 'U' or 'u', the leading n -by- n upper triangular part of the array a must contain the upper triangular matrix and the strictly lower triangular part of a is not referenced. Before entry

	with <i>uplo</i> = 'L' or 'l', the leading <i>n</i> -by- <i>n</i> lower triangular part of the array <i>a</i> must contain the lower triangular matrix and the strictly upper triangular part of <i>a</i> is not referenced. When <i>diag</i> = 'U' or 'u', the diagonal elements of <i>a</i> are not referenced either, but are assumed to be unity.
<i>lda</i>	INTEGER. Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, n)$.
<i>x</i>	REAL for <i>strmv</i> DOUBLE PRECISION for <i>dtrmv</i> COMPLEX for <i>ctrmv</i> DOUBLE COMPLEX for <i>ztrmv</i> Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.

Output Parameters

<i>x</i>	Overwritten with the transformed vector <i>x</i> .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *trmv* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (n, n) .
<i>x</i>	Holds the vector of length (n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.

?trsv

Solves a system of linear equations whose coefficients are in a triangular matrix.

Syntax

FORTRAN 77:

```
call strsv(uplo, trans, diag, n, a, lda, x, incx )
call dtrsv(uplo, trans, diag, n, a, lda, x, incx )
call ctrsv(uplo, trans, diag, n, a, lda, x, incx )
call ztrsv(uplo, trans, diag, n, a, lda, x, incx )
```

Fortran 95:

```
call trsv(a, x [,uplo] [, trans] [,diag])
```

Description

The ?trsv routines solve one of the systems of equations:

$A*x = b$, or $A'^*x = b$, or $\text{conjg}(A')*x = b$,

where:

b and x are n -element vectors,

A is an n -by- n unit, or non-unit, upper or lower triangular matrix.

The routine does not test for singularity or near-singularity.

Such tests must be performed before calling this routine.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the matrix A is upper or lower triangular: if <i>uplo</i> = 'U' or 'u', then the matrix is upper triangular; if <i>uplo</i> = 'L' or 'l', then the matrix is low triangular.
<i>trans</i>	CHARACTER*1. Specifies the systems of equations: if <i>trans</i> = 'N' or 'n', then $A*x = b$; if <i>trans</i> = 'T' or 't', then $A'^*x = b$; if <i>trans</i> = 'C' or 'c', then $\text{conjg}(A')*x = b$.

<i>diag</i>	CHARACTER*1. Specifies whether the matrix <i>a</i> is unit triangular: if <i>diag</i> = 'U' or 'u' then the matrix is unit triangular; if <i>diag</i> = 'N' or 'n', then the matrix is not unit triangular.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>a</i> . The value of <i>n</i> must be at least zero.
<i>a</i>	REAL for strsv DOUBLE PRECISION for dtrsv COMPLEX for ctrsv DOUBLE COMPLEX for ztrsv Array, DIMENSION (<i>lda</i> , <i>n</i>). Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>n</i> -by- <i>n</i> upper triangular part of the array <i>a</i> must contain the upper triangular matrix and the strictly lower triangular part of <i>a</i> is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', the leading <i>n</i> -by- <i>n</i> lower triangular part of the array <i>a</i> must contain the lower triangular matrix and the strictly upper triangular part of <i>a</i> is not referenced. When <i>diag</i> = 'U' or 'u', the diagonal elements of <i>a</i> are not referenced either, but are assumed to be unity.
<i>lda</i>	INTEGER. Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, n)$.
<i>x</i>	REAL for strsv DOUBLE PRECISION for dtrsv COMPLEX for ctrsv DOUBLE COMPLEX for ztrsv Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element right-hand side vector <i>b</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.

Output Parameters

<i>x</i>	Overwritten with the solution vector <i>x</i> .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `trsv` interface are the following:

<i>a</i>	Holds the matrix <i>a</i> of size (n,n) .
<i>x</i>	Holds the vector of length (n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.

BLAS Level 3 Routines

BLAS Level 3 routines perform matrix-matrix operations. [Table 2-3](#) lists the BLAS Level 3 routine groups and the data types associated with them.

Table 2-3 BLAS Level 3 Routine Groups and Their Data Types

Routine Group	Data Types	Description
?gemm	s, d, c, z	Matrix-matrix product of general matrices
?hemm	c, z	Matrix-matrix product of Hermitian matrices
?herk	c, z	Rank-k update of Hermitian matrices
?her2k	c, z	Rank-2k update of Hermitian matrices
?symm	s, d, c, z	Matrix-matrix product of symmetric matrices
?syrk	s, d, c, z	Rank-k update of symmetric matrices
?symm	s, d, c, z	Rank-2k update of symmetric matrices
?trmm	s, d, c, z	Matrix-matrix product of triangular matrices
?trsm	s, d, c, z	Linear matrix-matrix solution for triangular matrices

Symmetric Multiprocessing Version of Intel® MKL

Many applications spend considerable time for executing BLAS level 3 routines. This time can be scaled by the number of processors available on the system through using the symmetric multiprocessing(SMP) feature built into the Intel MKL Library. The performance enhancements based on the parallel use of the processors are available without any programming effort on your part.

To enhance performance, the library uses the following methods:

- The operation of BLAS level 3 matrix-matrix functions permits to restructure the code in a way which increases the localization of data reference, enhances cache memory use, and reduces the dependency on the memory bus.
- Once the code has been effectively blocked as described above, one of the matrices is distributed across the processors to be multiplied by the second matrix. Such distribution ensures effective cache management which reduces the dependency on the memory bus performance and brings good scaling results.

?gemm

Computes a scalar-matrix-matrix product and adds the result to a scalar-matrix product.

Syntax

FORTRAN 77:

```
call sgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call dgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call cgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call zgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
```

Fortran 95:

```
call gemm(a, b, c [,transa][,transb] [,alpha][,beta])
```

Description

The ?gemm routines perform a matrix-matrix operation with general matrices. The operation is defined as

$$C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C,$$

where:

$\text{op}(x)$ is one of $\text{op}(x) = x$, or $\text{op}(x) = x'$, or $\text{op}(x) = \text{conjg}(x')$,

α and β are scalars,

A , B and C are matrices:

$\text{op}(A)$ is an m -by- k matrix,

$\text{op}(B)$ is a k -by- n matrix,

C is an m -by- n matrix.

Input Parameters

<i>transa</i>	CHARACTER*1. Specifies the form of $\text{op}(A)$ used in the matrix multiplication: if <i>transa</i> = 'N' or 'n', then $\text{op}(A) = A$; if <i>transa</i> = 'T' or 't', then $\text{op}(A) = A'$; if <i>transa</i> = 'C' or 'c', then $\text{op}(A) = \text{conjg}(A')$.
<i>transb</i>	CHARACTER*1. Specifies the form of $\text{op}(B)$ used in the matrix multiplication: if <i>transb</i> = 'N' or 'n', then $\text{op}(B) = B$; if <i>transb</i> = 'T' or 't', then $\text{op}(B) = B'$; if <i>transb</i> = 'C' or 'c', then $\text{op}(B) = \text{conjg}(B')$.
<i>m</i>	INTEGER. Specifies the number of rows of the matrix $\text{op}(A)$ and of the matrix C. The value of <i>m</i> must be at least zero.
<i>n</i>	INTEGER. Specifies the number of columns of the matrix $\text{op}(B)$ and the number of columns of the matrix C. The value of <i>n</i> must be at least zero.
<i>k</i>	INTEGER. Specifies the number of columns of the matrix $\text{op}(A)$ and the number of rows of the matrix $\text{op}(B)$. The value of <i>k</i> must be at least zero.
<i>alpha</i>	REAL for sgemm DOUBLE PRECISION for dgemm COMPLEX for cgemm DOUBLE COMPLEX for zgemm Specifies the scalar <i>alpha</i>.
<i>a</i>	REAL for sgemm DOUBLE PRECISION for dgemm COMPLEX for cgemm

	DOUBLE COMPLEX for zgemm Array, DIMENSION (<i>lda</i>, <i>ka</i>), where <i>ka</i> is <i>k</i> when <i>transa</i> = 'N' or 'n', and is <i>m</i> otherwise. Before entry with <i>transa</i> = 'N' or 'n', the leading <i>m</i>-by-<i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i>, otherwise the leading <i>k</i>-by-<i>m</i> part of the array <i>a</i> must contain the matrix <i>A</i>.
<i>lda</i>	INTEGER. Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. When <i>transa</i> = 'N' or 'n', then <i>lda</i> must be at least $\max(1, m)$, otherwise <i>lda</i> must be at least $\max(1, k)$.
<i>b</i>	REAL for sgemm DOUBLE PRECISION for dgemm COMPLEX for cgemm DOUBLE COMPLEX for zgemm Array, DIMENSION (<i>ldb</i>, <i>kb</i>), where <i>kb</i> is <i>n</i> when <i>transb</i> = 'N' or 'n', and is <i>k</i> otherwise. Before entry with <i>transb</i> = 'N' or 'n', the leading <i>k</i>-by-<i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i>, otherwise the leading <i>n</i>-by-<i>k</i> part of the array <i>b</i> must contain the matrix <i>B</i>.
<i>ldb</i>	INTEGER. Specifies the first dimension of <i>b</i> as declared in the calling (sub)program. When <i>transb</i> = 'N' or 'n', then <i>ldb</i> must be at least $\max(1, k)$, otherwise <i>ldb</i> must be at least $\max(1, n)$.
<i>beta</i>	REAL for sgemm DOUBLE PRECISION for dgemm COMPLEX for cgemm DOUBLE COMPLEX for zgemm Specifies the scalar <i>beta</i>. When <i>beta</i> is equal to zero, then <i>c</i> need not be set on input.
<i>c</i>	REAL for sgemm DOUBLE PRECISION for dgemm COMPLEX for cgemm DOUBLE COMPLEX for zgemm Array, DIMENSION (<i>ldc</i>, <i>n</i>). Before entry, the leading <i>m</i>-by-<i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i>, except when <i>beta</i> is equal to zero, in which case <i>c</i> need not be set on entry.

ldc INTEGER. Specifies the first dimension of *c* as declared in the calling (sub)program. The value of *ldc* must be at least $\max(1, m)$.

Output Parameters

c Overwritten by the *m*-by-*n* matrix $(\alpha * \text{op}(A) * \text{op}(B) + \beta * C)$.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gemm` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (m_a, k_a) where $k_a = k$ if <i>transa</i> = 'N', $k_a = m$ otherwise, $m_a = m$ if <i>transa</i> = 'N', $m_a = k$ otherwise.
<i>b</i>	Holds the matrix <i>B</i> of size (m_b, k_b) where $k_b = n$ if <i>transb</i> = 'N', $k_b = k$ otherwise, $m_b = k$ if <i>transb</i> = 'N', $m_b = n$ otherwise.
<i>c</i>	Holds the matrix <i>C</i> of size (m, n) .
<i>transa</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>transb</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 1.

?gemm3m

Computes a scalar-matrix-matrix product using matrix multiplications and adds the result to a scalar-matrix product.

Syntax

FORTRAN 77:

```
call cgemm3m(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call zgemm3m(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
```

Fortran 95:

```
call gemm3m(a, b, c [,transa][,transb] [,alpha][,beta])
```

Description

The ?gemm3m routines perform a matrix-matrix operation with general complex matrices. These routines are similar to the ?gemm routines, but they used matrix multiplications (see *Application Notes* below).

The operation is defined as

$$C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C,$$

where:

$\text{op}(x)$ is one of $\text{op}(x) = x$, or $\text{op}(x) = x'$, or $\text{op}(x) = \text{conjg}(x')$,

α and β are scalars,

A , B and C are matrices:

$\text{op}(A)$ is an m -by- k matrix,

$\text{op}(B)$ is a k -by- n matrix,

C is an m -by- n matrix.

Input Parameters

<i>transa</i>	CHARACTER*1. Specifies the form of $\text{op}(A)$ used in the matrix multiplication: if <i>transa</i> = 'N' or 'n', then $\text{op}(A) = A$; if <i>transa</i> = 'T' or 't', then $\text{op}(A) = A'$;

<i>transb</i>	if <i>transa</i> = 'C' or 'c', then $\text{op}(A) = \text{conjg}(A')$. CHARACTER*1. Specifies the form of $\text{op}(B)$ used in the matrix multiplication: if <i>transb</i> = 'N' or 'n', then $\text{op}(B) = B$; if <i>transb</i> = 'T' or 't', then $\text{op}(B) = B'$; if <i>transb</i> = 'C' or 'c', then $\text{op}(B) = \text{conjg}(B')$.
<i>m</i>	INTEGER. Specifies the number of rows of the matrix $\text{op}(A)$ and of the matrix <i>C</i>. The value of <i>m</i> must be at least zero.
<i>n</i>	INTEGER. Specifies the number of columns of the matrix $\text{op}(B)$ and the number of columns of the matrix <i>C</i>. The value of <i>n</i> must be at least zero.
<i>k</i>	INTEGER. Specifies the number of columns of the matrix $\text{op}(A)$ and the number of rows of the matrix $\text{op}(B)$. The value of <i>k</i> must be at least zero.
<i>alpha</i>	COMPLEX for <i>cgemm3m</i> DOUBLE COMPLEX for <i>zgemm3m</i> Specifies the scalar <i>alpha</i>.
<i>a</i>	COMPLEX for <i>cgemm3m</i> DOUBLE COMPLEX for <i>zgemm3m</i> Array, DIMENSION (<i>lda</i>, <i>ka</i>), where <i>ka</i> is <i>k</i> when <i>transa</i> = 'N' or 'n', and is <i>m</i> otherwise. Before entry with <i>transa</i> = 'N' or 'n', the leading <i>m</i>-by-<i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i>, otherwise the leading <i>k</i>-by-<i>m</i> part of the array <i>a</i> must contain the matrix <i>A</i>.
<i>lda</i>	INTEGER. Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. When <i>transa</i> = 'N' or 'n', then <i>lda</i> must be at least $\max(1, m)$, otherwise <i>lda</i> must be at least $\max(1, k)$.
<i>b</i>	COMPLEX for <i>cgemm3m</i> DOUBLE COMPLEX for <i>zgemm3m</i> Array, DIMENSION (<i>ldb</i>, <i>kb</i>), where <i>kb</i> is <i>n</i> when <i>transb</i> = 'N' or 'n', and is <i>k</i> otherwise. Before entry with <i>transb</i> = 'N' or 'n', the leading <i>k</i>-by-<i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i>, otherwise the leading <i>n</i>-by-<i>k</i> part of the array <i>b</i> must contain the matrix <i>B</i>.

<i>ldb</i>	INTEGER. Specifies the first dimension of <i>b</i> as declared in the calling (sub)program. When <i>transb</i> = 'N' or 'n', then <i>ldb</i> must be at least $\max(1, k)$, otherwise <i>ldb</i> must be at least $\max(1, n)$.
<i>beta</i>	COMPLEX for <i>cgemm3m</i> DOUBLE COMPLEX for <i>zgemm3m</i> Specifies the scalar <i>beta</i> . When <i>beta</i> is equal to zero, then <i>c</i> need not be set on input.
<i>c</i>	COMPLEX for <i>cgemm3m</i> DOUBLE COMPLEX for <i>zgemm3m</i> Array, DIMENSION (<i>ldc</i> , <i>n</i>). Before entry, the leading <i>m</i> -by- <i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i> , except when <i>beta</i> is equal to zero, in which case <i>c</i> need not be set on entry.
<i>ldc</i>	INTEGER. Specifies the first dimension of <i>c</i> as declared in the calling (sub)program. The value of <i>ldc</i> must be at least $\max(1, m)$.

Output Parameters

<i>c</i>	Overwritten by the <i>m</i> -by- <i>n</i> matrix $(\alpha * \text{op}(A) * \text{op}(B) + \beta * C)$.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *gemm3m* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>ma</i> , <i>ka</i>) where <i>ka</i> = <i>k</i> if <i>transa</i> = 'N', <i>ka</i> = <i>m</i> otherwise, <i>ma</i> = <i>m</i> if <i>transa</i> = 'N', <i>ma</i> = <i>k</i> otherwise.
<i>b</i>	Holds the matrix <i>B</i> of size (<i>mb</i> , <i>kb</i>) where <i>kb</i> = <i>n</i> if <i>transb</i> = 'N', <i>kb</i> = <i>k</i> otherwise,

	$mb = k$ if $transb = 'N'$, $mb = n$ otherwise.
c	Holds the matrix c of size (m,n) .
$transa$	Must be 'N', 'C', or 'T'. The default value is 'N'.
$transb$	Must be 'N', 'C', or 'T'. The default value is 'N'.
$alpha$	The default value is 1.
$beta$	The default value is 1.

Application Notes

These routines perform the complex multiplication by forming the real and imaginary parts of the input matrices. It allows to use three real matrix multiplications and five real matrix additions, instead of the conventional four real matrix multiplications and two real matrix additions. The use of three real matrix multiplications only gives a 25% reduction of time in matrix operations. This can result in significant savings in computing time for large matrices.

If the errors in the floating point calculations satisfy the following conditions:

$$fl(x \text{ op } y) = (x \text{ op } y) (1 + \delta), |\delta| \leq u, \text{ op} = +, -, fl(x \pm y) = (x \pm y) (1 + \alpha) \pm y (1 + \beta), |\alpha|, |\beta| \leq u$$

then for n -by- n matrix $\hat{C} = fl(C_1 + iC_2) = fl((A_1 + iA_2)(B_1 + iB_2)) = \hat{C}_1 + i\hat{C}_2$ the following estimations are correct

$$\|\hat{C}_1 - C_2\| \leq 2(n+1)u\|A\|_{\infty}\|B\|_{\infty} + O(u^2),$$

$$\|\hat{C}_2 - C_1\| \leq 4(n+4)u\|A\|_{\infty}\|B\|_{\infty} + O(u^2),$$

$$\text{where } \|A\|_{\infty} = \max(\|A_1\|_{\infty}, \|A_2\|_{\infty}), \text{ and } \|B\|_{\infty} = \max(\|B_1\|_{\infty}, \|B_2\|_{\infty}).$$

and hence the matrix multiplications are stable.

?hemm

Computes a scalar-matrix-matrix product (either one of the matrices is Hermitian) and adds the result to scalar-matrix product.

Syntax

FORTRAN 77:

```
call chemm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc )
call zhemm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc )
```

Fortran 95:

```
call hemm(a, b, c [,side][,uplo] [,alpha][,beta])
```

Description

The ?hemm routines perform a matrix-matrix operation using Hermitian matrices. The operation is defined as

$$C := \alpha * A * B + \beta * C$$

or

$$C := \alpha * B * A + \beta * C,$$

where:

α and β are scalars,

A is an Hermitian matrix,

B and C are m -by- n matrices.

Input Parameters

<i>side</i>	CHARACTER*1. Specifies whether the Hermitian matrix A appears on the left or right in the operation as follows: if $side = 'L'$ or $'l'$, then $C := \alpha * A * B + \beta * C$; if $side = 'R'$ or $'r'$, then $C := \alpha * B * A + \beta * C$.
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian matrix A is used: If $uplo = 'U'$ or $'u'$, then the upper triangular part of the Hermitian matrix A is used.

	If <i>uplo</i> = 'L' or 'l', then the low triangular part of the Hermitian matrix <i>A</i> is used.
<i>m</i>	INTEGER. Specifies the number of rows of the matrix <i>C</i> . The value of <i>m</i> must be at least zero.
<i>n</i>	INTEGER. Specifies the number of columns of the matrix <i>C</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	COMPLEX for chemm DOUBLE COMPLEX for zhemm Specifies the scalar <i>alpha</i> .
<i>a</i>	COMPLEX for chemm DOUBLE COMPLEX for zhemm Array, DIMENSION (<i>lda</i> , <i>ka</i>), where <i>ka</i> is <i>m</i> when <i>side</i> = 'L' or 'l' and is <i>n</i> otherwise. Before entry with <i>side</i> = 'L' or 'l', the <i>m</i> -by- <i>m</i> part of the array <i>a</i> must contain the Hermitian matrix, such that when <i>uplo</i> = 'U' or 'u', the leading <i>m</i> -by- <i>m</i> upper triangular part of the array <i>a</i> must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of <i>a</i> is not referenced, and when <i>uplo</i> = 'L' or 'l', the leading <i>m</i> -by- <i>m</i> lower triangular part of the array <i>a</i> must contain the lower triangular part of the Hermitian matrix, and the strictly upper triangular part of <i>a</i> is not referenced. Before entry with <i>side</i> = 'R' or 'r', the <i>n</i> -by- <i>n</i> part of the array <i>a</i> must contain the Hermitian matrix, such that when <i>uplo</i> = 'U' or 'u', the leading <i>n</i> -by- <i>n</i> upper triangular part of the array <i>a</i> must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of <i>a</i> is not referenced, and when <i>uplo</i> = 'L' or 'l', the leading <i>n</i> -by- <i>n</i> lower triangular part of the array <i>a</i> must contain the lower triangular part of the Hermitian matrix, and the strictly upper triangular part of <i>a</i> is not referenced. The imaginary parts of the diagonal elements need not be set, they are assumed to be zero.
<i>lda</i>	INTEGER. Specifies the first dimension of <i>a</i> as declared in the calling (sub) program. When <i>side</i> = 'L' or 'l' then <i>lda</i> must be at least $\max(1, m)$, otherwise <i>lda</i> must be at least $\max(1, n)$.

<i>b</i>	COMPLEX for chemm DOUBLE COMPLEX for zhemm Array, DIMENSION (<i>ldb</i>,<i>n</i>). Before entry, the leading <i>m</i>-by-<i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i>.
<i>ldb</i>	INTEGER. Specifies the first dimension of <i>b</i> as declared in the calling (sub)program. The value of <i>ldb</i> must be at least $\max(1, m)$.
<i>beta</i>	COMPLEX for chemm DOUBLE COMPLEX for zhemm Specifies the scalar <i>beta</i>. When <i>beta</i> is supplied as zero, then <i>c</i> need not be set on input.
<i>c</i>	COMPLEX for chemm DOUBLE COMPLEX for zhemm Array, DIMENSION (<i>c</i>, <i>n</i>). Before entry, the leading <i>m</i>-by-<i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i>, except when <i>beta</i> is zero, in which case <i>c</i> need not be set on entry.
<i>ldc</i>	INTEGER. Specifies the first dimension of <i>c</i> as declared in the calling (sub)program. The value of <i>ldc</i> must be at least $\max(1, m)$.

Output Parameters

<i>c</i>	Overwritten by the <i>m</i> -by- <i>n</i> updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hemm` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>k</i>,<i>k</i>) where $k = m$ if <i>side</i> = 'L', $k = n$ otherwise.
<i>b</i>	Holds the matrix <i>B</i> of size (<i>m</i> , <i>n</i>).
<i>c</i>	Holds the matrix <i>C</i> of size (<i>m</i> , <i>n</i>).

<i>side</i>	Must be 'L' or 'R'. The default value is 'L'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 1.

?herk

Performs a rank- n update of a Hermitian matrix.

Syntax

FORTRAN 77:

```
call cherk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc )
call zherk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc )
```

Fortran 95:

```
call herk(a, c [,uplo] [, trans] [,alpha][,beta])
```

Description

The ?herk routines perform a matrix-matrix operation using Hermitian matrices. The operation is defined as

$$C := \alpha * A * \text{conjg}(A') + \beta * C,$$

or

$$C := \alpha * \text{conjg}(A') * A + \beta * C,$$

where:

alpha and *beta* are real scalars,

C is an n -by- n Hermitian matrix,

A is an n -by- k matrix in the first case and a k -by- n matrix in the second case.

Input Parameters

uplo CHARACTER*1. Specifies whether the upper or lower triangular part of the array *c* is used.
If *uplo* = 'U' or 'u', then the upper triangular part of the array *c* is used.

	If <code>uplo = 'L' or 'l'</code>, then the low triangular part of the array <code>c</code> is used.
<code>trans</code>	CHARACTER*1. Specifies the operation: if <code>trans = 'N' or 'n'</code>, then <code>C:= alpha*A*conjg(A')+beta*C</code>; if <code>trans = 'C' or 'c'</code>, then <code>C:= alpha*conjg(A')*A+beta*C</code>.
<code>n</code>	INTEGER. Specifies the order of the matrix <code>C</code>. The value of <code>n</code> must be at least zero.
<code>k</code>	INTEGER. With <code>trans = 'N' or 'n'</code>, <code>k</code> specifies the number of columns of the matrix <code>A</code>, and with <code>trans = 'C' or 'c'</code>, <code>k</code> specifies the number of rows of the matrix <code>A</code>. The value of <code>k</code> must be at least zero.
<code>alpha</code>	REAL for <code>cherk</code> DOUBLE PRECISION for <code>zherk</code> Specifies the scalar <code>alpha</code>.
<code>a</code>	COMPLEX for <code>cherk</code> DOUBLE COMPLEX for <code>zherk</code> Array, DIMENSION (<code>lda</code>, <code>ka</code>), where <code>ka</code> is <code>k</code> when <code>trans = 'N' or 'n'</code>, and is <code>n</code> otherwise. Before entry with <code>trans = 'N' or 'n'</code>, the leading <code>n</code>-by-<code>k</code> part of the array <code>a</code> must contain the matrix <code>a</code>, otherwise the leading <code>k</code>-by-<code>n</code> part of the array <code>a</code> must contain the matrix <code>A</code>.
<code>lda</code>	INTEGER. Specifies the first dimension of <code>a</code> as declared in the calling (sub)program. When <code>trans = 'N' or 'n'</code>, then <code>lda</code> must be at least <code>max(1, n)</code>, otherwise <code>lda</code> must be at least <code>max(1, k)</code>.
<code>beta</code>	REAL for <code>cherk</code> DOUBLE PRECISION for <code>zherk</code> Specifies the scalar <code>beta</code>.
<code>c</code>	COMPLEX for <code>cherk</code> DOUBLE COMPLEX for <code>zherk</code> Array, DIMENSION (<code>ldc</code>,<code>n</code>).

Before entry with `uplo = 'U' or 'u'`, the leading n -by- n upper triangular part of the array `c` must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of `c` is not referenced.

Before entry with `uplo = 'L' or 'l'`, the leading n -by- n lower triangular part of the array `c` must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of `c` is not referenced.

The imaginary parts of the diagonal elements need not be set, they are assumed to be zero.

`ldc`

INTEGER. Specifies the first dimension of `c` as declared in the calling (sub)program. The value of `ldc` must be at least $\max(1, n)$.

Output Parameters

`c`

With `uplo = 'U' or 'u'`, the upper triangular part of the array `c` is overwritten by the upper triangular part of the updated matrix.

With `uplo = 'L' or 'l'`, the lower triangular part of the array `c` is overwritten by the lower triangular part of the updated matrix.

The imaginary parts of the diagonal elements are set to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `herk` interface are the following:

`a`

Holds the matrix `A` of size (ma, ka) where

$ka = k$ if `transa = 'N'`,

$ka = n$ otherwise,

$ma = n$ if `transa = 'N'`,

$ma = k$ otherwise.

`c`

Holds the matrix `C` of size (n, n) .

`uplo`

Must be 'U' or 'L'. The default value is 'U'.

<i>trans</i>	Must be 'N' or 'C'. The default value is 'N'.
<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 1.

?her2k

Performs a rank-2k update of a Hermitian matrix.

Syntax

FORTRAN 77:

```
call cher2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call zher2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
```

Fortran 95:

```
call her2k(a, b, c [,uplo][,trans] [,alpha][,beta])
```

Description

The ?her2k routines perform a rank-2k matrix-matrix operation using Hermitian matrices. The operation is defined as

$$C := \alpha A * \text{conjg}(B') + \text{conjg}(\alpha) B * \text{conjg}(A') + \beta C,$$

or

$$C := \alpha * \text{conjg}(B') * A + \text{conjg}(\alpha) * \text{conjg}(A') * B + \beta C,$$

where:

alpha is a scalar and *beta* is a real scalar,

C is an *n*-by-*n* Hermitian matrix,

A and *B* are *n*-by-*k* matrices in the first case and *k*-by-*n* matrices in the second case.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>c</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangular of the array <i>c</i> is used.

	If <i>uplo</i> = 'L' or 'l', then the low triangular of the array <i>c</i> is used.
<i>trans</i>	CHARACTER*1. Specifies the operation: if <i>trans</i> = 'N' or 'n', then $C := \alpha * A * \text{conjg}(B') + \alpha * B * \text{conjg}(A') + \beta * C$; if <i>trans</i> = 'C' or 'c', then $C := \alpha * \text{conjg}(A') * B + \alpha * \text{conjg}(B') * A + \beta * C$.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>C</i> . The value of <i>n</i> must be at least zero.
<i>k</i>	INTEGER. With <i>trans</i> = 'N' or 'n', <i>k</i> specifies the number of columns of the matrix <i>A</i> , and with <i>trans</i> = 'C' or 'c', <i>k</i> specifies the number of rows of the matrix <i>A</i> . The value of <i>k</i> must be at least equal to zero.
<i>alpha</i>	COMPLEX for cher2k DOUBLE COMPLEX for zher2k Specifies the scalar <i>alpha</i> .
<i>a</i>	COMPLEX for cher2k DOUBLE COMPLEX for zher2k Array, DIMENSION (<i>lda</i> , <i>ka</i>), where <i>ka</i> is <i>k</i> when <i>trans</i> = 'N' or 'n', and is <i>n</i> otherwise. Before entry with <i>trans</i> = 'N' or 'n', the leading <i>n</i> -by- <i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i> , otherwise the leading <i>k</i> -by- <i>n</i> part of the array <i>a</i> must contain the matrix <i>A</i> .
<i>lda</i>	INTEGER. Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. When <i>trans</i> = 'N' or 'n', then <i>lda</i> must be at least $\max(1, n)$, otherwise <i>lda</i> must be at least $\max(1, k)$.
<i>beta</i>	REAL for cher2k DOUBLE PRECISION for zher2k Specifies the scalar <i>beta</i> .
<i>b</i>	COMPLEX for cher2k DOUBLE COMPLEX for zher2k

Array, DIMENSION (ldb , kb), where kb is k when $trans = 'N'$ or $'n'$, and is n otherwise. Before entry with $trans = 'N'$ or $'n'$, the leading n -by- k part of the array b must contain the matrix B , otherwise the leading k -by- n part of the array b must contain the matrix B .

ldb INTEGER. Specifies the first dimension of b as declared in the calling (sub)program. When $trans = 'N'$ or $'n'$, then ldb must be at least $\max(1, n)$, otherwise ldb must be at least $\max(1, k)$.

c COMPLEX for cher2k
DOUBLE COMPLEX for zher2k
Array, DIMENSION (ldc, n).
Before entry with $uplo = 'U'$ or $'u'$, the leading n -by- n upper triangular part of the array c must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of c is not referenced.
Before entry with $uplo = 'L'$ or $'l'$, the leading n -by- n lower triangular part of the array c must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of c is not referenced.
The imaginary parts of the diagonal elements need not be set, they are assumed to be zero.

ldc INTEGER. Specifies the first dimension of c as declared in the calling (sub)program. The value of ldc must be at least $\max(1, n)$.

Output Parameters

c With $uplo = 'U'$ or $'u'$, the upper triangular part of the array c is overwritten by the upper triangular part of the updated matrix.
With $uplo = 'L'$ or $'l'$, the lower triangular part of the array c is overwritten by the lower triangular part of the updated matrix.
The imaginary parts of the diagonal elements are set to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `her2k` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (m_a, k_a) where $k_a = k$ if <i>trans</i> = 'N', $k_a = n$ otherwise, $m_a = n$ if <i>trans</i> = 'N', $m_a = k$ otherwise.
<i>b</i>	Holds the matrix <i>B</i> of size (m_b, k_b) where $k_b = k$ if <i>trans</i> = 'N', $k_b = n$ otherwise, $m_b = n$ if <i>trans</i> = 'N', $m_b = k$ otherwise.
<i>c</i>	Holds the matrix <i>C</i> of size (n, n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N' or 'C'. The default value is 'N'.
<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 1.

?symm

Performs a scalar-matrix-matrix product(one matrix operand is symmetric) and adds the result to a scalar-matrix product.

Syntax

FORTRAN 77:

```
call ssymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc )
call dsymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc )
call csymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc )
call zsymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc )
```

Fortran 95:

```
call symm(a, b, c [,side][,uplo] [,alpha][,beta])
```

Description

The ?`symm` routines perform a matrix-matrix operation using symmetric matrices. The operation is defined as

$$C := \alpha * A * B + \beta * C,$$

or

$$C := \alpha * B * A + \beta * C,$$

where:

alpha and *beta* are scalars,

A is a symmetric matrix,

B and *C* are *m*-by-*n* matrices.

Input Parameters

<i>side</i>	CHARACTER*1. Specifies whether the symmetric matrix <i>A</i> appears on the left or right in the operation: if <i>side</i> = 'L' or 'l', then $C := \alpha * A * B + \beta * C$; if <i>side</i> = 'R' or 'r', then $C := \alpha * B * A + \beta * C$.
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix <i>A</i> is used: if <i>uplo</i> = 'U' or 'u', then the upper triangular part is used; if <i>uplo</i> = 'L' or 'l', then the lower triangular part is used.
<i>m</i>	INTEGER. Specifies the number of rows of the matrix <i>C</i> . The value of <i>m</i> must be at least zero.
<i>n</i>	INTEGER. Specifies the number of columns of the matrix <i>C</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for <code>ssymm</code> DOUBLE PRECISION for <code>dsymm</code> COMPLEX for <code>csymm</code> DOUBLE COMPLEX for <code>zsymm</code> Specifies the scalar <i>alpha</i> .

<i>a</i>	REAL for ssymm DOUBLE PRECISION for dsymm COMPLEX for csymm DOUBLE COMPLEX for zsymm Array, DIMENSION (<i>lda</i>, <i>ka</i>), where <i>ka</i> is <i>m</i> when <i>side</i> = 'L' or 'l' and is <i>n</i> otherwise. Before entry with <i>side</i> = 'L' or 'l', the <i>m</i>-by-<i>m</i> part of the array <i>a</i> must contain the symmetric matrix, such that when <i>uplo</i> = 'U' or 'u', the leading <i>m</i>-by-<i>m</i> upper triangular part of the array <i>a</i> must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of <i>a</i> is not referenced, and when <i>uplo</i> = 'L' or 'l', the leading <i>m</i>-by-<i>m</i> lower triangular part of the array <i>a</i> must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of <i>a</i> is not referenced. Before entry with <i>side</i> = 'R' or 'r', the <i>n</i>-by-<i>n</i> part of the array <i>a</i> must contain the symmetric matrix, such that when <i>uplo</i> = 'U' or 'u', the leading <i>n</i>-by-<i>n</i> upper triangular part of the array <i>a</i> must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of <i>a</i> is not referenced, and when <i>uplo</i> = 'L' or 'l', the leading <i>n</i>-by-<i>n</i> lower triangular part of the array <i>a</i> must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of <i>a</i> is not referenced.
<i>lda</i>	INTEGER. Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. When <i>side</i> = 'L' or 'l' then <i>lda</i> must be at least $\max(1, m)$, otherwise <i>lda</i> must be at least $\max(1, n)$.
<i>b</i>	REAL for ssymm DOUBLE PRECISION for dsymm COMPLEX for csymm DOUBLE COMPLEX for zsymm Array, DIMENSION (<i>ldb</i>, <i>n</i>). Before entry, the leading <i>m</i>-by-<i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i>.

<i>ldb</i>	INTEGER. Specifies the first dimension of <i>b</i> as declared in the calling (sub)program. The value of <i>ldb</i> must be at least $\max(1, m)$.
<i>beta</i>	REAL for <i>ssymm</i> DOUBLE PRECISION for <i>dsymm</i> COMPLEX for <i>csymm</i> DOUBLE COMPLEX for <i>zsymm</i> Specifies the scalar <i>beta</i> . When <i>beta</i> is set to zero, then <i>c</i> need not be set on input.
<i>c</i>	REAL for <i>ssymm</i> DOUBLE PRECISION for <i>dsymm</i> COMPLEX for <i>csymm</i> DOUBLE COMPLEX for <i>zsymm</i> Array, DIMENSION (<i>ldc</i> , <i>n</i>). Before entry, the leading <i>m</i> -by- <i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i> , except when <i>beta</i> is zero, in which case <i>c</i> need not be set on entry.
<i>ldc</i>	INTEGER. Specifies the first dimension of <i>c</i> as declared in the calling (sub)program. The value of <i>ldc</i> must be at least $\max(1, m)$.

Output Parameters

<i>c</i>	Overwritten by the <i>m</i> -by- <i>n</i> updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *symm* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>k</i> , <i>k</i>) where <i>k</i> = <i>m</i> if <i>side</i> = 'L', <i>k</i> = <i>n</i> otherwise.
<i>b</i>	Holds the matrix <i>B</i> of size (<i>m</i> , <i>n</i>).
<i>c</i>	Holds the matrix <i>C</i> of size (<i>m</i> , <i>n</i>).
<i>side</i>	Must be 'L' or 'R'. The default value is 'L'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 1.

?syrk

Performs a rank- n update of a symmetric matrix.

Syntax

FORTRAN 77:

```
call ssyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc )
call dsyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc )
call csyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc )
call zsyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc )
```

Fortran 95:

```
call syrk(a, c [,uplo] [, trans] [,alpha][,beta])
```

Description

The ?syrk routines perform a matrix-matrix operation using symmetric matrices. The operation is defined as

$$C := \alpha * A * A' + \beta * C,$$

or

$$C := \alpha * A' * A + \beta * C,$$

where:

alpha and *beta* are scalars,

C is an n -by- n symmetric matrix,

A is an n -by- k matrix in the first case and a k -by- n matrix in the second case.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>c</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the array <i>c</i> is used.

	If <i>uplo</i> = 'L' or 'l', then the low triangular part of the array <i>c</i> is used.
<i>trans</i>	CHARACTER*1. Specifies the operation: if <i>trans</i> = 'N' or 'n', then $C := \alpha * A * A' + \beta * C$; if <i>trans</i> = 'T' or 't', then $C := \alpha * A' * A + \beta * C$; if <i>trans</i> = 'C' or 'c', then $C := \alpha * A' * A + \beta * C$.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>c</i>. The value of <i>n</i> must be at least zero.
<i>k</i>	INTEGER. On entry with <i>trans</i> = 'N' or 'n', <i>k</i> specifies the number of columns of the matrix <i>a</i>, and on entry with <i>trans</i> = 'T' or 't' or 'C' or 'c', <i>k</i> specifies the number of rows of the matrix <i>a</i>. The value of <i>k</i> must be at least zero.
<i>alpha</i>	REAL for ssyrk DOUBLE PRECISION for dsyrk COMPLEX for csyrk DOUBLE COMPLEX for zsyrk Specifies the scalar <i>alpha</i>.
<i>a</i>	REAL for ssyrk DOUBLE PRECISION for dsyrk COMPLEX for csyrk DOUBLE COMPLEX for zsyrk Array, DIMENSION (<i>lda</i>, <i>ka</i>), where <i>ka</i> is <i>k</i> when <i>trans</i> = 'N' or 'n', and is <i>n</i> otherwise. Before entry with <i>trans</i> = 'N' or 'n', the leading <i>n</i>-by-<i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i>, otherwise the leading <i>k</i>-by-<i>n</i> part of the array <i>a</i> must contain the matrix <i>A</i>.
<i>lda</i>	INTEGER. Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. When <i>trans</i> = 'N' or 'n', then <i>lda</i> must be at least $\max(1, n)$, otherwise <i>lda</i> must be at least $\max(1, k)$.
<i>beta</i>	REAL for ssyrk DOUBLE PRECISION for dsyrk COMPLEX for csyrk DOUBLE COMPLEX for zsyrk Specifies the scalar <i>beta</i>.

c REAL for ssyrk
DOUBLE PRECISION for dsyrk
COMPLEX for csyrk
DOUBLE COMPLEX for zsyrk
Array, DIMENSION (*ldc*, *n*). Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *c* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *c* is not referenced.
Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *c* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *c* is not referenced.

ldc INTEGER. Specifies the first dimension of *c* as declared in the calling (sub)program. The value of *ldc* must be at least $\max(1, n)$.

Output Parameters

c With *uplo* = 'U' or 'u', the upper triangular part of the array *c* is overwritten by the upper triangular part of the updated matrix.
With *uplo* = 'L' or 'l', the lower triangular part of the array *c* is overwritten by the lower triangular part of the updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `syrk` interface are the following:

a Holds the matrix *A* of size (*ma*, *ka*) where
 $ka = k$ if *transa* = 'N',
 $ka = n$ otherwise,
 $ma = n$ if *transa* = 'N',
 $ma = k$ otherwise.

c Holds the matrix *C* of size (*n*, *n*).

<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 1.

?syr2k

Performs a rank-2k update of a symmetric matrix.

Syntax

FORTRAN 77:

```
call ssyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc )
call dsyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc )
call csyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc )
call zsyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc )
```

Fortran 95:

```
call syr2k(a, b, c [,uplo][,trans] [,alpha][,beta])
```

Description

The ?syr2k routines perform a rank-2k matrix-matrix operation using symmetric matrices. The operation is defined as

$$C := \alpha A B' + \alpha B A' + \beta C,$$

or

$$C := \alpha A' B + \alpha B' A + \beta C,$$

where:

alpha and *beta* are scalars,

C is an *n*-by-*n* symmetric matrix,

A and *B* are *n*-by-*k* matrices in the first case, and *k*-by-*n* matrices in the second case.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>c</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the array <i>c</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the array <i>c</i> is used.
<i>trans</i>	CHARACTER*1. Specifies the operation: if <i>trans</i> = 'N' or 'n', then $C := \alpha * A * B' + \alpha * B * A' + \beta * C$; if <i>trans</i> = 'T' or 't', then $C := \alpha * A' * B + \alpha * B' * A + \beta * C$; if <i>trans</i> = 'C' or 'c', then $C := \alpha * A' * B + \alpha * B' * A + \beta * C$.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>C</i>. The value of <i>n</i> must be at least zero.
<i>k</i>	INTEGER. On entry with <i>trans</i> = 'N' or 'n', <i>k</i> specifies the number of columns of the matrices <i>A</i> and <i>B</i>, and on entry with <i>trans</i> = 'T' or 't' or 'C' or 'c', <i>k</i> specifies the number of rows of the matrices <i>A</i> and <i>B</i>. The value of <i>k</i> must be at least zero.
<i>alpha</i>	REAL for ssyr2k DOUBLE PRECISION for dsyr2k COMPLEX for csyr2k DOUBLE COMPLEX for zsyr2k Specifies the scalar <i>alpha</i>.
<i>a</i>	REAL for ssyr2k DOUBLE PRECISION for dsyr2k COMPLEX for csyr2k DOUBLE COMPLEX for zsyr2k Array, DIMENSION (<i>lda</i>, <i>ka</i>), where <i>ka</i> is <i>k</i> when <i>trans</i> = 'N' or 'n', and is <i>n</i> otherwise. Before entry with <i>trans</i> = 'N' or 'n', the leading <i>n</i>-by-<i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i>, otherwise the leading <i>k</i>-by-<i>n</i> part of the array <i>a</i> must contain the matrix <i>A</i>.

<i>lda</i>	INTEGER. Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. When <i>trans</i> = 'N' or 'n', then <i>lda</i> must be at least $\max(1, n)$, otherwise <i>lda</i> must be at least $\max(1, k)$.
<i>b</i>	REAL for ssyr2k DOUBLE PRECISION for dsyr2k COMPLEX for csyr2k DOUBLE COMPLEX for zsyr2k Array, DIMENSION (<i>ldb</i>, <i>kb</i>) where <i>kb</i> is <i>k</i> when <i>trans</i> = 'N' or 'n' and is 'n' otherwise. Before entry with <i>trans</i> = 'N' or 'n', the leading <i>n</i>-by-<i>k</i> part of the array <i>b</i> must contain the matrix <i>B</i>, otherwise the leading <i>k</i>-by-<i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i>.
<i>ldb</i>	INTEGER. Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. When <i>trans</i> = 'N' or 'n', then <i>ldb</i> must be at least $\max(1, n)$, otherwise <i>ldb</i> must be at least $\max(1, k)$.
<i>beta</i>	REAL for ssyr2k DOUBLE PRECISION for dsyr2k COMPLEX for csyr2k DOUBLE COMPLEX for zsyr2k Specifies the scalar <i>beta</i>.
<i>c</i>	REAL for ssyr2k DOUBLE PRECISION for dsyr2k COMPLEX for csyr2k DOUBLE COMPLEX for zsyr2k Array, DIMENSION (<i>ldc</i>, <i>n</i>). Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>n</i>-by-<i>n</i> upper triangular part of the array <i>c</i> must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of <i>c</i> is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', the leading <i>n</i>-by-<i>n</i> lower triangular part of the array <i>c</i> must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of <i>c</i> is not referenced.

ldc INTEGER. Specifies the first dimension of *c* as declared in the calling (sub)program. The value of *ldc* must be at least $\max(1, n)$.

Output Parameters

c With *uplo* = 'U' or 'u', the upper triangular part of the array *c* is overwritten by the upper triangular part of the updated matrix.
With *uplo* = 'L' or 'l', the lower triangular part of the array *c* is overwritten by the lower triangular part of the updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *syr2k* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (ma, ka) where $ka = k$ if <i>trans</i> = 'N', $ka = n$ otherwise, $ma = n$ if <i>trans</i> = 'N', $ma = k$ otherwise.
<i>b</i>	Holds the matrix <i>B</i> of size (mb, kb) where $kb = k$ if <i>trans</i> = 'N', $kb = n$ otherwise, $mb = n$ if <i>trans</i> = 'N', $mb = k$ otherwise.
<i>c</i>	Holds the matrix <i>C</i> of size (n, n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 1.

?trmm

Computes a scalar-matrix-matrix product (one matrix operand is triangular).

Syntax

FORTRAN 77:

```
call strmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb )
call dtrmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb )
call ctrmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb )
call ztrmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb )
```

Fortran 95:

```
call trmm(a, b [,side] [, uplo] [,transa][,diag] [,alpha])
```

Description

The ?trmm routines perform a matrix-matrix operation using triangular matrices. The operation is defined as

$B := \alpha * \text{op}(A) * B$

or

$B := \alpha * B * \text{op}(A)$

where:

α is a scalar,

B is an m -by- n matrix,

A is a unit, or non-unit, upper or lower triangular matrix

$\text{op}(A)$ is one of $\text{op}(A) = A$, or $\text{op}(A) = A'$, or $\text{op}(A) = \text{conjg}(A')$.

Input Parameters

<i>side</i>	CHARACTER*1. Specifies whether $\text{op}(A)$ appears on the left or right of B in the operation: if <i>side</i> = 'L' or 'l', then $B := \alpha * \text{op}(A) * B$; if <i>side</i> = 'R' or 'r', then $B := \alpha * B * \text{op}(A)$.

<i>uplo</i>	CHARACTER*1. Specifies whether the matrix <i>A</i> is upper or lower triangular: if <i>uplo</i> = 'U' or 'u', then the matrix is upper triangular; if <i>uplo</i> = 'L' or 'l', then the matrix is low triangular.
<i>transa</i>	CHARACTER*1. Specifies the form of $\text{op}(A)$ used in the matrix multiplication: if <i>transa</i> = 'N' or 'n', then $\text{op}(A) = A$; if <i>transa</i> = 'T' or 't', then $\text{op}(A) = A'$; if <i>transa</i> = 'C' or 'c', then $\text{op}(A) = \text{conjg}(A')$.
<i>diag</i>	CHARACTER*1. Specifies whether the matrix <i>A</i> is unit triangular: if <i>diag</i> = 'U' or 'u' then the matrix is unit triangular; if <i>diag</i> = 'N' or 'n', then the matrix is not unit triangular.
<i>m</i>	INTEGER. Specifies the number of rows of <i>B</i>. The value of <i>m</i> must be at least zero.
<i>n</i>	INTEGER. Specifies the number of columns of <i>B</i>. The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for <i>strmm</i> DOUBLE PRECISION for <i>dtrmm</i> COMPLEX for <i>ctrmm</i> DOUBLE COMPLEX for <i>ztrmm</i> Specifies the scalar <i>alpha</i>. When <i>alpha</i> is zero, then <i>a</i> is not referenced and <i>b</i> need not be set before entry.
<i>a</i>	REAL for <i>strmm</i> DOUBLE PRECISION for <i>dtrmm</i> COMPLEX for <i>ctrmm</i> DOUBLE COMPLEX for <i>ztrmm</i> Array, DIMENSION (<i>lda</i>,<i>k</i>), where <i>k</i> is <i>m</i> when <i>side</i> = 'L' or 'l' and is <i>n</i> when <i>side</i> = 'R' or 'r'. Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>k</i> by <i>k</i> upper triangular part of the array <i>a</i> must contain the upper triangular matrix and the strictly lower triangular part of <i>a</i> is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', the leading <i>k</i> by <i>k</i> lower triangular part of the array <i>a</i> must contain the lower triangular matrix and the strictly upper triangular part of <i>a</i> is not referenced.

	When <i>diag</i> = 'U' or 'u', the diagonal elements of <i>a</i> are not referenced either, but are assumed to be unity.
<i>lda</i>	INTEGER. Specifies the first dimension of <i>a</i> as declared in the calling (sub)program. When <i>side</i> = 'L' or 'l', then <i>lda</i> must be at least $\max(1, m)$, when <i>side</i> = 'R' or 'r', then <i>lda</i> must be at least $\max(1, n)$.
<i>b</i>	REAL for <i>strmm</i> DOUBLE PRECISION for <i>dtrmm</i> COMPLEX for <i>ctrmm</i> DOUBLE COMPLEX for <i>ztrmm</i> Array, DIMENSION (<i>ldb</i> , <i>n</i>). Before entry, the leading <i>m</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i> .
<i>ldb</i>	INTEGER. Specifies the first dimension of <i>b</i> as declared in the calling (sub)program. The value of <i>ldb</i> must be at least $\max(1, m)$.

Output Parameters

<i>b</i>	Overwritten by the transformed matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *trmm* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>k</i> , <i>k</i>) where <i>k</i> = <i>m</i> if <i>side</i> = 'L', <i>k</i> = <i>n</i> otherwise.
<i>b</i>	Holds the matrix <i>B</i> of size (<i>m</i> , <i>n</i>).
<i>side</i>	Must be 'L' or 'R'. The default value is 'L'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>transa</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.

alpha

The default value is 1.

?trsm

Solves a matrix equation (one matrix operand is triangular).

Syntax

FORTRAN 77:

```
call strsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb )
call dtrsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb )
call ctrsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb )
call ztrsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb )
```

Fortran 95:

```
call trsm(a, b [,side] [, uplo] [,transa][,diag] [,alpha])
```

Description

The ?trsm routines solve one of the following matrix equations:

$$\text{op}(A) * X = \alpha * B,$$

or

$$X * \text{op}(A) = \alpha * B,$$

where:

alpha is a scalar,

X and *B* are *m*-by-*n* matrices,

A is a unit, or non-unit, upper or lower triangular matrix

op(*A*) is one of *op*(*A*) = *A*, or *op*(*A*) = *A'*, or *op*(*A*) = *conjg*(*A'*).

The matrix *B* is overwritten by the solution matrix *X*.

Input Parameters

side

CHARACTER*1. Specifies whether *op*(*A*) appears on the left or right of *X* in the equation:

	if <i>side</i> = 'L' or 'l', then $\text{op}(A) * X = \alpha * B$; if <i>side</i> = 'R' or 'r', then $X * \text{op}(A) = \alpha * B$.
<i>uplo</i>	CHARACTER*1. Specifies whether the matrix <i>A</i> is upper or lower triangular: if <i>uplo</i> = 'U' or 'u', then the matrix is upper triangular; if <i>uplo</i> = 'L' or 'l', then the matrix is low triangular.
<i>transa</i>	CHARACTER*1. Specifies the form of $\text{op}(A)$ used in the matrix multiplication: if <i>transa</i> = 'N' or 'n', then $\text{op}(A) = A$; if <i>transa</i> = 'T' or 't', then $\text{op}(A) = A'$; if <i>transa</i> = 'C' or 'c', then $\text{op}(A) = \text{conjg}(A)$.
<i>diag</i>	CHARACTER*1. Specifies whether the matrix <i>A</i> is unit triangular: if <i>diag</i> = 'U' or 'u' then the matrix is unit triangular; if <i>diag</i> = 'N' or 'n', then the matrix is not unit triangular.
<i>m</i>	INTEGER. Specifies the number of rows of <i>B</i>. The value of <i>m</i> must be at least zero.
<i>n</i>	INTEGER. Specifies the number of columns of <i>B</i>. The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for <i>strsm</i> DOUBLE PRECISION for <i>dtrsm</i> COMPLEX for <i>ctrsm</i> DOUBLE COMPLEX for <i>ztrsm</i> Specifies the scalar <i>alpha</i>. When <i>alpha</i> is zero, then <i>a</i> is not referenced and <i>b</i> need not be set before entry.
<i>a</i>	REAL for <i>strsm</i> DOUBLE PRECISION for <i>dtrsm</i> COMPLEX for <i>ctrsm</i> DOUBLE COMPLEX for <i>ztrsm</i> Array, DIMENSION (<i>lda</i>, <i>k</i>), where <i>k</i> is <i>m</i> when <i>side</i> = 'L' or 'l' and is <i>n</i> when <i>side</i> = 'R' or 'r'. Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>k</i> by <i>k</i> upper triangular part of the array <i>a</i> must contain the upper triangular matrix and the strictly lower triangular part of <i>a</i> is not referenced.

Before entry with *uplo* = 'L' or 'l', the leading *k* by *k* lower triangular part of the array *a* must contain the lower triangular matrix and the strictly upper triangular part of *a* is not referenced.

When *diag* = 'U' or 'u', the diagonal elements of *a* are not referenced either, but are assumed to be unity.

lda INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. When *side* = 'L' or 'l', then *lda* must be at least $\max(1, m)$, when *side* = 'R' or 'r', then *lda* must be at least $\max(1, n)$.

b REAL for *strsm*
DOUBLE PRECISION for *dtrsm*
COMPLEX for *ctrsm*
DOUBLE COMPLEX for *ztrsm*
Array, DIMENSION (*ldb*, *n*). Before entry, the leading *m*-by-*n* part of the array *b* must contain the right-hand side matrix *B*.

ldb INTEGER. Specifies the first dimension of *b* as declared in the calling (sub)program. The value of *ldb* must be at least $\max(1, +m)$.

Output Parameters

b Overwritten by the solution matrix *X*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *trsm* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>k</i> , <i>k</i>) where <i>k</i> = <i>m</i> if <i>side</i> = 'L', <i>k</i> = <i>n</i> otherwise.
<i>b</i>	Holds the matrix <i>B</i> of size (<i>m</i> , <i>n</i>).
<i>side</i>	Must be 'L' or 'R'. The default value is 'L'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>transa</i>	Must be 'N', 'C', or 'T'.

	The default value is 'N'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.
<i>alpha</i>	The default value is 1.

Sparse BLAS Level 1 Routines

This section describes Sparse BLAS Level 1, an extension of BLAS Level 1 included in the Intel® Math Kernel Library beginning with the Intel MKL release 2.1. Sparse BLAS Level 1 is a group of routines and functions that perform a number of common vector operations on sparse vectors stored in compressed form.

Sparse vectors are those in which the majority of elements are zeros. Sparse BLAS routines and functions are specially implemented to take advantage of vector sparsity. This allows you to achieve large savings in computer time and memory. If nz is the number of non-zero vector elements, the computer time taken by Sparse BLAS operations will be $O(nz)$.

Vector Arguments

Compressed sparse vectors. Let a be a vector stored in an array, and assume that the only non-zero elements of a are the following:

$a(k_1), a(k_2), a(k_3) \dots a(k_{nz}),$

where nz is the total number of non-zero elements in a .

In Sparse BLAS, this vector can be represented in compressed form by two FORTRAN arrays, x (values) and $indx$ (indices). Each array has nz elements:

$x(1)=a(k_1), x(2)=a(k_2), \dots x(nz)=a(k_{nz}),$

$indx(1)=k_1, indx(2)=k_2, \dots indx(nz)=k_{nz}.$

Thus, a sparse vector is fully determined by the triple $(nz, x, indx)$. If you pass a negative or zero value of nz to Sparse BLAS, the subroutines do not modify any arrays or variables.

Full-storage vectors. Sparse BLAS routines can also use a vector argument fully stored in a single FORTRAN array (a full-storage vector). If y is a full-storage vector, its elements must be stored contiguously: the first element in $y(1)$, the second in $y(2)$, and so on. This corresponds to an increment $incy = 1$ in BLAS Level 1. No increment value for full-storage vectors is passed as an argument to Sparse BLAS routines or functions.

Naming Conventions

Similar to BLAS, the names of Sparse BLAS subprograms have prefixes that determine the data type involved: *s* and *d* for single- and double-precision real; *c* and *z* for single- and double-precision complex respectively.

If a Sparse BLAS routine is an extension of a “dense” one, the subprogram name is formed by appending the suffix *i* (standing for *indexed*) to the name of the corresponding “dense” subprogram. For example, the Sparse BLAS routine `saxpyi` corresponds to the BLAS routine `saxpy`, and the Sparse BLAS function `cdotci` corresponds to the BLAS function `cdotc`.

Routines and Data Types

Routines and data types supported in the Intel MKL implementation of Sparse BLAS are listed in Table 2-4 .

Table 2-4 Sparse BLAS Routines and Their Data Types

Routine/Function	Data Types	Description
?axpyi	s, d, c, z	Scalar-vector product plus vector (routines)
?doti	s, d	Dot product (functions)
?dotci	c, z	Complex dot product conjugated (functions)
?dotui	c, z	Complex dot product unconjugated (functions)
?gthr	s, d, c, z	Gathering a full-storage sparse vector into compressed form <i>nz</i> , <i>x</i> , <i>indx</i> (routines)
?gthrz	s, d, c, z	Gathering a full-storage sparse vector into compressed form and assigning zeros to gathered elements in the full-storage vector (routines)
?roti	s, d	Givens rotation (routines)
?sctr	s, d, c, z	Scattering a vector from compressed form to full-storage form (routines)

BLAS Level 1 Routines That Can Work With Sparse Vectors

The following BLAS Level 1 routines will give correct results when you pass to them a compressed-form array x (with the increment $incx=1$):

<code>?asum</code>	sum of absolute values of vector elements
<code>?copy</code>	copying a vector
<code>?nrm2</code>	Euclidean norm of a vector
<code>?scal</code>	scaling a vector
<code>i?amax</code>	index of the element with the largest absolute value for real flavors, or the largest sum $ Re(x(i)) + Im(x(i)) $ for complex flavors.
<code>i?amin</code>	index of the element with the smallest absolute value for real flavors, or the smallest sum $ Re(x(i)) + Im(x(i)) $ for complex flavors.

The result i returned by `i?amax` and `i?amin` should be interpreted as index in the compressed-form array, so that the largest (smallest) value is $x(i)$; the corresponding index in full-storage array is $indx(i)$.

You can also call `?roti` to compute the parameters of Givens rotation and then pass these parameters to the Sparse BLAS routines `?roti`.

`?axpyi`

Adds a scalar multiple of compressed sparse vector to a full-storage vector.

Syntax

FORTRAN 77:

```
call saxpyi(nz, a, x, indx, y)
call daxpyi(nz, a, x, indx, y)
call caxpyi(nz, a, x, indx, y)
call zaxpyi(nz, a, x, indx, y)
```

Fortran 95:

```
call axpyi(x, indx, y [, a])
```

Description

The ?axpyi routines perform a vector-vector operation defined as

$$y := a * x + y$$

where:

a is a scalar,

x is a sparse vector stored in compressed form,

y is a vector in full storage form.

The ?axpyi routines reference or modify only the elements of y whose indices are listed in the array $indx$.

The values in $indx$ must be distinct.

Input Parameters

nz	INTEGER. The number of elements in x and $indx$.
a	REAL for saxpyi DOUBLE PRECISION for daxpyi COMPLEX for caxpyi DOUBLE COMPLEX for zaxpyi Specifies the scalar a .
x	REAL for saxpyi DOUBLE PRECISION for daxpyi COMPLEX for caxpyi DOUBLE COMPLEX for zaxpyi Array, DIMENSION at least nz .
$indx$	INTEGER. Specifies the indices for the elements of x . Array, DIMENSION at least nz .
y	REAL for saxpyi DOUBLE PRECISION for daxpyi COMPLEX for caxpyi DOUBLE COMPLEX for zaxpyi Array, DIMENSION at least $\max(indx(i))$.

Output Parameters

y	Contains the updated vector y .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `axpyi` interface are the following:

<code>x</code>	Holds the vector of length (<code>nz</code>).
<code>indx</code>	Holds the vector of length (<code>nz</code>).
<code>y</code>	Holds the vector of length (<code>nz</code>).
<code>a</code>	The default value is 1.

?doti

Computes the dot product of a compressed sparse real vector by a full-storage real vector.

Syntax

FORTRAN 77:

```
res = sdoti(nz, x, indx, y )
res = ddoti(nz, x, indx, y )
```

Fortran 95:

```
res = doti(x, indx, y)
```

Description

The `?doti` functions return the dot product of `x` and `y` defined as

$$\text{res} = x(1)*y(\text{indx}(1)) + x(2)*y(\text{indx}(2)) + \dots + x(nz)*y(\text{indx}(nz))$$

where the triple (`nz`, `x`, `indx`) defines a sparse real vector stored in compressed form, and `y` is a real vector in full storage form. The functions reference only the elements of `y` whose indices are listed in the array `indx`. The values in `indx` must be distinct.

Input Parameters

<code>nz</code>	INTEGER. The number of elements in <code>x</code> and <code>indx</code> .
<code>x</code>	REAL for <code>sdoti</code>

`DOUBLE PRECISION` for `ddoti`
 Array, `DIMENSION` at least `nz`.

`indx` `INTEGER`. Specifies the indices for the elements of `x`.
 Array, `DIMENSION` at least `nz`.

`y` `REAL` for `sdoti`
 `DOUBLE PRECISION` for `ddoti`
 Array, `DIMENSION` at least `max(indx(i))`.

Output Parameters

`res` `REAL` for `sdoti`
 `DOUBLE PRECISION` for `ddoti`
 Contains the dot product of `x` and `y`, if `nz` is positive.
 Otherwise, `res` contains 0.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `doti` interface are the following:

`x` Holds the vector of length (`nz`).

`indx` Holds the vector of length (`nz`).

`y` Holds the vector of length (`nz`).

?dotci

Computes the conjugated dot product of a compressed sparse complex vector with a full-storage complex vector.

Syntax

FORTRAN 77:

```
res = cdotci(nz, x, indx, y )
res = zdotci(nz, x, indx, y )
```

Fortran 95:

```
res = dotci(x, indx, y)
```

Description

The `?dotci` functions return the dot product of x and y defined as

$$\text{conjg}(x(1)) * y(\text{indx}(1)) + \dots + \text{conjg}(x(nz)) * y(\text{indx}(nz))$$

where the triple (nz, x, indx) defines a sparse complex vector stored in compressed form, and y is a real vector in full storage form. The functions reference only the elements of y whose indices are listed in the array indx . The values in indx must be distinct.

Input Parameters

nz	INTEGER. The number of elements in x and indx .
x	COMPLEX for <code>cdotci</code> DOUBLE COMPLEX for <code>zdotci</code> Array, DIMENSION at least nz .
indx	INTEGER. Specifies the indices for the elements of x . Array, DIMENSION at least nz .
y	COMPLEX for <code>cdotci</code> DOUBLE COMPLEX for <code>zdotci</code> Array, DIMENSION at least $\max(\text{indx}(i))$.

Output Parameters

res	COMPLEX for <code>cdotci</code> DOUBLE COMPLEX for <code>zdotci</code> Contains the conjugated dot product of x and y , if nz is positive. Otherwise, res contains 0.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `dotci` interface are the following:

x	Holds the vector of length (nz) .
indx	Holds the vector of length (nz) .

y Holds the vector of length (nz).

?dotui

Computes the dot product of a compressed sparse complex vector by a full-storage complex vector.

Syntax

FORTRAN 77:

```
res = cdotui(nz, x, indx, y )
```

```
res = zdotui(nz, x, indx, y )
```

Fortran 95:

```
res = dotui(x, indx, y)
```

Description

The ?dotui functions return the dot product of x and y defined as

$$res = x(1)*y(indx(1)) + x(2)*y(indx(2)) + \dots + x(nz)*y(indx(nz))$$

where the triple ($nz, x, indx$) defines a sparse complex vector stored in compressed form, and y is a real vector in full storage form. The functions reference only the elements of y whose indices are listed in the array $indx$. The values in $indx$ must be distinct.

Input Parameters

nz	INTEGER. The number of elements in x and $indx$.
x	COMPLEX for cdotui DOUBLE COMPLEX for zdotui Array, DIMENSION at least nz .
$indx$	INTEGER. Specifies the indices for the elements of x . Array, DIMENSION at least nz .
y	COMPLEX for cdotui DOUBLE COMPLEX for zdotui Array, DIMENSION at least $\max(indx(i))$.

Output Parameters

<code>res</code>	COMPLEX for <code>cdotui</code> DOUBLE COMPLEX for <code>zdotui</code> Contains the dot product of x and y , if nz is positive. Otherwise, <code>res</code> contains 0.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `dotui` interface are the following:

x	Holds the vector of length (nz).
<code>indx</code>	Holds the vector of length (nz).
y	Holds the vector of length (nz).

?gthr

Gathers a full-storage sparse vector's elements into compressed form.

Syntax

FORTRAN 77:

```
call sgthr(nz, y, x, indx )
call dgthr(nz, y, x, indx )
call cgthr(nz, y, x, indx )
call zgthr(nz, y, x, indx )
```

Fortran 95:

```
res = gthr(x, indx, y)
```

Description

The `?gthr` routines gather the specified elements of a full-storage sparse vector y into compressed form($nz, x, indx$). The routines reference only the elements of y whose indices are listed in the array `indx`:

$x(i) = y(indx(i))$, for $i=1, 2, \dots, +nz$.

Input Parameters

nz INTEGER. The number of elements of *y* to be gathered.

indx INTEGER. Specifies indices of elements to be gathered.
Array, DIMENSION at least *nz*.

y REAL for sgthr
DOUBLE PRECISION for dgthr
COMPLEX for cgthr
DOUBLE COMPLEX for zgthr
Array, DIMENSION at least $\max(indx(i))$.

Output Parameters

x REAL for sgthr
DOUBLE PRECISION for dgthr
COMPLEX for cgthr
DOUBLE COMPLEX for zgthr
Array, DIMENSION at least *nz*.
Contains the vector converted to the compressed form.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *gthr* interface are the following:

x Holds the vector of length (*nz*).

indx Holds the vector of length (*nz*).

y Holds the vector of length (*nz*).

?gthrz

Gathers a sparse vector's elements into compressed form, replacing them by zeros.

Syntax

FORTRAN 77:

```
call sgthrz(nz, y, x, indx )
call dgthrz(nz, y, x, indx )
call cgthrz(nz, y, x, indx )
call zgthrz(nz, y, x, indx )
```

Fortran 95:

```
res = gthrz(x, indx, y)
```

Description

The ?gthrz routines gather the elements with indices specified by the array *indx* from a full-storage vector *y* into compressed form (*nz*, *x*, *indx*) and overwrite the gathered elements of *y* by zeros. Other elements of *y* are not referenced or modified (see also ?gthr).

Input Parameters

<i>nz</i>	INTEGER. The number of elements of <i>y</i> to be gathered.
<i>indx</i>	INTEGER. Specifies indices of elements to be gathered. Array, DIMENSION at least <i>nz</i> .
<i>y</i>	REAL for sgthrz DOUBLE PRECISION for dgthrz COMPLEX for cgthrz DOUBLE COMPLEX for zgthrz Array, DIMENSION at least $\max(indx(i))$.

Output Parameters

<i>x</i>	REAL for sgthrz DOUBLE PRECISION for dgthrz COMPLEX for cgthrz DOUBLE COMPLEX for zgthrz

	Array, <code>DIMENSION</code> at least <code>nz</code> .
	Contains the vector converted to the compressed form.
<code>y</code>	The updated vector <code>y</code> .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gthrz` interface are the following:

<code>x</code>	Holds the vector of length (<code>nz</code>).
<code>indx</code>	Holds the vector of length (<code>nz</code>).
<code>y</code>	Holds the vector of length (<code>nz</code>).

?roti

Applies Givens rotation to sparse vectors one of which is in compressed form.

Syntax

FORTRAN 77:

```
call sroti(nz, x, indx, y, c, s)
call droti(nz, x, indx, y, c, s)
```

Fortran 95:

```
call roti(x, indx, y [, c] [,s])
```

Description

The `?roti` routines apply the Givens rotation to elements of two real vectors, `x` (in compressed form `nz, x, indx`) and `y` (in full storage form):

$$x(i) = c*x(i) + s*y(indx(i))$$

$$y(indx(i)) = c*y(indx(i)) - s*x(i)$$

The routines reference only the elements of `y` whose indices are listed in the array `indx`. The values in `indx` must be distinct.

Input Parameters

<i>nz</i>	INTEGER. The number of elements in <i>x</i> and <i>indx</i> .
<i>x</i>	REAL for <code>sroti</code> DOUBLE PRECISION for <code>droti</code> Array, DIMENSION at least <i>nz</i> .
<i>indx</i>	INTEGER. Specifies the indices for the elements of <i>x</i> . Array, DIMENSION at least <i>nz</i> .
<i>y</i>	REAL for <code>sroti</code> DOUBLE PRECISION for <code>droti</code> Array, DIMENSION at least $\max(\text{indx}(i))$.
<i>c</i>	A scalar: REAL for <code>sroti</code> DOUBLE PRECISION for <code>droti</code> .
<i>s</i>	A scalar: REAL for <code>sroti</code> DOUBLE PRECISION for <code>droti</code> .

Output Parameters

<i>x</i> and <i>y</i>	The updated arrays.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `roti` interface are the following:

<i>x</i>	Holds the vector of length (<i>nz</i>).
<i>indx</i>	Holds the vector of length (<i>nz</i>).
<i>y</i>	Holds the vector of length (<i>nz</i>).
<i>c</i>	The default value is 1.
<i>s</i>	The default value is 1.

?sctr

Converts compressed sparse vectors into full storage form.

Syntax

FORTRAN 77:

```
call ssctr(nz, x, indx, y )
call dsctr(nz, x, indx, y )
call csctr(nz, x, indx, y )
call zsctr(nz, x, indx, y )
```

Fortran 95:

```
call sctr(x, indx, y)
```

Description

The ?sctr routines scatter the elements of the compressed sparse vector ($nz, x, indx$) to a full-storage vector y . The routines modify only the elements of y whose indices are listed in the array $indx$:

$y(indx(i)) = x(i), \text{ for } i=1, 2, \dots, +nz.$

Input Parameters

nz	INTEGER. The number of elements of x to be scattered.
$indx$	INTEGER. Specifies indices of elements to be scattered. Array, DIMENSION at least nz .
x	REAL for ssctr DOUBLE PRECISION for dsctr COMPLEX for csctr DOUBLE COMPLEX for zsctr Array, DIMENSION at least nz . Contains the vector to be converted to full-storage form.

Output Parameters

y	REAL for ssctr

DOUBLE PRECISION for dsctr
 COMPLEX for csctr
 DOUBLE COMPLEX for zsctr
 Array, DIMENSION at least $\max(\text{indx}(i))$.
 Contains the vector y with updated elements.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sctr` interface are the following:

x	Holds the vector of length (nz) .
$indx$	Holds the vector of length (nz) .
y	Holds the vector of length (nz) .

Sparse BLAS Level 2 and Level 3 Routines

This section describes Sparse BLAS Level 2 and Level 3 routines included in the Intel® Math Kernel Library (Intel® MKL) . Sparse BLAS Level 2 is a group of routines and functions that perform operations between a sparse matrix and dense vectors. Sparse BLAS Level 3 is a group of routines and functions that perform operations between a sparse matrix and dense matrices.

The terms and concepts required to understand the use of the Intel MKL Sparse BLAS Level 2 and Level 3 routines are discussed in the [Linear Solvers Basics](#) appendix.

This Sparse BLAS routines can be useful to implement iterative methods for solving large sparse systems of equations or eigenvalue problems. For example, these routines can be considered as building blocks for “[Iterative Sparse Solvers based on Reverse Communication Interface \(RCI ISS\)](#)” described in the Chapter 8 of the manual.

Intel MKL provides Sparse BLAS Level 2 and Level 3 routines with typical (or conventional) interface similar to the interface used in the NIST* Sparse BLAS library [[Rem05](#)].

Some software packages and libraries (the [PARDISO* Solver](#) used in Intel MKL, *Sparskit 2* [[Saad94](#)], the Compaq* Extended Math Library (CXML)[[CXML01](#)]) use different (early) variation of the compressed sparse row (CSR) format and support only Level 2 operations with simplified interfaces. Intel MKL provides an additional set of Sparse BLAS Level 2 routines with similar simplified interfaces. Each of these routines operates only on a matrix of the fixed type.

The routines described in this section support both one-based indexing and zero-based indexing of the input data (see details in the section [One-based and Zero-based Indexing](#)).

Naming Conventions in Sparse BLAS Level 2 and Level 3

Each Sparse BLAS Level 2 and Level 3 routine has a six- or eight-character base name preceded by the prefix `mkl_` or `mkl_cspblas_`.

The routines with typical (conventional) interface have six-character base names in accordance with the template:

```
mkl_<character> <data> <operation>( )
```

The routines with simplified interfaces have eight-character base names in accordance with the templates:

```
mkl_<character> <data> <mtype> <operation>( )
```

for routines with one-based indexing; and

```
mkl_cspblas_<character> <data> <mtype> <operation>( )
```

for routines with zero-based indexing.

The `<character>` field indicates the data type:

s	real, single precision
c	complex, single precision
d	real, double precision
z	complex, double precision

The `<data>` field indicates the sparse matrix storage format (see section [Sparse Matrix Storage Formats](#)):

coo	coordinate format
csr	compressed sparse row format and its variations
csc	compressed sparse column format and its variations
dia	diagonal format
sky	skyline storage format
bsr	block sparse row format and its variations

The `<operation>` field indicates the type of operation:

mv	matrix-vector product (Level 2)
mm	matrix-matrix product (Level 3)
sv	solving a single triangular system (Level 2)
sm	solving triangular systems with multiple right-hand sides (Level 3)

The field `<mtype>` indicates the matrix type:

<code>ge</code>	sparse representation of a general matrix
<code>sy</code>	sparse representation of the upper or lower triangle of a symmetric matrix
<code>tr</code>	sparse representation of a triangular matrix

Sparse Matrix Storage Formats

The current version of Intel MKL Sparse BLAS Level 2 and Level 3 routines support the following point entry [Duff86] storage formats for sparse matrices:

- *compressed sparse row* format (CSR) and its variations;
- *compressed sparse column* format (CSC);
- *coordinate* format;
- *diagonal* format;
- *skyline* storage format;

and one block entry storage format:

- *block sparse row* format (BSR) and its variations.

For more information see "[Sparse Matrix Storage Formats](#)" in the Appendix A ".

Routines and Supported Operations

This section describes operations supported by the Intel Sparse BLAS Level 2 and Level 3 routines. The following notations are used here:

A is a sparse matrix;
 B and C are dense matrices;
 D is a diagonal scaling matrix;
 x and y are dense vectors;
 α and β are scalars;

$\text{op}(A)$ is one of the possible operations:

$\text{op}(A) = A$;
 $\text{op}(A) = A^T$ - transpose of A ;

$\text{op}(A) = \text{conj}(A')$ - conjugated transpose of A .

$\text{inv}(\text{op}(A))$ denotes the inverse of $\text{op}(A)$.

Intel MKL Sparse BLAS Level 2 and Level 3 routines support the following operations:

- computing the vector product between a sparse matrix and a dense vector:

$$y := \alpha * \text{op}(A) * x + \beta * y$$

- solving a single triangular system:

$$y := \alpha * \text{inv}(\text{op}(A)) * x$$

- computing a product between sparse matrix and dense matrix:

$$C := \alpha * \text{op}(A) * B + \beta * C$$

- solving a sparse triangular system with multiple right-hand sides:

$$C := \alpha * \text{inv}(\text{op}(A)) * B$$

Intel MKL provides an additional set of Sparse BLAS Level 2 routines with *simplified interfaces*. Each of these routines operates on a matrix of the fixed type. The following operations are supported:

- computing the vector product between a sparse matrix and a dense vector (for general and symmetric matrices):

$$y := \text{op}(A) * x$$

- solving a single triangular system (for triangular matrices):

$$y := \text{inv}(\text{op}(A)) * x$$

Matrix type is indicated by the field `<mtype>` in the routine name (see section [Naming Conventions in Sparse BLAS Level 2 and Level 3](#)).



NOTE. The routines with simplified interfaces support only four sparse matrix storage formats, specifically:

CSR format in the 3-array variation accepted in the direct sparse solvers and in the CXML;

diagonal format accepted in the CXML;

coordinate format;

BSR format in the 3-array variation.

Note that routines with both typical (conventional) and simplified interfaces use the same computational kernels that work with certain internal data structures.

Complete list of all routines is given in the [Table 2-9](#) .

Interface Consideration

One-Based and Zero-Based Indexing

The Intel MKL Sparse BLAS Leve2 and Level 3 routines support one-based and zero-based indexing of data arrays.

Routines with typical interfaces support zero-based indexing for the following sparse data storage formats: CSR, CSC, BSR, and COO. Routines with simplified interfaces support zero based indexing for the following sparse data storage formats: CSR, BSR, and COO. See the complete list of [Sparse BLAS Level 2 and Level 3 Routines](#) .

The one-based indexing uses the convention of starting array indices at 1. The zero-based indexing uses the convention of starting array indices at 0. For example, indices of the 5-element array *x* can be presented in case of one-based indexing as follows:

```
Element index:      1      2      3      4      5
```

```
Element value:      1.0   5.0   7.0   8.0   9.0
```

and in case of zero-based indexing as follows:

```
Element index:      0      1      2      3      4
```

```
Element value:      1.0   5.0   7.0   8.0   9.0
```

The detailed descriptions of the one-based and zero-based variants of the sparse data storage formats are given in the “[Sparse Matrix Storage Formats](#)” in Appendix A

Most parameters of the routines are identical for both one-based and zero-based indexing, but some of them have certain differences. The following table lists all these differences.

Parameter	One-based Indexing	Zero-based Indexing
<i>val</i>	Array containing non-zero elements of the matrix <i>A</i> , its length is <i>pntre</i> (<i>m</i>) – <i>pntrb</i> (1).	Array containing non-zero elements of the matrix <i>A</i> , its length is <i>pntre</i> (<i>m</i> –1) – <i>pntrb</i> (0).

<i>pntrb</i>	Array of length m . This array contains row indices, such that $pntrb(I) - pntrb(1)+1$ is the first index of row I in the arrays <i>val</i> and <i>indx</i>	Array of length m . This array contains row indices, such that $pntrb(I) - pntrb(0)$ is the first index of row I in the arrays <i>val</i> and <i>indx</i> .
<i>pntre</i>	Array of length m . This array contains row indices, such that $pntre(I) - pntrb(1)$ is the last index of row I in the arrays <i>val</i> and <i>indx</i> .	Array of length m . This array contains row indices, such that $pntre(I) - pntrb(0)-1$ is the last index of row I in the arrays <i>val</i> and <i>indx</i> .
<i>ia</i>	Array of length $m + 1$, containing indices of elements in the array <i>a</i> , such that $ia(I)$ is the index in the array <i>a</i> of the first non-zero element from the row I . The value of the last element $ia(m + 1)$ is equal to the number of non-zeros plus one.	Array of length $m+1$, containing indices of elements in the array <i>a</i> , such that $ia(I)$ is the index in the array <i>a</i> of the first non-zero element from the row I . The value of the last element $ia(m)$ is equal to the number of non-zeros.
<i>ldb</i>	Specifies the first dimension of <i>b</i> as declared in the calling (sub)program.	Specifies the second dimension of <i>b</i> as declared in the calling (sub)program.
<i>ldc</i>	Specifies the first dimension of <i>c</i> as declared in the calling (sub)program.	Specifies the second dimension of <i>c</i> as declared in the calling (sub)program.

Difference Between Fortran and C Interfaces

Intel MKL provides both Fortran and C interfaces to all Sparse BLAS Leve2 and Level 3 routines. Parameter descriptions are common for both interfaces with the exception of data types that refer to the FORTRAN 77 standard types. Correspondence between data types specific to the Fortran and C interfaces are given below:

Fortran	C
REAL*4	float
REAL*8	double
INTEGER*4	int
INTEGER*8	long long int
CHARACTER	char

For routines with C interfaces all parameters (including scalars) must be passed by references. Another difference is how two-dimensional arrays are represented. In Fortran the column-major order is used, and in C - row-major order. This changes the meaning of the parameters *ldb* and *ldc* (see the table above).

Differences Between Intel MKL and NIST* Interfaces

The Intel MKL Sparse BLAS Level 3 routines have the following conventional interfaces:

`mkl_xyyyymm(transa, m, n, k, alpha, matdescra, arg(A), b, ldb, beta, c, ldc)`, for matrix-matrix product;

`mkl_xyyysm(transa, m, n, alpha, matdescra, arg(A), b, ldb, c, ldc)`, for triangular solvers with multiple right-hand sides.

Here *x* denotes data type, and *yyy* - sparse matrix data structure (storage format).

The analogous NIST* Sparse BLAS (NSB) library routines have the following interfaces:

`xyyyymm(transa, m, n, k, alpha, descra, arg(A), b, ldb, beta, c, ldc, work, lwork)`, for matrix-matrix product;

`xyyyysm(transa, m, n, unitd, dv, alpha, descra, arg(A), b, ldb, beta, c, ldc, work, lwork)`, for triangular solvers with multiple right-hand sides.

Some similar arguments are used in both libraries. The argument *transa* indicates what operation is performed and is slightly different in the NSB library (see [Table 2-5](#)). The arguments *m* and *k* are the number of rows and column in the matrix *A*, respectively, *n* is the number of columns in the matrix *C*. The arguments *alpha* and *beta* are scalar *alpha* and *beta* respectively (*beta* is not used in the Intel MKL triangular solvers.) The arguments *b* and *c* are rectangular arrays with the first dimension *ldb* and *ldc*, respectively. *arg(A)* denotes the list of arguments that describe the sparse representation of *A*.

Table 2-5 Parameter *transa*

	MKL interface	NSB interface	Operation
data type	CHARACTER*1	INTEGER	
value	N or n	0	$op(A) = A$
	T or t	1	$op(A) = A'$
	C or c	2	$op(A) = A'$

Parameter matdescra

The parameter *matdescra* describes the relevant characteristic of the matrix *A*. This manual describes *matdescra* as an array of six elements in line with the NIST* implementation. However, only the first four elements of the array are used in the current versions of the Intel MKL Sparse BLAS routines. Elements *matdescra* (5) and *matdescra*(6) are reserved for future use. Note that whether *matdescra* is described in the user's program as an array of length 6 or 4 is of no importance because the array is declared as a pointer in the Intel MKL routines. To learn more about declaration of the *matdescra* array see Sparse BLAS examples located in the following subdirectory of the Intel MKL installation directory: *examples/spblas/*. The table below list elements of the parameter *matdescra*, their values and meanings. The parameter *matdescra* corresponds to the argument *descra* from NSB library.

Table 2-6 Possible Values of the Parameter *matdescra* (*descra*)

MKL interface		NSB interfaceMatrix characteristics		
	one-based indexing	zero-based indexing		
data type	CHARACTER	Char	INTEGER	
1st element	<i>matdescra</i> (1)	<i>matdescra</i> (0)	<i>descra</i> (1)	matrix structure
value	G	G	0	general
	S	S	1	symmetric ($A = A'$)
	H	H	2	Hermitian ($A = \text{conjg}(A')$)
	T	T	3	triangular
	A	A	4	skew(anti)-symmetric ($A = -A'$)
	D	D	5	diagonal
2nd element	<i>matdescra</i> (2)	<i>matdescra</i> (1)	<i>descra</i> (2)	upper/lower triangular indicator
value	L	L	1	lower
	U	U	2	upper
3rd element	<i>matdescra</i> (3)	<i>matdescra</i> (2)	<i>descra</i> (3)	main diagonal type

	MKL interface		NSB interface	Matrix characteristics
value	N	N	0	non-unit
	U	U	1	unit
4th element	<i>matdescra</i> (4) <i>matdescra</i> (3)			type of indexing
value	F			one-based indexing
		C		zero-based indexing

In some cases possible element values of the parameter *matdescra* depend on the values of other elements. The [Table 2-7 "Possible Combinations of Element Values of the Parameter *matdescra*"](#) lists all possible combinations of element values for both multiplication routines and triangular solvers.

Table 2-7 Possible Combinations of Element Values of the Parameter *matdescra*

Routines	<i>matdescra</i> (1)	<i>matdescra</i> (2)	<i>matdescra</i> (3)	<i>matdescra</i> (4)
Multiplication Routines	G	ignored	ignored	F (default) or C
	S or H	L (default)	N (default)	F (default) or C
	S or H	L (default)	U	F (default) or C
	S or H	U	N (default)	F (default) or C
	S or H	U	U	F (default) or C
	A	L (default)	ignored	F (default) or C
	A	U	ignored	F (default) or C
Multiplication Routines and Triangular Solvers	T	L	U	F (default) or C
	T	L	N	F (default) or C
	T	U	U	F (default) or C
	T	U	N	F (default) or C
	D	ignored	N (default)	F (default) or C
	D	ignored	U	F (default) or C

For a matrix in the skyline format with main diagonal declared to be unit, diagonal elements must be stored in the sparse representation even if they are zero. In all other formats diagonal elements can be stored (if needed) in the sparse representation if they are not zero.

Operations with Partial Matrices

One of the distinctive feature of the Intel MKL Sparse BLAS routines is a possibility to perform operations only on partial matrices composed of certain parts (triangles and main diagonal) of the input sparse matrix. It can be done by setting properly first three elements of the parameter *matdescra*.

An arbitrary sparse matrix A can be decomposed as

$$A = L + D + U$$

where L is the strict lower triangle of A , U is the strict upper triangle of A , D is the main diagonal.

Table 2-8 shows correspondence between the output matrices and values of the parameter *matdescra* for the sparse matrix A for multiplication routines.

Table 2-8 Output Matrices for Multiplication Routines

matdescra(1)	matdescra(2)	matdescra(3)	Output Matrix
G	ignored	ignored	$\alpha * \text{op}(A) * x + \beta * y$ $\alpha * \text{op}(A) * B + \beta * C$
S or H	L	N	$\alpha * \text{op}(L+D+L') * x + \beta * y$ $\alpha * \text{op}(L+D+L') * B + \beta * C$
S or H	L	U	$\alpha * \text{op}(L+I+L') * x + \beta * y$ $\alpha * \text{op}(L+I+L') * B + \beta * C$
S or H	U	N	$\alpha * \text{op}(U'+D+U) * x + \beta * y$ $\alpha * \text{op}(U'+D+U) * B + \beta * C$
S or H	U	U	$\alpha * \text{op}(U'+I+U) * x + \beta * y$ $\alpha * \text{op}(U'+I+U) * B + \beta * C$
T	L	U	$\alpha * \text{op}(L+I) * x + \beta * y$ $\alpha * \text{op}(L+I) * B + \beta * C$

matdescra(1)	matdescra(2)	matdescra(3)	Output Matrix
T	L	N	$\alpha * \text{op}(L+D) * x + \beta * y$ $\alpha * \text{op}(L+D) * B + \beta * C$
T	U	U	$\alpha * \text{op}(U+I) * x + \beta * y$ $\alpha * \text{op}(U+I) * B + \beta * C$
T	U	N	$\alpha * \text{op}(U+D) * x + \beta * y$ $\alpha * \text{op}(U+D) * B + \beta * C$
A	L	ignored	$\alpha * \text{op}(L-L') * x + \beta * y$ $\alpha * \text{op}(L-L') * B + \beta * C$
A	U	ignored	$\alpha * \text{op}(U-U') * x + \beta * y$ $\alpha * \text{op}(U-U') * B + \beta * C$
D	ignored	N	$\alpha * D * x + \beta * y$ $\alpha * D * B + \beta * C$
D	ignored	U	$\alpha * x + \beta * y$ $\alpha * B + \beta * C$

Table 2-9 shows correspondence between the output matrices and values of the parameter *matdescra* for the sparse matrix *A* for triangular solvers.

Table 2-9 Output Matrices for Triangular Solvers

matdescra(1)	matdescra(2)	matdescra(3)	Output Matrix
T	L	N	$\alpha * \text{inv}(\text{op}(L+L)) * x$ $\alpha * \text{inv}(\text{op}(L+L)) * B$
T	L	U	$\alpha * \text{inv}(\text{op}(L+L)) * x$ $\alpha * \text{inv}(\text{op}(L+L)) * B$
T	U	N	$\alpha * \text{inv}(\text{op}(U+U)) * x$ $\alpha * \text{inv}(\text{op}(U+U)) * B$
T	U	U	$\alpha * \text{inv}(\text{op}(U+U)) * x$

matdescra(1)	matdescra(2)	matdescra(3)	Output Matrix
			$\alpha * \text{inv}(\text{op}(U+U)) * B$
D	ignored	N	$\alpha * \text{inv}(D) * x$
			$\alpha * \text{inv}(D) * B$
D	ignored	U	$\alpha * x$
			$\alpha * B$

Sparse BLAS Level 2 and Level 3 Routines.

Table 2-9 lists the sparse BLAS Level 2 and Level 3 routines described in more detail later in this section.

Table 2-9 Sparse BLAS Level 2 and Level 3 Routines

Routine/Function	Description

Simplified interface, one-based indexing

<code>mkl_?csrghemv</code>	Computes matrix - vector product of a sparse general matrix in the CSR format (3-array variation)
<code>mkl_dbsrghemv</code>	Computes matrix - vector product of a sparse general matrix in the BSR format (3-array variation).
<code>mkl_dcoogemv</code>	Computes matrix - vector product of a sparse general matrix in the coordinate format.
<code>mkl_ddiagemv</code>	Computes matrix - vector product of a sparse general matrix in the diagonal format.
<code>mkl_dcsrcsymv</code>	Computes matrix - vector product of a sparse symmetrical matrix in the CSR format (3-array variation)
<code>mkl_dbsrcsymv</code>	Computes matrix - vector product of a sparse symmetrical matrix in the BSR format (3-array variation).
<code>mkl_dcoosymv</code>	Computes matrix - vector product of a sparse symmetrical matrix in the coordinate format.

Routine/Function	Description
<code>mkl_ddiasymv</code>	Computes matrix - vector product of a sparse symmetrical matrix in the diagonal format.
<code>mkl_dcsrtrsv</code>	Triangular solvers with simplified interface for a sparse matrix in the CSR format (3-array variation).
<code>mkl_dbsrtrsv</code>	Triangular solver with simplified interface for a sparse matrix in the BSR format (3-array variation).
<code>mkl_dcootrsv</code>	Triangular solvers with simplified interface for a sparse matrix in the coordinate format.
<code>mkl_ddiattrsv</code>	Triangular solvers with simplified interface for a sparse matrix in the diagonal format.

Simplified interface, zero-based indexing

<code>mkl_cspblas_dcsrgemv</code>	Computes matrix - vector product of a sparse general matrix in the CSR format (3-array variation) with zero-based indexing.
<code>mkl_cspblas_dbsrgemv</code>	Computes matrix - vector product of a sparse general matrix in the BSR format (3-array variation) with zero-based indexing.
<code>mkl_cspblas_dcoogemv</code>	Computes matrix - vector product of a sparse general matrix in the coordinate format with zero-based indexing.
<code>mkl_cspblas_dcsrsymv</code>	Computes matrix - vector product of a sparse symmetrical matrix in the CSR format (3-array variation) with zero-based indexing.
<code>mkl_cspblas_dbsrsymv</code>	Computes matrix - vector product of a sparse symmetrical matrix in the BSR format (3-array variation) with zero-based indexing.
<code>mkl_cspblas_dcoosymv</code>	Computes matrix - vector product of a sparse symmetrical matrix in the coordinate format with zero-based indexing.
<code>mkl_cspblas_dcsrtrsv</code>	Triangular solvers with simplified interface for a sparse matrix in the CSR format (3-array variation) with zero-based indexing.
<code>mkl_cspblas_dbsrtrsv</code>	Triangular solver with simplified interface for a sparse matrix in the BSR format (3-array variation) with zero-based indexing.

Routine/Function	Description
<code>mk1_cspblas_dcootrsv</code>	Triangular solver with simplified interface for a sparse matrix in the coordinate format with zero-based indexing.

Typical (conventional) interface, one-based and zero-based indexing

<code>mk1_dcsrcmv</code>	Computes matrix - vector product of a sparse matrix in the CSR format.
<code>mk1_dbsrcmv</code>	Computes matrix - vector product of a sparse matrix in the BSR format.
<code>mk1_dcscmv</code>	Computes matrix - vector product for a sparse matrix in the CSC format.
<code>mk1_dcoomv</code>	Computes matrix - vector product for a sparse matrix in the coordinate format.
<code>mk1_dcsrcsv</code>	Solves a system of linear equations for a sparse matrix in the CSR format.
<code>mk1_dbsrcsv</code>	Solves a system of linear equations for a sparse matrix in the BSR format.
<code>mk1_dcscsv</code>	Solves a system of linear equations for a sparse matrix in the CSC format.
<code>mk1_dcoosv</code>	Solves a system of linear equations for a sparse matrix in the coordinate format.
<code>mk1_dcsrcmm</code>	Computes matrix - matrix product of a sparse matrix in the CSR format
<code>mk1_dbsrcmm</code>	Computes matrix - matrix product of a sparse matrix in the BSR format.
<code>mk1_dcscmm</code>	Computes matrix - matrix product of a sparse matrix in the CSC format
<code>mk1_dcoomm</code>	Computes matrix - matrix product of a sparse matrix in the coordinate format.

Routine/Function	Description
<code>mk1_dcsrsm</code>	Solves a system of linear matrix equations for a sparse matrix in the CSR format.
<code>mk1_dbsrsm</code>	Solves a system of linear matrix equations for a sparse matrix in the BSR format.
<code>mk1_dcscsm</code>	Solves a system of linear matrix equations for a sparse matrix in the CSC format.
<code>mk1_dcoosm</code>	Solves a system of linear matrix equations for a sparse matrix in the coordinate format.

Typical (conventional) interface, one-based indexing

<code>mk1_ddiamv</code>	Computes matrix - vector product of a sparse matrix in the diagonal format.
<code>mk1_dskymv</code>	Computes matrix - vector product for a sparse matrix in the skyline storage format.
<code>mk1_ddiasv</code>	Solves a system of linear equations for a sparse matrix in the diagonal format.
<code>mk1_dskysv</code>	Solves a system of linear equations for a sparse matrix in the skyline format.
<code>mk1_ddiamm</code>	Computes matrix - matrix product of a sparse matrix in the diagonal format.
<code>mk1_dskymm</code>	Computes matrix - matrix product of a sparse matrix in the skyline storage format.
<code>mk1_ddiasm</code>	Solves a system of linear matrix equations for a sparse matrix in the diagonal format.
<code>mk1_dskysm</code>	Solves a system of linear matrix equations for a sparse matrix in the skyline storage format.

mkl_?csrgev

Computes matrix - vector product of a sparse general matrix stored in the CSR format (3-array variation) with one-based indexing.

Syntax

Fortran:

```
call mkl_scsrgev(transa, m, a, ia, ja, x, y)
call mkl_dcsrgev(transa, m, a, ia, ja, x, y)
```

C:

```
mkl_scsrgev(&transa, &m, a, ia, ja, x, y);
mkl_dcsrgev(&transa, &m, a, ia, ja, x, y);
```

Description

The `mkl_?csrgev` routine performs a matrix-vector operation defined as

$y := A * x$

or

$y := A' * x,$

where:

x and y are vectors,

A is an m -by- m sparse square matrix in the CSR format (3-array variation), A' is the transpose of A .



NOTE. This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

transa CHARACTER*1. Specifies the operation.

	If <i>transa</i> = 'N' or 'n', then as $y := A * x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := A' * x$,
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>a</i>	REAL*4 for mkl_scsrgev. REAL*8 for mkl_dcscrgev. Array containing non-zero elements of the matrix <i>A</i>. Its length is equal to the number of non-zero elements in the matrix <i>A</i>. Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.
<i>ia</i>	INTEGER. Array of length $m + 1$, containing indices of elements in the array <i>a</i>, such that <i>ia</i>(<i>i</i>) is the index in the array <i>a</i> of the first non-zero element from the row <i>i</i>. The value of the last element <i>ia</i>($m + 1$) is equal to the number of non-zeros plus one. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.
<i>ja</i>	INTEGER. Array containing the column indices for each non-zero element of the matrix <i>A</i>. Its length is equal to the length of the array <i>a</i>. Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.
<i>x</i>	REAL*4 for mkl_scsrgev. REAL*8 for mkl_dcscrgev. Array, DIMENSION is <i>m</i>. On entry, the array <i>x</i> must contain the vector <i>x</i>.

Output Parameters

<i>y</i>	REAL*4 for mkl_scsrgev. REAL*8 for mkl_dcscrgev. Array, DIMENSION at least <i>m</i>. On exit, the array <i>y</i> must contain the vector <i>y</i>.

Interfaces

FORTRAN 77:

```

SUBROUTINE mkl_scsrgemv(transa, m, a, ia, ja, x, y)
    CHARACTER*1  transa
    INTEGER      m
    INTEGER      ia(*), ja(*)
    REAL*4       a(*), x(*), y(*)
SUBROUTINE mkl_dcsrgemv(transa, m, a, ia, ja, x, y)
    CHARACTER*1  transa
    INTEGER      m
    INTEGER      ia(*), ja(*)
    REAL*8       a(*), x(*), y(*)

```

C:

```

void mkl_scsrgemv(char *transa, int *m, float *a,
    int *ia, int *ja, float *x, float *y);
void mkl_dcsrgemv(char *transa, int *m, double *a,
    int *ia, int *ja, double *x, double *y);

```

mkl_dbsrgemv

Computes matrix - vector product of a sparse general matrix stored in the BSR format (3-array variation) with one-based indexing.

Syntax

Fortran:

```
call mkl_dbsrgemv(transa, m, lb, a, ia, ja, x, y)
```

C:

```
mkl_dbsrgemv(&transa, &m, &lb, a, ia, ja, x, y);
```

Description

The `mkl_dbsrgemv` routine performs a matrix-vector operation defined as

$y := A * x$

or

$y := A' * x,$

where:

x and y are vectors,

A is an m -by- m block sparse square matrix in the BSR format (3-array variation), A' is the transpose of A .



NOTE. This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then the matrix-vector product is computed as $y := A * x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $y := A' * x,$
<i>m</i>	INTEGER. Number of block rows of the matrix <i>A</i> .
<i>lb</i>	INTEGER. Size of the block in the matrix <i>A</i> .
<i>a</i>	REAL*8. Array containing elements of non-zero blocks of the matrix <i>A</i> . Its length is equal to the number of non-zero blocks in the matrix <i>A</i> multiplied by <i>lb*lb</i> . Refer to <i>values</i> array description in BSR Format for more details.
<i>ia</i>	INTEGER. Array of length $(m + 1)$, containing indices of block in the array <i>a</i> , such that <i>ia</i> (<i>i</i>) is the index in the array <i>a</i> of the first non-zero element from the row <i>i</i> . The

value of the last element $ia(m + 1)$ is equal to the number of non-zero blocks plus one. Refer to *rowIndex* array description in for more details.

ja REAL*8. Array containing the column indices for each non-zero block in the matrix *A*.
Its length is equal to the number of non-zero blocks of the matrix *A*. Refer to *columns* array description in [BSR Format](#) for more details.

x REAL*8.
Array, DIMENSION ($m*lb$).
On entry, the array *x* must contain the vector *x*.

Output Parameters

y REAL*8.
Array, DIMENSION at least ($m*lb$).
On exit, the array *y* must contain the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_dbsrgemv(transa, m, lb, a, ia, ja, x, y)
  CHARACTER*1  transa
  INTEGER      m, lb
  INTEGER      ia(*), ja(*)
  REAL*8       a(*), x(*), y(*)
```

C:

```
void mkl_dbsrgemv(char *transa, int *m, int *lb, double *a,
int *ia, int *ja, double *x, double *y);
```

mkl_dcoogemv

Computes matrix - vector product of a sparse general matrix stored in the coordinate format with one-based indexing.

Syntax

Fortran:

```
call mkl_dcoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

C:

```
mkl_dcoogemv(&transa, &m, val, rowind, colind, &nnz, x, y);
```

Description

The `mkl_dcoogemv` routine performs a matrix-vector operation defined as

$y := A * x$

or

$y := A' * x,$

where:

x and y are vectors,

A is an m -by- m sparse square matrix in the coordinate format, A' is the transpose of A .



NOTE. This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then the matrix-vector product is computed as $y := A * x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $y := A' * x,$

<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>val</i>	REAL*8. Array of length <i>nnz</i> , contains non-zero elements of the matrix <i>A</i> in the arbitrary order. Refer to <i>values</i> array description in Coordinate Format for more details.
<i>rowind</i>	INTEGER. Array of length <i>nnz</i> , contains the row indices for each non-zero element of the matrix <i>A</i> . Refer to <i>rows</i> array description in Coordinate Format for more details.
<i>colind</i>	INTEGER. Array of length <i>nnz</i> , contains the column indices for each non-zero element of the matrix <i>A</i> . Refer to <i>columns</i> array description in Coordinate Format for more details.
<i>nnz</i>	INTEGER. Specifies the number of non-zero element of the matrix <i>A</i> . Refer to <i>nnz</i> description in Coordinate Format for more details.
<i>x</i>	REAL*8. Array, DIMENSION is <i>m</i> . One entry, the array <i>x</i> must contain the vector <i>x</i> .

Output Parameters

<i>y</i>	REAL*8. Array, DIMENSION at least <i>m</i> . On exit, the array <i>y</i> must contain the vector <i>y</i> .

Interfaces

FORTRAN 77:

```

SUBROUTINE mkl_dcoogemv(transa, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1  transa
  INTEGER      m, nnz
  INTEGER      rowind(*), colind(*)
  REAL*8      val(*), x(*), y(*)

```

C:

```
void mkl_dcoogemv(char *transa, int *m, double *val, int *rowind,  
int *colind, int *nnz, double *x, double *y);
```

mkl_ddiagemv

Computes matrix - vector product of a sparse general matrix stored in the diagonal format with one-based indexing.

Syntax

Fortran:

```
call mkl_ddiagemv(transa, m, val, lval, idiag, ndiag, x, y)
```

C:

```
mkl_ddiagemv(&transa, &m, val, &lval, idiag, &ndiag, x, y);
```

Description

The `mkl_ddiagemv` routine performs a matrix-vector operation defined as

$y := A * x$

or

$y := A' * x,$

where:

x and y are vectors,

A is an m -by- m sparse square matrix in the diagonal storage format, A' is the transpose of A .



NOTE. This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

transa CHARACTER*1. Specifies the operation.

	If <i>transa</i> = 'N' or 'n', then $y := A*x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := A'*x$,
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>val</i>	REAL*8. Two-dimensional array of size <i>lval</i> by <i>ndiag</i> , contains non-zero diagonals of the matrix <i>A</i> . Refer to <i>values</i> array description in Diagonal Storage Scheme for more details.
<i>lval</i>	INTEGER. Leading dimension of <i>val</i> , $lval \geq m$. Refer to <i>lval</i> description in Diagonal Storage Scheme for more details.
<i>idiag</i>	INTEGER. Array of length <i>ndiag</i> , contains the distances between main diagonal and each non-zero diagonals in the matrix <i>A</i> . Refer to <i>distance</i> array description in Diagonal Storage Scheme for more details.
<i>ndiag</i>	INTEGER. Specifies the number of non-zero diagonals of the matrix <i>A</i> .
<i>x</i>	REAL*8. Array, DIMENSION is <i>m</i> . On entry, the array <i>x</i> must contain the vector <i>x</i> .

Output Parameters

<i>y</i>	REAL*8. Array, DIMENSION at least <i>m</i> . On exit, the array <i>y</i> must contain the vector <i>y</i> .

Interfaces

FORTRAN 77:

```

SUBROUTINE mkl_ddiagemv(transa, m, val, lval, idiag, ndiag, x, y)
  CHARACTER*1  transa
  INTEGER      m, lval, ndiag
  INTEGER      idiag(*)
  REAL*8       val(lval,*), x(*), y(*)

```

C:

```
void mkl_ddiagemv(char *transa, int *m, double *val, int *lval,  
    int *idiag, int *ndiag, double *x, double *y);
```

mkl_dcsrsvmv

Computes matrix - vector product of a sparse symmetrical matrix stored in the CSR format (3-array variation) with one-based indexing.

Syntax

Fortran:

```
call mkl_dcsrsvmv(uplo, m, a, ia, ja, x, y)
```

C:

```
mkl_dcsrsvmv(&uplo, &m, a, ia, ja, x, y);
```

Description

The `mkl_dcsrsvmv` routine performs a matrix-vector operation defined as

$y := A * x$

or

$y := A' * x,$

where:

x and y are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the CSR format (3-array variation), A' is the transpose of A .



NOTE. This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or low triangle of the matrix <i>A</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix <i>A</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix <i>A</i> is used.
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>a</i>	REAL*8. Array containing non-zero elements of the matrix <i>A</i>. Its length is equal to the number of non-zero elements in the matrix <i>A</i>. Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.
<i>ia</i>	INTEGER. Array of length $m + 1$, containing indices of elements in the array <i>a</i>, such that <i>ia</i>(<i>i</i>) is the index in the array <i>a</i> of the first non-zero element from the row <i>i</i>. The value of the last element <i>ia</i>($m + 1$) is equal to the number of non-zeros plus one. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.
<i>ja</i>	REAL*8. Array containing the column indices for each non-zero element of the matrix <i>A</i>. Its length is equal to the length of the array <i>a</i>. Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.
<i>x</i>	REAL*8. Array, DIMENSION is <i>m</i>. On entry, the array <i>x</i> must contain the vector <i>x</i>.

Output Parameters

<i>y</i>	REAL*8. Array, DIMENSION at least <i>m</i>. On exit, the array <i>y</i> must contain the vector <i>y</i>.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_dcsrsymv(uplo, m, a, ia, ja, x, y)

  CHARACTER*1  uplo
  INTEGER      m
  INTEGER      ia(*), ja(*)
  REAL*8       a(*), x(*), y(*)
```

C:

```
void mkl_dcsrsymv(char *uplo, int *m, double *a,
  int *ia, int *ja, double *x, double *y);
```

mkl_dbsrsymv

Computes matrix - vector product of a sparse symmetrical matrix stored in the BSR format (3-array variation) with one-based indexing.

Syntax

Fortran:

```
call mkl_dbsrsymv(uplo, m, lb, a, ia, ja, x, y)
```

C:

```
mkl_dbsrsymv(&uplo, &m, &lb, a, ia, ja, x, y);
```

Description

The `mkl_dbsrsymv` routine performs a matrix-vector operation defined as

$y := A * x$

or

$y := A' * x,$

where:

x and y are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the BSR format (3-array variation), A' is the transpose of A .



NOTE. This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or low triangle of the matrix A is considered. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix A is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix A is used.
<i>m</i>	INTEGER. Number of block rows of the matrix A .
<i>lb</i>	INTEGER. Size of the block in the matrix A .
<i>a</i>	REAL*8. Array containing elements of non-zero blocks of the matrix A . Its length is equal to the number of non-zero blocks in the matrix A multiplied by $lb*lb$. Refer to <i>values</i> array description in BSR Format for more details.
<i>ia</i>	INTEGER. Array of length $(m + 1)$, containing indices of block in the array <i>a</i> , such that $ia(i)$ is the index in the array <i>a</i> of the first non-zero element from the row i . The value of the last element $ia(m + 1)$ is equal to the number of non-zero blocks plus one. Refer to <i>rowIndex</i> array description in BSR Format for more details.
<i>ja</i>	REAL*8. Array containing the column indices for each non-zero block in the matrix A . Its length is equal to the number of non-zero blocks of the matrix A . Refer to <i>columns</i> array description in BSR Format for more details.
<i>x</i>	REAL*8.

Array, DIMENSION $(m*lb)$.
On entry, the array x must contain the vector x .

Output Parameters

y REAL*8.
Array, DIMENSION at least $(m*lb)$.
On exit, the array y must contain the vector y .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_dbsrsymv(uplo, m, lb, a, ia, ja, x, y)
  CHARACTER*1 uplo
  INTEGER      m, lb
  INTEGER      ia(*), ja(*)
  REAL*8       a(*), x(*), y(*)
```

C:

```
void mkl_dbsrsymv(char *uplo, int *m, int *lb,
  double *a, int *ia, int *ja, double *x, double *y);
```

mkl_dcoosymv

Computes matrix - vector product of a sparse symmetrical matrix stored in the coordinate format with one-based indexing.

Syntax

Fortran:

```
call mkl_dcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
```

C:

```
mkl_dcoosymv(&uplo, &m, val, rowind, colind, &nnz, x, y);
```

Description

The `mk1_dcoosymv` routine performs a matrix-vector operation defined as

$$y := A * x$$

or

$$y := A' * x,$$

where:

x and y are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the coordinate format, A' is the transpose of A .



NOTE. This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or low triangle of the matrix A is used. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix A is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix A is used.
<i>m</i>	INTEGER. Number of rows of the matrix A.
<i>val</i>	REAL*8. Array of length <i>nnz</i>, contains non-zero elements of the matrix A in the arbitrary order. Refer to <i>values</i> array description in Coordinate Format for more details.
<i>rowind</i>	INTEGER. Array of length <i>nnz</i>, contains the row indices for each non-zero element of the matrix A. Refer to <i>rows</i> array description in Coordinate Format for more details.

<i>colind</i>	INTEGER. Array of length <i>nnz</i> , contains the column indices for each non-zero element of the matrix <i>A</i> . Refer to <i>columns</i> array description in Coordinate Format for more details.
<i>nnz</i>	INTEGER. Specifies the number of non-zero element of the matrix <i>A</i> . Refer to <i>nnz</i> description in Coordinate Format for more details.
<i>x</i>	REAL*8. Array, DIMENSION is <i>m</i> . On entry, the array <i>x</i> must contain the vector <i>x</i> .

Output Parameters

<i>y</i>	REAL*8. Array, DIMENSION at least <i>m</i> . On exit, the array <i>y</i> must contain the vector <i>y</i> .

Interfaces

FORTRAN 77:

```

SUBROUTINE mkl_dcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1  uplo
  INTEGER      m, nnz
  INTEGER      rowind(*), colind(*)
  REAL*8       val(*), x(*), y(*)

```

C:

```

void mkl_dcoosymv(char *uplo, int *m, double *val, int *rowind,
  int *colind, int *nnz, double *x, double *y);

```

mkl_ddiasymv

Computes matrix - vector product of a sparse symmetrical matrix stored in the diagonal format with one-based indexing.

Syntax

Fortran:

```
call mkl_ddiasymv(uplo, m, val, lval, iddiag, ndiag, x, y)
```

C:

```
mkl_ddiasymv(&uplo, &m, val, &lval, iddiag, &ndiag, x, y);
```

Description

The `mkl_ddiasymv` routine performs a matrix-vector operation defined as

$y := A * x$

or

$y := A' * x,$

where:

x and y are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix, A' is the transpose of A .



NOTE. This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or low triangle of the matrix A is used. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix A is used.

	If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix <i>A</i> is used.
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>val</i>	REAL*8. Two-dimensional array of size <i>lval</i> by <i>ndiag</i> , contains non-zero diagonals of the matrix <i>A</i> . Refer to <i>values</i> array description in Diagonal Storage Scheme for more details.
<i>lval</i>	INTEGER. Leading dimension of <i>val</i> , <i>lval</i> ≥ <i>m</i> . Refer to <i>lval</i> description in Diagonal Storage Scheme for more details.
<i>idiag</i>	INTEGER. Array of length <i>ndiag</i> , contains the distances between main diagonal and each non-zero diagonals in the matrix <i>A</i> . Refer to <i>distance</i> array description in Diagonal Storage Scheme for more details.
<i>ndiag</i>	INTEGER. Specifies the number of non-zero diagonals of the matrix <i>A</i> .
<i>x</i>	REAL*8. Array, DIMENSION is <i>m</i> . On entry, the array <i>x</i> must contain the vector <i>x</i> .

Output Parameters

<i>y</i>	REAL*8. Array, DIMENSION at least <i>m</i> . On exit, the array <i>y</i> must contain the vector <i>y</i> .

Interfaces

FORTRAN 77:

```

SUBROUTINE mkl_ddiasymv(uplo, m, val, lval, idiag, ndiag, x, y)
  CHARACTER*1  uplo
  INTEGER      m, lval, ndiag
  INTEGER      idiag(*)
  REAL*8       val(lval,*), x(*), y(*)

```


C:

```
void mkl_ddiasymv(char *uplo, int *m, double *val, int *lval,  
    int *idiag, int *ndiag, double *x, double *y);
```

mkl_dcsrtrsv

Triangular solvers with simplified interface for a sparse matrix in the CSR format (3-array variation) with one-based indexing.

Syntax

Fortran:

```
call mkl_dcsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
```

C:

```
mkl_dcsrtrsv(&uplo, &transa, &diag, &m, a, ia, ja, x, y);
```

Description

The `mkl_dcsrtrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the CSR format (3 array variation):

$$A*y = x$$

or

$$A'*y = x,$$

where:

x and y are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .



NOTE. This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or low triangle of the matrix <i>A</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix <i>A</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix <i>A</i> is used.
<i>transa</i>	CHARACTER*1. Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then $A*y = x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $A'*y = x$,
<i>diag</i>	CHARACTER*1. Specifies whether <i>A</i> is unit triangular. If <i>diag</i> = 'U' or 'u', then <i>A</i> is a unit triangular. If <i>diag</i> = 'N' or 'n', then <i>A</i> is not unit triangular.
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>a</i>	REAL*8. Array containing non-zero elements of the matrix <i>A</i> . Its length is equal to the number of non-zero elements in the matrix <i>A</i> . Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.
<i>ia</i>	INTEGER. Array of length $m + 1$, containing indices of elements in the array <i>a</i> , such that <i>ia</i> (<i>i</i>) is the index in the array <i>a</i> of the first non-zero element from the row <i>i</i> . The value of the last element <i>ia</i> ($m + 1$) is equal to the number of non-zeros plus one. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.
<i>ja</i>	REAL*8. Array containing the column indices for each non-zero element of the matrix <i>A</i> . Its length is equal to the length of the array <i>a</i> . Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.



NOTE. Column indices must be sorted in increasing order for each row.

x REAL*8.
 Array, DIMENSION is *m*.
 On entry, the array *x* must contain the vector *x*.

Output Parameters

y REAL*8.
 Array, DIMENSION at least *m*.
 Contains the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_dcsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
  CHARACTER*1 uplo, transa, diag
  INTEGER      m
  INTEGER      ia(*), ja(*)
  REAL*8       a(*), x(*), y(*)
```

C:

```
void mkl_dcsrtrsv(char *uplo, char *transa, char *diag, int *m,
  double *a, int *ia, int *ja, double *x, double *y);
```

mkl_dbsrtrsv

Triangular solver with simplified interface for a sparse matrix stored in the BSR format (3-array variation) with one-based indexing.

Syntax

Fortran:

```
call mkl_dbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
```

C:

```
mkl_dbsrtrsv(&uplo, &transa, &diag, &m, &lb, a, ia, ja, x, y);
```

Description

The `mkl_dbsrtrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the BSR format (3-array variation) :

$y := A * x$

or

$y := A' * x,$

where:

x and y are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .



NOTE. This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>uplo</i>	CHARACTER*1. Specifies the upper or low triangle of the matrix A is used. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix A is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix A is used.
<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then the matrix-vector product is computed as $y := A * x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $y := A' * x$.
<i>diag</i>	CHARACTER*1. Specifies whether A is a unit triangular matrix. If <i>diag</i> = 'U' or 'u', then A is a unit triangular. If <i>diag</i> = 'N' or 'n', then A is not a unit triangular.
<i>m</i>	INTEGER. Number of block rows of the matrix A.

<i>lb</i>	INTEGER. Size of the block in the matrix <i>A</i> .
<i>a</i>	REAL*8. Array containing elements of non-zero blocks of the matrix <i>A</i> . Its length is equal to the number of non-zero blocks in the matrix <i>A</i> multiplied by <i>lb*lb</i> . Refer to <i>values</i> array description in BSR Format for more details.
<i>ia</i>	INTEGER. Array of length $(m + 1)$, containing indices of block in the array <i>a</i> , such that <i>ia(i)</i> is the index in the array <i>a</i> of the first non-zero element from the row <i>i</i> . The value of the last element <i>ia(m + 1)</i> is equal to the number of non-zero blocks plus one. Refer to <i>rowIndex</i> array description in BSR Format for more details.
<i>ja</i>	REAL*8. Array containing the column indices for each non-zero block in the matrix <i>A</i> . Its length is equal to the number of non-zero blocks of the matrix <i>A</i> . Refer to <i>columns</i> array description in BSR Format for more details.
<i>x</i>	REAL*8. Array, DIMENSION $(m*lb)$. On entry, the array <i>x</i> must contain the vector <i>x</i> .

Output Parameters

<i>y</i>	REAL*8. Array, DIMENSION at least $(m*lb)$. On exit, the array <i>y</i> must contain the vector <i>y</i> .

Interfaces

FORTRAN 77:

```

SUBROUTINE mkl_dbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
    CHARACTER*1 uplo, transa, diag
    INTEGER      m, lb
    INTEGER      ia(*), ja(*)
    REAL*8       a(*), x(*), y(*)

```

C:

```
void mkl_dbsrtrsv(char *uplo, char *transa, char *diag, int *m,  
    int *lb, double *a, int *ia, int *ja, double *x, double *y);
```

mkl_dcootrsv

Triangular solvers with simplified interface for a sparse matrix in the coordinate format with one-based indexing.

Syntax

Fortran:

```
call mkl_dcootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
```

C:

```
mkl_dcootrsv(&uplo, &transa, &diag, &m, val, rowind, colind, &nnz, x, y);
```

Description

The `mkl_dcootrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the coordinate format:

$$A * y = x$$

or

$$A' * y = x,$$

where:

x and y are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .



NOTE. This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or low triangle of the matrix <i>A</i> is considered. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix <i>A</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix <i>A</i> is used.
<i>transa</i>	CHARACTER*1. Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then $A*y = x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $A'*y = x$,
<i>diag</i>	CHARACTER*1. Specifies whether <i>A</i> is unit triangular. If <i>diag</i> = 'U' or 'u', then <i>A</i> is unit triangular. If <i>diag</i> = 'N' or 'n', then <i>A</i> is not unit triangular.
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>val</i>	REAL*8. Array of length <i>nnz</i>, contains non-zero elements of the matrix <i>A</i> in the arbitrary order. Refer to <i>values</i> array description in Coordinate Format for more details.
<i>rowind</i>	INTEGER. Array of length <i>nnz</i>, contains the row indices for each non-zero element of the matrix <i>A</i>. Refer to <i>rows</i> array description in Coordinate Format for more details.
<i>colind</i>	INTEGER. Array of length <i>nnz</i>, contains the column indices for each non-zero element of the matrix <i>A</i>. Refer to <i>columns</i> array description in Coordinate Format for more details.
<i>nnz</i>	INTEGER. Specifies the number of non-zero element of the matrix <i>A</i>. Refer to <i>nnz</i> description in Coordinate Format for more details.
<i>x</i>	REAL*8. Array, DIMENSION is <i>m</i>. On entry, the array <i>x</i> must contain the vector <i>x</i>.

Output Parameters

y REAL*8.
 Array, DIMENSION at least *m*.
 Contains the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_dcootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1  uplo, transa, diag
  INTEGER      m, nnz
  INTEGER      rowind(*), colind(*)
  REAL*8       val(*), x(*), y(*)
```

C:

```
void mkl_dcootrsv(char *uplo, char *transa, char *diag, int *m,
  double *alpha, char *matdescra, double *val, int *rowind,
  int *colind, int *nnz, double *x, double *y);
```

mkl_ddiattrsv

Triangular solvers with simplified interface for a sparse matrix in the diagonal format with one-based indexing.

Syntax

Fortran:

```
call mkl_ddiattrsv(uplo, transa, diag, m, val, lval, idiag, ndiag, x, y)
```

C:

```
mkl_ddiattrsv(&uplo, &transa, &diag, &m, val, &lval, idiag, &ndiag, x, y);
```


Description

The `mkl_ddiattrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the diagonal format:

$$A*y = x$$

or

$$A'*y = x,$$

where:

x and y are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .



NOTE. This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or low triangle of the matrix A is used. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix A is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix A is used.
<i>transa</i>	CHARACTER*1. Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then $A*y = x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $A'*y = x$,
<i>diag</i>	CHARACTER*1. Specifies whether A is unit triangular. If <i>diag</i> = 'U' or 'u', then A is unit triangular. If <i>diag</i> = 'N' or 'n', then A is not unit triangular.
<i>m</i>	INTEGER. Number of rows of the matrix A.

<i>val</i>	REAL*8. Two-dimensional array of size <i>lval</i> by <i>ndiag</i> , contains non-zero diagonals of the matrix <i>A</i> . Refer to <i>values</i> array description in Diagonal Storage Scheme for more details.
<i>lval</i>	INTEGER. Leading dimension of <i>val</i> , $lval \geq m$. Refer to <i>lval</i> description in Diagonal Storage Scheme for more details.
<i>idiag</i>	INTEGER. Array of length <i>ndiag</i> , contains the distances between main diagonal and each non-zero diagonals in the matrix <i>A</i> .



NOTE. All elements of this array must be sorted in increasing order.

Refer to *distance* array description in [Diagonal Storage Scheme](#) for more details.

<i>ndiag</i>	INTEGER. Specifies the number of non-zero diagonals of the matrix <i>A</i> .
<i>x</i>	REAL*8. Array, DIMENSION is <i>m</i> . On entry, the array <i>x</i> must contain the vector <i>x</i> .

Output Parameters

<i>y</i>	REAL*8. Array, DIMENSION at least <i>m</i> . Contains the vector <i>y</i> .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_ddiatsrv(uplo, transa, diag, m, val, lval, idiag, ndiag, x, y)
  CHARACTER*1  uplo, transa, diag
  INTEGER      m, lval, ndiag
  INTEGER      idiag(*)
  REAL*8       val(lval,*), x(*), y(*)
```

C:

```
void mkl_ddiatsrv(char *uplo, char *transa, char *diag, int *m, double
  *val, int *lval, int *idiag, int *ndiag, double *x, double *y);
```

mkl_cspblas_dcsrgemv

Computes matrix - vector product of a sparse general matrix stored in the CSR format (3-array variation) with zero-based indexing.

Syntax

Fortran:

```
call mkl_cspblas_dcsrgemv(transa, m, a, ia, ja, x, y)
```

C:

```
mkl_cspblas_dcsrgemv(&transa, &m, a, ia, ja, x, y);
```

Description

The `mkl_cspblas_dcsrgemv` routine performs a matrix-vector operation defined as

$y := A * x$

or

$y := A' * x,$

where:

x and y are vectors,

A is an m -by- m sparse square matrix in the CSR format (3-array variation) with zero-based indexing, A' is the transpose of A .



NOTE. This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then the matrix-vector product is computed as $y := A*x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $y := A'*x$,
<i>m</i>	INTEGER. Number of rows of the matrix A .
<i>a</i>	REAL*8. Array containing non-zero elements of the matrix A. Its length is equal to the number of non-zero elements in the matrix A. Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.
<i>ia</i>	INTEGER. Array of length $m + 1$, containing indices of elements in the array <i>a</i>, such that <i>ia</i>(<i>i</i>) is the index in the array <i>a</i> of the first non-zero element from the row <i>i</i>. The value of the last element <i>ia</i>(<i>m</i>) is equal to the number of non-zeros. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.
<i>ja</i>	REAL*8. Array containing the column indices for each non-zero element of the matrix A. Its length is equal to the length of the array <i>a</i>. Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.
<i>x</i>	REAL*8. Array, DIMENSION is <i>m</i>. One entry, the array <i>x</i> must contain the vector <i>x</i>.

Output Parameters

y REAL*8.
 Array, DIMENSION at least *m*.
 On exit, the array *y* must contain the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_dcsrgemv(transa, m, a, ia, ja, x, y)
  CHARACTER*1 transa
  INTEGER      m
  INTEGER      ia(*), ja(*)
  REAL*8       a(*), x(*), y(*)
```

C:

```
void mkl_cspblas_dcsrgemv(char *transa, int *m, double *a,
int *ia, int *ja, double *x, double *y);
```

mkl_cspblas_dbsrgemv

Computes matrix - vector product of a sparse general matrix stored in the BSR format (3-array variation) with zero-based indexing.

Syntax

Fortran:

```
call mkl_cspblas_dbsrgemv(transa, m, lb, a, ia, ja, x, y)
```

C:

```
mkl_cspblas_dbsrgemv(&transa, &m, &lb, a, ia, ja, x, y);
```

Description

The `mkl_cspblas_dbsrgemv` routine performs a matrix-vector operation defined as

$y := A \cdot x$

or

$y := A' * x,$

where:

x and y are vectors,

A is an m -by- m block sparse square matrix in the BSR format (3-array variation) with zero-based indexing, A' is the transpose of A .



NOTE. This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then the matrix-vector product is computed as $y := A * x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $y := A' * x,$
<i>m</i>	INTEGER. Number of block rows of the matrix A .
<i>lb</i>	INTEGER. Size of the block in the matrix A .
<i>a</i>	REAL*8. Array containing elements of non-zero blocks of the matrix A . Its length is equal to the number of non-zero blocks in the matrix A multiplied by $lb * lb$. Refer to <i>values</i> array description in BSR Format for more details.
<i>ia</i>	INTEGER. Array of length $(m + 1)$, containing indices of block in the array a , such that $ia(i)$ is the index in the array a of the first non-zero element from the row i . The value of the last element $ia(m + 1)$ is equal to the number of non-zero blocks. Refer to <i>rowIndex</i> array description in BSR Format for more details.
<i>ja</i>	REAL*8. Array containing the column indices for each non-zero block in the matrix A .

Its length is equal to the number of non-zero blocks of the matrix *A*. Refer to *columns* array description in [BSR Format](#) for more details.

x REAL*8.
 Array, DIMENSION (*m*lb*).
 On entry, the array *x* must contain the vector *x*.

Output Parameters

y REAL*8.
 Array, DIMENSION at least (*m*lb*).
 On exit, the array *y* must contain the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_dbsrgemv(transa, m, lb, a, ia, ja, x, y)
    CHARACTER*1  transa
    INTEGER      m, lb
    INTEGER      ia(*), ja(*)
    REAL*8       a(*), x(*), y(*)
```

C:

```
void mkl_cspblas_dbsrgemv(char *transa, int *m, int *lb,
    double *a, int *ia, int *ja, double *x, double *y);
```

mkl_cspblas_dcoogemv

Computes matrix - vector product of a sparse general matrix stored in the coordinate format with zero-based indexing.

Syntax

Fortran:

```
call mkl_cspblas_dcoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

C:

```
mkl_cspblas_dcoogemv(&transa, &m, val, rowind, colind, &nnz, x, y);
```

Description

The `mkl_cspblas_dcoogemv` routine performs a matrix-vector operation defined as

$y := A * x$

or

$y := A' * x,$

where:

x and y are vectors,

A is an m -by- m sparse square matrix in the coordinate format with zero-based indexing, A' is the transpose of A .



NOTE. This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then the matrix-vector product is computed as $y := A * x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $y := A' * x,$
<i>m</i>	INTEGER. Number of rows of the matrix A.
<i>val</i>	REAL*8. Array of length <i>nnz</i>, contains non-zero elements of the matrix A in the arbitrary order. Refer to <i>values</i> array description in Coordinate Format for more details.
<i>rowind</i>	INTEGER. Array of length <i>nnz</i>, contains the row indices for each non-zero element of the matrix A. Refer to <i>rows</i> array description in Coordinate Format for more details.

<i>colind</i>	INTEGER. Array of length <i>nnz</i> , contains the column indices for each non-zero element of the matrix <i>A</i> . Refer to <i>columns</i> array description in Coordinate Format for more details.
<i>nnz</i>	INTEGER. Specifies the number of non-zero element of the matrix <i>A</i> . Refer to <i>nnz</i> description in Coordinate Format for more details.
<i>x</i>	REAL*8. Array, DIMENSION is <i>m</i> . On entry, the array <i>x</i> must contain the vector <i>x</i> .

Output Parameters

<i>y</i>	REAL*8. Array, DIMENSION at least <i>m</i> . On exit, the array <i>y</i> must contain the vector <i>y</i> .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_dcoogemv(transa, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1  transa
  INTEGER      m, nnz
  INTEGER      rowind(*), colind(*)
  REAL*8       val(*), x(*), y(*)
```

C:

```
void mkl_cspblas_dcoogemv(char *transa, int *m, double *val, int *rowind,
  int *colind, int *nnz, double *x, double *y);
```

mkl_cspblas_dcsrsvmv

Computes matrix - vector product of a sparse symmetrical matrix stored in the CSR format (3-array variation) with zero-based indexing.

Syntax

Fortran:

```
call mkl_cspblas_dcsrsvmv(uplo, m, a, ia, ja, x, y)
```

C:

```
mkl_cspblas_dcsrsvmv(&uplo, &m, a, ia, ja, x, y);
```

Description

The `mkl_cspblas_dcsrsvmv` routine performs a matrix-vector operation defined as

$y := A * x$

or

$y := A' * x,$

where:

x and y are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the CSR format (3-array variation) with zero-based indexing, A' is the transpose of A .



NOTE. This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<code>uplo</code>	CHARACTER*1. Specifies whether the upper or low triangle of the matrix A is used.

If `uplo = 'U'` or `'u'`, then the upper triangle of the matrix `A` is used.

If `uplo = 'L'` or `'l'`, then the low triangle of the matrix `A` is used.

<code>m</code>	INTEGER. Number of rows of the matrix <code>A</code> .
<code>a</code>	REAL*8. Array containing non-zero elements of the matrix <code>A</code> . Its length is equal to the number of non-zero elements in the matrix <code>A</code> . Refer to <code>values</code> array description in Sparse Matrix Storage Formats for more details.
<code>ia</code>	INTEGER. Array of length $m + 1$, containing indices of elements in the array <code>a</code> , such that <code>ia(i)</code> is the index in the array <code>a</code> of the first non-zero element from the row <code>i</code> . The value of the last element <code>ia(m + 1)</code> is equal to the number of non-zeros. Refer to <code>rowIndex</code> array description in Sparse Matrix Storage Formats for more details.
<code>ja</code>	REAL*8. Array containing the column indices for each non-zero element of the matrix <code>A</code> . Its length is equal to the length of the array <code>a</code> . Refer to <code>columns</code> array description in Sparse Matrix Storage Formats for more details.
<code>x</code>	REAL*8. Array, DIMENSION is <code>m</code> . On entry, the array <code>x</code> must contain the vector <code>x</code> .

Output Parameters

<code>y</code>	REAL*8. Array, DIMENSION at least <code>m</code> . On exit, the array <code>y</code> must contain the vector <code>y</code> .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_dcsrsymv(uplo, m, a, ia, ja, x, y)
    CHARACTER*1 uplo
    INTEGER      m
    INTEGER      ia(*), ja(*)
    REAL*8       a(*), x(*), y(*)
```

C:

```
void mkl_cspblas_dcsrsymv(char *uplo, int *m, double *a,
    int *ia, int *ja, double *x, double *y);
```

mkl_cspblas_dbsrsymv

Computes matrix - vector product of a sparse symmetrical matrix stored in the BSR format (3-arrays variation) with zero-based indexing.

Syntax

Fortran:

```
call mkl_cspblas_dbsrsymv(uplo, m, lb, a, ia, ja, x, y)
```

C:

```
mkl_cspblas_dbsrsymv(&uplo, &m, &lb, a, ia, ja, x, y);
```

Description

The `mkl_cspblas_dbsrsymv` routine performs a matrix-vector operation defined as

$y := A * x$

or

$y := A' * x,$

where:

x and y are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the BSR format (3-array variation) with zero-based indexing, A' is the transpose of A .



NOTE. This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or low triangle of the matrix A is used. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix A is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix A is used.
<i>m</i>	INTEGER. Number of block rows of the matrix A .
<i>lb</i>	INTEGER. Size of the block in the matrix A .
<i>a</i>	REAL*8. Array containing elements of non-zero blocks of the matrix A . Its length is equal to the number of non-zero blocks in the matrix A multiplied by $lb*lb$. Refer to <i>values</i> array description in BSR Format for more details.
<i>ia</i>	INTEGER. Array of length $(m + 1)$, containing indices of block in the array <i>a</i> , such that <i>ia</i> (<i>i</i>) is the index in the array <i>a</i> of the first non-zero element from the row <i>i</i> . The value of the last element <i>ia</i> ($m + 1$) is equal to the number of non-zero blocks plus one. Refer to <i>rowIndex</i> array description in BSR Format for more details.
<i>ja</i>	REAL*8. Array containing the column indices for each non-zero block in the matrix A . Its length is equal to the number of non-zero blocks of the matrix A . Refer to <i>columns</i> array description in BSR Format for more details.
<i>x</i>	REAL*8.

Array, DIMENSION $(m*lb)$.
On entry, the array x must contain the vector x .

Output Parameters

y REAL*8.
Array, DIMENSION at least $(m*lb)$.
On exit, the array y must contain the vector y .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_dbsrsymv(uplo, m, lb, a, ia, ja, x, y)
  CHARACTER*1 uplo
  INTEGER      m, lb
  INTEGER      ia(*), ja(*)
  REAL*8       a(*), x(*), y(*)
```

C:

```
void mkl_cspblas_dbsrsymv(char *uplo, int *m, int *lb,
double *a, int *ia, int *ja, double *x, double *y);
```

mkl_cspblas_dcoosymv

Computes matrix - vector product of a sparse symmetrical matrix stored in the coordinate format with zero-based indexing .

Syntax

Fortran:

```
call mkl_cspblas_dcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
```

C:

```
mkl_cspblas_dcoosymv(&uplo, &m, val, rowind, colind, &nnz, x, y);
```

Description

The `mk1_cspblas_dcoosymv` routine performs a matrix-vector operation defined as

$$y := A * x$$

or

$$y := A' * x,$$

where:

x and y are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the coordinate format with zero-based indexing, A' is the transpose of A .



NOTE. This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or low triangle of the matrix A is used. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix A is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix A is used.
<i>m</i>	INTEGER. Number of rows of the matrix A .
<i>val</i>	REAL*8. Array of length <i>nnz</i>, contains non-zero elements of the matrix A in the arbitrary order. Refer to <i>values</i> array description in Coordinate Format for more details.
<i>rowind</i>	INTEGER. Array of length <i>nnz</i>, contains the row indices for each non-zero element of the matrix A. Refer to <i>rows</i> array description in Coordinate Format for more details.

<i>colind</i>	INTEGER. Array of length <i>nnz</i> , contains the column indices for each non-zero element of the matrix <i>A</i> . Refer to <i>columns</i> array description in Coordinate Format for more details.
<i>nnz</i>	INTEGER. Specifies the number of non-zero element of the matrix <i>A</i> . Refer to <i>nnz</i> description in Coordinate Format for more details.
<i>x</i>	REAL*8. Array, DIMENSION is <i>m</i> . On entry, the array <i>x</i> must contain the vector <i>x</i> .

Output Parameters

<i>y</i>	REAL*8. Array, DIMENSION at least <i>m</i> . On exit, the array <i>y</i> must contain the vector <i>y</i> .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_dcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1  uplo
  INTEGER      m, nnz
  INTEGER      rowind(*), colind(*)
  REAL*8       val(*), x(*), y(*)
```

C:

```
void mkl_cspblas_dcoosymv(char *uplo, int *m, double *val, int *rowind,
  int *colind, int *nnz, double *x, double *y);
```


mkl_cspblas_dcsrtrsv

Triangular solvers with simplified interface for a sparse matrix in the CSR format (3-array variation) with zero-based indexing.

Syntax

Fortran:

```
call mkl_cspblas_dcsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
```

C:

```
mkl_cspblas_dcsrtrsv(&uplo, &transa, &diag, &m, a, ia, ja, x, y);
```

Description

The `mkl_cspblas_dcsrtrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the CSR format (3-array variation) with zero-based indexing:

$$A*y = x$$

or

$$A'*y = x,$$

where:

x and y are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .



NOTE. This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

`uplo` CHARACTER*1. Specifies whether the upper or low triangle of the matrix A is used.

	If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix <i>A</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix <i>A</i> is used.
<i>transa</i>	CHARACTER*1. Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then $A*y = x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $A'*y = x$,
<i>diag</i>	CHARACTER*1. Specifies whether matrix <i>A</i> is unit triangular. If <i>diag</i> = 'U' or 'u', then <i>A</i> is unit triangular. If <i>diag</i> = 'N' or 'n', then <i>A</i> is not unit triangular.
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>a</i>	REAL*8. Array containing non-zero elements of the matrix <i>A</i>. Its length is equal to the number of non-zero elements in the matrix <i>A</i>. Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.
<i>ia</i>	INTEGER. Array of length <i>m</i>+1, containing indices of elements in the array <i>a</i>, such that <i>ia</i>(<i>i</i>) is the index in the array <i>a</i> of the first non-zero element from the row <i>i</i>. The value of the last element <i>ia</i>(<i>m</i>) is equal to the number of non-zeros. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.
<i>ja</i>	REAL*8. Array containing the column indices for each non-zero element of the matrix <i>A</i>. Its length is equal to the length of the array <i>a</i>. Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.
	 NOTE. Column indices must be sorted in increasing order for each row.
<i>x</i>	REAL*8. Array, DIMENSION is <i>m</i>. On entry, the array <i>x</i> must contain the vector <i>x</i>.

Output Parameters

y REAL*8.
Array, DIMENSION at least *m*.
Contains the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_dcsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
  CHARACTER*1 uplo, transa, diag
  INTEGER      m
  INTEGER      ia(*), ja(*)
  REAL*8       a(*), x(*), y(*)
```

C:

```
void mkl_cspblas_dcsrtrsv(char *uplo, char *transa, char *diag, int *m,
double *a, int *ia, int *ja, double *x, double *y);
```

mkl_cspblas_dbsrtrsv

Triangular solver with simplified interface for a sparse matrix stored in the BSR format (3-array variation) with zero-based indexing.

Syntax

Fortran:

```
call mkl_cspblas_dbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
```

C:

```
mkl_cspblas_dbsrtrsv(&uplo, &transa, &diag, &m, &lb, a, ia, ja, x, y);
```

Description

The `mk1_cspblas_dbsrtrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the BSR format (3-array variation) with zero-based indexing:

$y := A * x$

or

$y := A' * x,$

where:

x and y are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .



NOTE. This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>uplo</i>	CHARACTER*1. Specifies the upper or low triangle of the matrix A is used. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix A is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix A is used.
<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then the matrix-vector product is computed as $y := A * x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $y := A' * x$.
<i>diag</i>	CHARACTER*1. Specifies whether matrix A is unit triangular or not. If <i>diag</i> = 'U' or 'u', A is unit triangular. If <i>diag</i> = 'N' or 'n', A is not unit triangular.

<i>m</i>	INTEGER. Number of block rows of the matrix <i>A</i> .
<i>lb</i>	INTEGER. Size of the block in the matrix <i>A</i> .
<i>a</i>	REAL*8. Array containing elements of non-zero blocks of the matrix <i>A</i> . Its length is equal to the number of non-zero blocks in the matrix <i>A</i> multiplied by <i>lb*lb</i> . Refer to <i>values</i> array description in BSR Format for more details.
<i>ia</i>	INTEGER. Array of length $(m + 1)$, containing indices of block in the array <i>a</i> , such that <i>ia(I)</i> is the index in the array <i>a</i> of the first non-zero element from the row <i>I</i> . The value of the last element <i>ia(m + 1)</i> is equal to the number of non-zero blocks. Refer to <i>rowIndex</i> array description in BSR Format for more details.
<i>ja</i>	REAL*8. Array containing the column indices for each non-zero block in the matrix <i>A</i> . Its length is equal to the number of non-zero blocks of the matrix <i>A</i> . Refer to <i>columns</i> array description in BSR Format for more details.
<i>x</i>	REAL*8. Array, DIMENSION $(m*lb)$. On entry, the array <i>x</i> must contain the vector <i>x</i> .

Output Parameters

<i>y</i>	REAL*8. Array, DIMENSION at least $(m*lb)$. On exit, the array <i>y</i> must contain the vector <i>y</i> .

Interfaces

FORTRAN 77:

```

SUBROUTINE mkl_cspblas_dbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
    CHARACTER*1 uplo, transa, diag
    INTEGER      m, lb
    INTEGER      ia(*), ja(*)
    REAL*8       a(*), x(*), y(*)

```

C:

```
void mkl_cspblas_dbsrtrsv(char *uplo, char *transa, char *diag, int *m,  
int *lb, double *a, int *ia, int *ja, double *x, double *y);
```

mkl_cspblas_dcootrsv

Triangular solvers with simplified interface for a sparse matrix in the coordinate format with zero-based indexing .

Syntax

Fortran:

```
call mkl_cspblas_dcootrsv(uplo, transa, diag, m, val, rowind, colind, nnz,  
x, y)
```

C:

```
mkl_cspblas_dcootrsv(&uplo, &transa, &diag, &m, val, rowind, colind, &nnz,  
x, y);
```

Description

The `mkl_cspblas_dcootrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the coordinate format with zero-based indexing:

$$A * y = x$$

or

$$A' * y = x,$$

where:

x and y are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .



NOTE. This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or low triangle of the matrix <i>A</i> is considered. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix <i>A</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix <i>A</i> is used.
<i>transa</i>	CHARACTER*1. Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then $A*y = x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $A'*y = x$,
<i>diag</i>	CHARACTER*1. Specifies whether <i>A</i> is unit triangular. If <i>diag</i> = 'U' or 'u', then <i>A</i> is unit triangular. If <i>diag</i> = 'N' or 'n', then <i>A</i> is not unit triangular.
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>val</i>	REAL*8. Array of length <i>nnz</i>, contains non-zero elements of the matrix <i>A</i> in the arbitrary order. Refer to <i>values</i> array description in Coordinate Format for more details.
<i>rowind</i>	INTEGER. Array of length <i>nnz</i>, contains the row indices for each non-zero element of the matrix <i>A</i>. Refer to <i>rows</i> array description in Coordinate Format for more details.
<i>colind</i>	INTEGER. Array of length <i>nnz</i>, contains the column indices for each non-zero element of the matrix <i>A</i>. Refer to <i>columns</i> array description in Coordinate Format for more details.
<i>nnz</i>	INTEGER. Specifies the number of non-zero element of the matrix <i>A</i>. Refer to <i>nnz</i> description in Coordinate Format for more details.
<i>x</i>	REAL*8. Array, DIMENSION is <i>m</i>. On entry, the array <i>x</i> must contain the vector <i>x</i>.

Output Parameters

y REAL*8.
 Array, DIMENSION at least *m*.
 Contains the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_dcootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1  uplo, transa, diag
  INTEGER      m, nnz
  INTEGER      rowind(*), colind(*)
  REAL*8       val(*), x(*), y(*)
```

C:

```
void mkl_cspblas_dcootrsv(char *uplo, char *transa, char *diag, int *m,
  double *alpha, char *matdescra, double *val, int *rowind,
  int *colind, int *nnz, double *x, double *y);
```

mkl_dcsrsv

Computes matrix - vector product of a sparse matrix stored in the CSR format.

Syntax

Fortran:

```
call mkl_dcsrsv(transa, m, k, alpha, matdescra, val, indx, pntreb, pntre, x,
  beta, y)
```

C:

```
mkl_dcsrsv(&transa, &m, &k, &alpha, matdescra, val, indx, pntreb, pntre, x,
  &beta, y);
```


Description

The `mkl_dcsrsv` routine performs a matrix-vector operation defined as

$$y := \alpha A x + \beta y$$

or

$$y := \alpha A' x + \beta y,$$

where:

α and β are scalars,

x and y are vectors,

A is an m -by- k sparse matrix in the CSR format, A' is the transpose of A .



NOTE. This routine supports CSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then $y := \alpha A x + \beta y$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha A' x + \beta y$,
<i>m</i>	INTEGER. Number of rows of the matrix A .
<i>k</i>	INTEGER. Number of columns of the matrix A .
<i>alpha</i>	REAL*8. Specifies the scalar α .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
<i>val</i>	REAL*8. Array containing non-zero elements of the matrix A . For one-based indexing its length is $pntrb(m) - pntrb(1)$. For zero-based indexing its length is $pntrb(m-1) - pntrb(0)$.

	Refer to <i>values</i> array description in CSR Format for more details.
<i>indx</i>	INTEGER. Array containing the column indices for each non-zero element of the matrix <i>A</i> . Its length is equal to length of the <i>val</i> array. Refer to <i>columns</i> array description in CSR Format for more details.
<i>pntrb</i>	INTEGER. Array of length <i>m</i> . For one-based indexing this array contains row indices, such that <i>pntrb(i) - pntrb(1)+1</i> is the first index of row <i>i</i> in the arrays <i>val</i> and <i>indx</i> . For zero-based indexing this array contains row indices, such that <i>pntrb(i) - pntrb(0)</i> is the first index of row <i>i</i> in the arrays <i>val</i> and <i>indx</i> . Refer to <i>pointerb</i> array description in CSR Format for more details.
<i>pntrr</i>	INTEGER. Array of length <i>m</i> . For one-based indexing this array contains row indices, such that <i>pntrr(i) - pntrb(1)</i> is the last index of row <i>i</i> in the arrays <i>val</i> and <i>indx</i> . For zero-based indexing this array contains row indices, such that <i>pntrr(i) - pntrb(0)-1</i> is the last index of row <i>i</i> in the arrays <i>val</i> and <i>indx</i> . Refer to <i>pointerE</i> array description in CSR Format for more details.
<i>x</i>	REAL*8. Array, DIMENSION at least <i>k</i> if <i>transa</i> = 'N' or 'n' and at least <i>m</i> otherwise. On entry, the array <i>x</i> must contain the vector <i>x</i> .
<i>beta</i>	REAL*8. Specifies the scalar <i>beta</i> .
<i>y</i>	REAL*8. Array, DIMENSION at least <i>m</i> if <i>transa</i> = 'N' or 'n' and at least <i>k</i> otherwise. On entry, the array <i>y</i> must contain the vector <i>y</i> .

Output Parameters

<i>y</i>	Overwritten by the updated vector <i>y</i> .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_dcsrsv(transa, m, k, alpha, matdescra, val, indx,
    pntrb, pntre, x, beta, y)
    CHARACTER*1    transa
    CHARACTER       matdescra(*)
    INTEGER         m, k
    INTEGER         indx(*), pntrb(m), pntre(m)
    REAL*8         alpha, beta
    REAL*8         val(*), x(*), y(*)
```

C:

```
void mkl_dcsrsv(char *transa, int *m, int *k, double *alpha, char *matdescra,
    double *val, int *indx, int *pntrb, int *pntre, double *x, double *beta, double *y);
```

mkl_dbsrmv

Computes matrix - vector product of a sparse matrix stored in the BSR format.

Syntax

Fortran:

```
call mkl_dbsrmv(transa, m, k, lb, alpha, matdescra, val, indx, pntrb, pntre,
    x, beta, y)
```

C:

```
mkl_dbsrmv(&transa, &m, &k, &lb, &alpha, matdescra, val, indx, pntrb, pntre,
    x, &beta, y);
```

Description

The `mkl_dbsrmv` routine performs a matrix-vector operation defined as

$$y := \alpha A x + \beta y$$

or

$$y := \alpha A'x + \beta y,$$

where:

α and β are scalars,

x and y are vectors,

A is an m -by- k block sparse matrix in the BSR format, A' is the transpose of A .



NOTE. This routine supports BSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then the matrix-vector product is computed as $y := \alpha A^*x + \beta y$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $y := \alpha A'^*x + \beta y$,
<i>m</i>	INTEGER. Number of block rows of the matrix <i>A</i> .
<i>k</i>	INTEGER. Number of block columns of the matrix <i>A</i> .
<i>lb</i>	INTEGER. Size of the block in the matrix <i>A</i> .
<i>alpha</i>	REAL*8. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
<i>val</i>	REAL*8. Array containing elements of non-zero blocks of the matrix <i>A</i> . Its length is equal to the number of non-zero blocks in the matrix <i>A</i> multiplied by $lb*lb$. Refer to <i>values</i> array description in BSR Format for more details.

<i>indx</i>	INTEGER. Array containing the column indices for each non-zero block in the matrix <i>A</i>. Its length is equal to the number of non-zero blocks in the matrix <i>A</i>. Refer to <i>columns</i> array description in BSR Format for more details.
<i>pntrb</i>	INTEGER. Array of length <i>m</i>. For one-based indexing: this array contains row indices, such that $pntrb(i) - pntrb(1) + 1$ is the first index of block row <i>i</i> in the array <i>indx</i>. For zero-based indexing: this array contains row indices, such that $pntrb(i) - pntrb(0)$ is the first index of block row <i>i</i> in the array <i>indx</i>. Refer to <i>pointerB</i> array description in BSR Format for more details.
<i>pntre</i>	INTEGER. Array of length <i>m</i>. For one-based indexing this array contains row indices, such that $pntre(i) - pntrb(1)$ is the last index of block row <i>i</i> in the array <i>indx</i>. For zero-based indexing this array contains row indices, such that $pntre(i) - pntrb(0) - 1$ is the last index of block row <i>i</i> in the array <i>indx</i>. Refer to <i>pointerE</i> array description in BSR Format for more details.
<i>x</i>	REAL*8. Array, DIMENSION at least $(k*lb)$ if <i>transa</i> = 'N' or 'n', and at least $(m*lb)$ otherwise. On entry, the array <i>x</i> must contain the vector <i>x</i>.
<i>beta</i>	REAL*8. Specifies the scalar <i>beta</i> .
<i>y</i>	REAL*8. Array, DIMENSION at least $(m*lb)$ if <i>transa</i> = 'N' or 'n', and at least $(k*lb)$ otherwise. On entry, the array <i>y</i> must contain the vector <i>y</i>.

Output Parameters

<i>y</i>	Overwritten by the updated vector <i>y</i> .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_dbsrmv(transa, m, k, lb, alpha, matdescra, val, indx,  
  pntrb, pntre, x, beta, y)  
  CHARACTER*1  transa  
  CHARACTER    matdescra(*)  
  INTEGER      m, k, lb  
  INTEGER      indx(*), pntrb(m), pntre(m)  
  REAL*8       alpha, beta  
  REAL*8       val(*), x(*), y(*)
```

C:

```
void mkl_dbsrmv(char *transa, int *m, int *k, int *lb,  
  
double *alpha, char *matdescra, double *val, int *indx,  
  
int *pntrb, int *pntre, double *x, double *beta, double *y);
```

mkl_dcscmv

Computes matrix - vector product for a sparse matrix in the CSC format.

Syntax

Fortran:

```
call mkl_dcscmv(transa, m, k, alpha, matdescra, val, indx, pntrb, pntre, x,  
beta, y)
```

C:

```
mkl_dcscmv(&transa, &m, &k, &alpha, matdescra, val, indx, pntrb, pntre, x,  
&beta, y);
```

Description

The `mk1_dcscmv` routine performs a matrix-vector operation defined as

$$y := \alpha A x + \beta y$$

or

$$y := \alpha A' x + \beta y,$$

where:

α and β are scalars,

x and y are vectors,

A is an m -by- k sparse matrix in compressed sparse column (CSC) format, A' is the transpose of A .



NOTE. This routine supports CSC format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then $y := \alpha A x + \beta y$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha A' x + \beta y$,
<i>m</i>	INTEGER. Number of rows of the matrix A .
<i>k</i>	INTEGER. Number of columns of the matrix A .
<i>alpha</i>	REAL*8. Specifies the scalar α .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
<i>val</i>	REAL*8. Array containing non-zero elements of the matrix A . For one-based indexing its length is $\text{pntrc}(k) - \text{pntrb}(1)$.

	For zero-based indexing its length is $pntre(m-1) - pntrb(0)$. Refer to <i>values</i> array description in CSC Format for more details.
<i>indx</i>	INTEGER. Array containing the row indices for each non-zero element of the matrix <i>A</i>. Its length is equal to length of the <i>val</i> array. Refer to <i>rows</i> array description in CSC Format for more details.
<i>pntrb</i>	INTEGER. Array of length <i>k</i>. For one-based indexing this array contains column indices, such that $pntrb(i) - pntrb(1) + 1$ is the first index of column <i>i</i> in the arrays <i>val</i> and <i>indx</i>. For zero-based indexing this array contains column indices, such that $pntrb(i) - pntrb(0)$ is the first index of column <i>i</i> in the arrays <i>val</i> and <i>indx</i>. Refer to <i>pointerb</i> array description in CSC Format for more details.
<i>pntre</i>	INTEGER. Array of length <i>k</i>. For one-based indexing this array contains column indices, such that $pntre(i) - pntrb(1)$ is the last index of column <i>i</i> in the arrays <i>val</i> and <i>indx</i>. For zero-based indexing this array contains column indices, such that $pntre(i) - pntrb(1) - 1$ is the last index of column <i>i</i> in the arrays <i>val</i> and <i>indx</i>. Refer to <i>pointerE</i> array description in CSC Format for more details.
<i>x</i>	REAL*8. Array, DIMENSION at least <i>k</i> if <i>transa</i> = 'N' or 'n' and at least <i>m</i> otherwise. On entry, the array <i>x</i> must contain the vector <i>x</i>.
<i>beta</i>	REAL*8. Specifies the scalar <i>beta</i>.
<i>y</i>	REAL*8. Array, DIMENSION at least <i>m</i> if <i>transa</i> = 'N' or 'n' and at least <i>k</i> otherwise. On entry, the array <i>y</i> must contain the vector <i>y</i>.

Output Parameters

y Overwritten by the updated vector y .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_dcscmv(transa, m, k, alpha, matdescra, val, indx,
  pntrb, pntre, x, beta, y)
  CHARACTER*1  transa
  CHARACTER    matdescra(*)
  INTEGER      m, k, ldb, ldc
  INTEGER      indx(*), pntrb(m), pntre(m)
  REAL*8       alpha, beta
  REAL*8       val(*), x(*), y(*)
```

C:

```
void mkl_dcscmv(char *transa, int *m, int *k, double *alpha,
  char *matdescra, double *val, int *indx, int *pntrb,
  int *pntre, double *x, double *beta, double *y);
```

mkl_dcoomv

Computes matrix - vector product for a sparse matrix in the coordinate format.

Syntax

Fortran:

```
call mkl_dcoomv(transa, m, k, alpha, matdescra, val, rowind, colind, nnz, x,
  beta, y)
```

C:

```
mkl_dcoomv(&transa, &m, &k, &alpha, matdescra, val, rowind, colind, &nnz, x,
&beta, y);
```

Description

The `mkl_dcoomv` routine performs a matrix-vector operation defined as

$$y := \alpha * A * x + \beta * y$$

or

$$y := \alpha * A' * x + \beta * y,$$

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* sparse matrix in compressed coordinate format, *A'* is the transpose of *A*.



NOTE. This routine supports coordinate format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then $y := \alpha * A * x + \beta * y$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha * A' * x + \beta * y$,
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>k</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>alpha</i>	REAL*8. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .

<i>val</i>	REAL*8. Array of length <i>nnz</i> , contains non-zero elements of the matrix <i>A</i> in the arbitrary order. Refer to <i>values</i> array description in Coordinate Format for more details.
<i>rowind</i>	INTEGER. Array of length <i>nnz</i> , contains the row indices for each non-zero element of the matrix <i>A</i> . Refer to <i>rows</i> array description in Coordinate Format for more details.
<i>colind</i>	INTEGER. Array of length <i>nnz</i> , contains the column indices for each non-zero element of the matrix <i>A</i> . Refer to <i>columns</i> array description in Coordinate Format for more details.
<i>nnz</i>	INTEGER. Specifies the number of non-zero element of the matrix <i>A</i> . Refer to <i>nnz</i> description in Coordinate Format for more details.
<i>x</i>	REAL*8. Array, DIMENSION at least <i>k</i> if <i>transa</i> = 'N' or 'n' and at least <i>m</i> otherwise. On entry, the array <i>x</i> must contain the vector <i>x</i> .
<i>beta</i>	REAL*8. Specifies the scalar <i>beta</i> .
<i>y</i>	REAL*8. Array, DIMENSION at least <i>m</i> if <i>transa</i> = 'N' or 'n' and at least <i>k</i> otherwise. On entry, the array <i>y</i> must contain the vector <i>y</i> .

Output Parameters

<i>y</i>	Overwritten by the updated vector <i>y</i> .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_dcoomv(transa, m, k, alpha, matdescra, val, rowind, colind, nnz, x, beta,
y)

  CHARACTER*1    transa
  CHARACTER      matdescra(*)
  INTEGER        m, k, nnz
  INTEGER        rowind(*), colind(*)
  REAL*8         alpha, beta
  REAL*8         val(*), x(*), y(*)
```

C:

```
void mkl_dcoomv(char *transa, int *m, int *k, double *alpha, char *matdescra,
double *val, int *rowind, int *colind, int *nnz, double *x, double *beta, double *y);
```

mkl_dcsrsv

Solves a system of linear equations for a sparse matrix in the CSR format.

Syntax

Fortran:

```
call mkl_dcsrsv(transa, m, alpha, matdescra, val, indx, pntreb, pntre, x, y)
```

C:

```
mkl_dcsrsv(&transa, &m, &alpha, matdescra, val, indx, pntreb, pntre, x, y);
```

Description

The `mkl_dcsrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the CSR format:

$$y := \alpha * \text{inv}(A) * x$$

or

$$y := \alpha * \text{inv}(A') * x,$$

where:

α is scalar, x and y are vectors, A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .



NOTE. This routine supports CSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then $y := \alpha * \text{inv}(A) * x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha * \text{inv}(A') * x$,
<i>m</i>	INTEGER. Number of columns of the matrix A .
<i>alpha</i>	REAL*8. Specifies the scalar α .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
<i>val</i>	REAL*8. Array containing non-zero elements of the matrix A . For one-based indexing its length is $\text{pntrc}(m) - \text{pntrb}(1)$. For zero-based indexing its length is $\text{pntrc}(m-1) - \text{pntrb}(0)$. Refer to <i>values</i> array description in CSR Format for more details.
<i>indx</i>	INTEGER. Array containing the column indices for each non-zero element of the matrix A . Its length is equal to length of the <i>val</i> array. Refer to <i>columns</i> array description in CSR Format for more details.



NOTE. Column indices must be sorted in increasing order for each row.

pntrb

INTEGER. Array of length m .

For one-based indexing this array contains row indices, such that $pntrb(i) - pntrb(1) + 1$ is the first index of row i in the arrays *val* and *indx*.

For zero-based indexing this array contains row indices, such that $pntrb(i) - pntrb(0)$ is the first index of row i in the arrays *val* and *indx*. Refer to *pointerb* array description in [CSR Format](#) for more details.

pntrb

INTEGER. Array of length m .

For one-based indexing this array contains row indices, such that $pntrb(i) - pntrb(1)$ is the last index of row i in the arrays *val* and *indx*.

For zero-based indexing this array contains row indices, such that $pntrb(i) - pntrb(0) - 1$ is the last index of row i in the arrays *val* and *indx*. Refer to *pointerE* array description in [CSR Format](#) for more details.

x

REAL*8.

Array, DIMENSION at least m .

On entry, the array *x* must contain the vector *x*. The elements are accessed with unit increment.

y

REAL*8.

Array, DIMENSION at least m .

On entry, the array *y* must contain the vector *y*. The elements are accessed with unit increment.

Output Parameters

y

Contains solution vector *x*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_dcsrsv(transa, m, alpha, matdescra, val, indx, pntbr, pntre, x, y)

  CHARACTER*1  transa
  CHARACTER    matdescra(*)
  INTEGER      m
  INTEGER      indx(*), pntbr(m), pntre(m)
  REAL*8       alpha
  REAL*8       val(*)
  REAL*8       x(*), y(*)
```

C:

```
void mkl_dcsrsv(char *transa, int *m, double *alpha, char *matdescra,
  double *val, int *indx, int *pntbr, int *pntre, double *x, double *y);
```

mkl_dbsrsv

Solves a system of linear equations for a sparse matrix in the BSR format.

Syntax

Fortran:

```
call mkl_dbsrsv(transa, m, lb, alpha, matdescra, val, indx, pntbr, pntre, x,
y)
```

C:

```
mkl_dbsrsv(&transa, &m, &lb, &alpha, matdescra, val, indx, pntbr, pntre, x,
y);
```

Description

The `mkl_dcsrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the BSR format:

$$y := \alpha * \text{inv}(A) * x$$

or

$$y := \alpha * \text{inv}(A') * x,$$

where:

α is scalar, x and y are vectors, A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .



NOTE. This routine supports BSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then $y := \alpha * \text{inv}(A) * x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha * \text{inv}(A') * x$,
<i>m</i>	INTEGER. Number of block columns of the matrix <i>A</i> .
<i>lb</i>	INTEGER. Size of the block in the matrix <i>A</i> .
<i>alpha</i>	REAL*8. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
<i>val</i>	REAL*8. Array containing elements of non-zero blocks of the matrix <i>A</i> . Its length is equal to the number of non-zero blocks in the matrix <i>A</i> multiplied by <i>lb*lb</i> . Refer to the <i>values</i> array description in BSR Format for more details.

<i>indx</i>	INTEGER. Array containing the column indices for each non-zero block in the matrix <i>A</i>. Its length is equal to the number of non-zero blocks in the matrix <i>A</i>. Refer to the <i>columns</i> array description in BSR Format for more details.
<i>pntrb</i>	INTEGER. Array of length <i>m</i>. For one-based indexing: this array contains row indices, such that $pntrb(i) - pntrb(1) + 1$ is the first index of block row <i>i</i> in the array <i>indx</i>. For zero-based indexing: this array contains row indices, such that $pntrb(i) - pntrb(0)$ is the first index of block row <i>i</i> in the array <i>indx</i>. Refer to <i>pointerB</i> array description in BSR Format for more details.
<i>pntre</i>	INTEGER. Array of length <i>m</i>. For one-based indexing this array contains row indices, such that $pntre(i) - pntrb(1)$ is the last index of block row <i>i</i> in the array <i>indx</i>. For zero-based indexing this array contains row indices, such that $pntre(i) - pntrb(0) - 1$ is the last index of block row <i>i</i> in the array <i>indx</i>. Refer to <i>pointerE</i> array description in BSR Format for more details.
<i>x</i>	REAL*8. Array, DIMENSION at least $(m*lb)$. On entry, the array <i>x</i> must contain the vector <i>x</i>. The elements are accessed with unit increment.
<i>y</i>	REAL*8. Array, DIMENSION at least $(m*lb)$. On entry, the array <i>y</i> must contain the vector <i>y</i>. The elements are accessed with unit increment.

Output Parameters

<i>y</i>	Contains solution vector <i>x</i> .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_dbsrsv(transa, m, lb, alpha, matdescra, val, indx, pntrb, pntre, x, y)
  CHARACTER*1  transa
  CHARACTER    matdescra(*)
  INTEGER      m, lb
  INTEGER      indx(*), pntrb(m), pntre(m)
  REAL*8       alpha
  REAL*8       val(*)
  REAL*8       x(*), y(*)
```

C:

```
void mkl_dbsrsv(char *transa, int *m, int *lb, double *alpha, char *matdescra,
  double *val, int *indx, int *pntrb, int *pntre, double *x, double *y);
```

mkl_dcscsv

Solves a system of linear equations for a sparse matrix in the CSC format.

Syntax

Fortran:

```
call mkl_dcscsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

C:

```
mkl_dcscsv(&transa, &m, &alpha, matdescra, val, indx, pntrb, pntre, x, y);
```

Description

The `mkl_dcscsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the CSC format:

$$y := \alpha * \text{inv}(A) * x$$

or

$$y := \alpha * \text{inv}(A') * x,$$

where:

α is scalar, x and y are vectors, A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .



NOTE. This routine supports CSC format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then $y := \alpha * \text{inv}(A) * x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha * \text{inv}(A') * x$,
<i>m</i>	INTEGER. Number of columns of the matrix A .
<i>alpha</i>	REAL*8. Specifies the scalar α .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
<i>val</i>	REAL*8. Array containing non-zero elements of the matrix A . For one-based indexing its length is $\text{pntrc}(m) - \text{pntrb}(1)$. For zero-based indexing its length is $\text{pntrc}(m-1) - \text{pntrb}(0)$. Refer to <i>values</i> array description in CSC Format for more details.
<i>indx</i>	INTEGER. Array containing the row indices for each non-zero element of the matrix A . Its length is equal to length of the <i>val</i> array. Refer to <i>columns</i> array description in CSC Format for more details.



NOTE. Row indices must be sorted in increasing order for each column.

pntrb

INTEGER. Array of length *m*.

For one-based indexing this array contains column indices, such that $pntrb(i) - pntrb(1) + 1$ is the first index of column *i* in the arrays *val* and *indx*.

For zero-based indexing this array contains column indices, such that $pntrb(i) - pntrb(0)$ is the first index of column *i* in the arrays *val* and *indx*.

Refer to *pointerb* array description in [CSC Format](#) for more details.

pntrc

INTEGER. Array of length *m*.

For one-based indexing this array contains column indices, such that $pntrc(i) - pntrb(1)$ is the last index of column *i* in the arrays *val* and *indx*.

For zero-based indexing this array contains column indices, such that $pntrc(i) - pntrb(1) - 1$ is the last index of column *i* in the arrays *val* and *indx*.

Refer to *pointerE* array description in [CSC Format](#) for more details.

x

REAL*8.

Array, DIMENSION at least *m*.

On entry, the array *x* must contain the vector *x*. The elements are accessed with unit increment.

y

REAL*8.

Array, DIMENSION at least *m*.

On entry, the array *y* must contain the vector *y*. The elements are accessed with unit increment.

Output Parameters

y

Contains the solution vector *x*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_dcscsv(transa, m, alpha, matdescra, val, indx, pntbr, pntre, x, y)

  CHARACTER*1  transa
  CHARACTER    matdescra(*)
  INTEGER      m
  INTEGER      indx(*), pntbr(m), pntre(m)
  REAL*8       alpha
  REAL*8       val(*)
  REAL*8       x(*), y(*)
```

C:

```
void mkl_dcscsv(char *transa, int *m, double *alpha, char *matdescra,
  double *val, int *indx, int *pntbr, int *pntre, double *x, double *y);
```

mkl_dcoosv

Solves a system of linear equations for a sparse matrix in the coordinate format.

Syntax

Fortran:

```
call mkl_dcoosv(transa, m, k, alpha, matdescra, val, rowind, colind, nnz, x,
y)
```

C:

```
mkl_dcoosv(&transa, &m, &k, &alpha, matdescra, val, rowind, colind, &nnz, x,
y);
```

Description

The `mkl_dcoosv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the coordinate format:

$$y := \alpha * \text{inv}(A) * x$$

or

$$y := \alpha * \text{inv}(A') * x,$$

where:

α is scalar, x and y are vectors, A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .



NOTE. This routine supports coordinate format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>transa</i>	CHARACTER*1. Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then $y := \alpha * \text{inv}(A) * x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha * \text{inv}(A') * x$,
<i>m</i>	INTEGER. Number of rows of the matrix A .
<i>alpha</i>	REAL*8. Specifies the scalar α .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
<i>val</i>	REAL*8. Array of length <i>nnz</i> , contains non-zero elements of the matrix A in the arbitrary order. Refer to <i>values</i> array description in Coordinate Format for more details.
<i>rowind</i>	INTEGER. Array of length <i>nnz</i> , contains the row indices for each non-zero element of the matrix A .

	Refer to <i>rows</i> array description in Coordinate Format for more details.
<i>colind</i>	INTEGER. Array of length <i>nnz</i> , contains the column indices for each non-zero element of the matrix <i>A</i> . Refer to <i>columns</i> array description in Coordinate Format for more details.
<i>nnz</i>	INTEGER. Specifies the number of non-zero element of the matrix <i>A</i> . Refer to <i>nnz</i> description in Coordinate Format for more details.
<i>x</i>	REAL*8. Array, DIMENSION at least <i>m</i> . On entry, the array <i>x</i> must contain the vector <i>x</i> . The elements are accessed with unit increment.
<i>y</i>	REAL*8. Array, DIMENSION at least <i>m</i> . On entry, the array <i>y</i> must contain the vector <i>y</i> . The elements are accessed with unit increment.

Output Parameters

<i>y</i>	Contains solution vector <i>x</i> .

Interfaces

FORTRAN 77:

```

SUBROUTINE mkl_dcoosv(transa, m, alpha, matdescra, val, rowind, colind, nnz, x, y)
  CHARACTER*1  transa
  CHARACTER    matdescra(*)
  INTEGER      m, nnz
  INTEGER      rowind(*), colind(*)
  REAL*8       alpha
  REAL*8       val(*)
  REAL*8       x(*), y(*)

```

C:

```
void mkl_dcoosv(char *transa, int *m, double *alpha, char *matdescra,
               double *val, int *rowind, int *colind, int *nnz, double *x, double *y);
```

mkl_dcsrmm

Computes matrix - matrix product of a sparse matrix stored in the CSR format.

Syntax

Fortran:

```
call mkl_dcsrmm(transa, m, n, k, alpha, matdescra, val, indx, pntreb, pntre,
               b, ldb, beta, c, ldc)
```

C:

```
mkl_dcsrmm(&transa, &m, &n, &k, &alpha, matdescra, val, indx, pntreb, pntre,
           b, &ldb, &beta, c, &ldc);
```

Description

The `mkl_dcsrmm` routine performs a matrix-matrix operation defined as

$$C := \alpha A * B + \beta C$$

or

$$C := \alpha A' * B + \beta C,$$

where:

alpha and *beta* are scalars,

B and *C* are dense matrices, *A* is an *m*-by-*k* sparse matrix in compressed sparse row (CSR) format, *A'* is the transpose of *A*.



NOTE. This routine supports CSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then $C := \alpha * A * B + \beta * C$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $C := \alpha * A' * B + \beta * C$,
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>C</i> .
<i>k</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>alpha</i>	REAL*8. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
<i>val</i>	REAL*8. Array containing non-zero elements of the matrix <i>A</i>. For one-based indexing its length is $pntrb(m) - pntrb(1)$. For zero-based indexing its length is $pntrb(m-1) - pntrb(0)$. Refer to <i>values</i> array description in CSR Format for more details.
<i>indx</i>	INTEGER. Array containing the column indices for each non-zero element of the matrix <i>A</i>. Its length is equal to length of the <i>val</i> array. Refer to <i>columns</i> array description in CSR Format for more details.
<i>pntrb</i>	INTEGER. Array of length <i>m</i>. For one-based indexing this array contains row indices, such that $pntrb(i) - pntrb(1) + 1$ is the first index of row <i>i</i> in the arrays <i>val</i> and <i>indx</i>. For zero-based indexing this array contains row indices, such that $pntrb(i) - pntrb(0)$ is the first index of row <i>i</i> in the arrays <i>val</i> and <i>indx</i>. Refer to <i>pointerb</i> array description in CSR Format for more details.

<i>pntrc</i>	INTEGER. Array of length <i>m</i>. For one-based indexing this array contains row indices, such that <i>pntrc</i>(<i>i</i>) - <i>pntrb</i>(1) is the last index of row <i>i</i> in the arrays <i>val</i> and <i>indx</i>. For zero-based indexing this array contains row indices, such that <i>pntrc</i>(<i>i</i>) - <i>pntrb</i>(0) - 1 is the last index of row <i>i</i> in the arrays <i>val</i> and <i>indx</i>. Refer to <i>pointerE</i> array description in CSR Format for more details.
<i>b</i>	REAL*8. Array, DIMENSION (<i>ldb</i>, <i>n</i>) for one-based indexing, and (<i>m</i>, <i>ldb</i>) for zero-based indexing. On entry with <i>transa</i>= 'N' or 'n', the leading <i>k</i>-by-<i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i>, otherwise the leading <i>m</i>-by-<i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i>.
<i>ldb</i>	INTEGER. Specifies the first dimension of <i>b</i> for one-based indexing, and the second dimension of <i>b</i> for zero-based indexing, as declared in the calling (sub)program.
<i>beta</i>	REAL*8. Specifies the scalar <i>beta</i>.
<i>c</i>	REAL*8. Array, DIMENSION (<i>ldc</i>, <i>n</i>) for one-based indexing, and (<i>m</i>, <i>ldc</i>) for zero-based indexing. On entry, the leading <i>m</i>-by-<i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i>, otherwise the leading <i>k</i>-by-<i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i>.
<i>ldc</i>	INTEGER. Specifies the first dimension of <i>c</i> for one-based indexing, and the second dimension of <i>c</i> for zero-based indexing, as declared in the calling (sub)program.

Output Parameters

<i>c</i>	Overwritten by the matrix (<i>alpha</i>*<i>A</i>*<i>B</i> + <i>beta</i>* <i>C</i>) or (<i>alpha</i>*<i>A</i>'*<i>B</i> + <i>beta</i>*<i>C</i>).

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_dcsrmm(transa, m, n, k, alpha, matdescra, val, indx,
    pntrb, pntre, b, ldb, beta, c, ldc)
    CHARACTER*1
    transa
    CHARACTER      matdescra(*)
    INTEGER        m, n, k, ldb, ldc
    INTEGER        indx(*), pntrb(m), pntre(m)
    REAL*8         alpha, beta
    REAL*8         val(*), b(ldb,*), c(ldc,*)
```

C:

```
void mkl_dcsrmm(char *transa, int *m, int *n, int *k, double *alpha,
    char *matdescra, double *val, int *indx, int *pntrb, int *pntre,
    double *b, int *ldb, double *beta, double *c, int *ldc,);
```

mkl_dbsrmm

Computes matrix - matrix product of a sparse matrix stored in the BSR format.

Syntax

Fortran:

```
call mkl_dbsrmm(transa, m, n, k, lb, alpha, matdescra, val, indx, pntrb,
    pntre, b, ldb, beta, c, ldc)
```

C:

```
mkl_dbsrmm(&transa, &m, &n, &k, &lb, &alpha, matdescra, val, indx, pntrb,
    pntre, b, &ldb, &beta, c, &ldc);
```

Description

The `mkl_dbsrmm` routine performs a matrix-matrix operation defined as

$$C := \alpha * A * B + \beta * C$$

or

$$C := \alpha * A' * B + \beta * C,$$

where:

alpha and *beta* are scalars,

B and *C* are dense matrices, *A* is an *m*-by-*k* sparse matrix in block sparse row (BSR) format, *A'* is the transpose of *A*.



NOTE. This routine supports BSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then the matrix-matrix product is computed as $C := \alpha * A * B + \beta * C$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $C := \alpha * A' * B + \beta * C$,
<i>m</i>	INTEGER. Number of block rows of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>C</i> .
<i>k</i>	INTEGER. Number of block columns of the matrix <i>A</i> .
<i>lb</i>	INTEGER. Size of the block in the matrix <i>A</i> .
<i>alpha</i>	REAL*8. Specifies the scalar <i>alpha</i> .

<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
<i>val</i>	REAL*8. Array containing elements of non-zero blocks of the matrix <i>A</i> . Its length is equal to the number of non-zero blocks in the matrix <i>A</i> multiplied by <i>lb*lb</i> . Refer to the <i>values</i> array description in BSR Format for more details.
<i>indx</i>	INTEGER. Array containing the column indices for each non-zero block in the matrix <i>A</i> . Its length is equal to the number of non-zero blocks in the matrix <i>A</i> . Refer to the <i>columns</i> array description in BSR Format for more details.
<i>pntrb</i>	INTEGER. Array of length <i>m</i>. For one-based indexing: this array contains row indices, such that $pntrb(i) - pntrb(1) + 1$ is the first index of block row <i>i</i> in the array <i>indx</i>. For zero-based indexing: this array contains row indices, such that $pntrb(i) - pntrb(0)$ is the first index of block row <i>i</i> in the array <i>indx</i>. Refer to <i>pointerB</i> array description in BSR Format for more details.
<i>pntrb</i>	INTEGER. Array of length <i>m</i>. For one-based indexing this array contains row indices, such that $pntrb(i) - pntrb(1)$ is the last index of block row <i>i</i> in the array <i>indx</i>. For zero-based indexing this array contains row indices, such that $pntrb(i) - pntrb(0) - 1$ is the last index of block row <i>i</i> in the array <i>indx</i>. Refer to <i>pointerE</i> array description in BSR Format for more details.
<i>b</i>	REAL*8. Array, DIMENSION (<i>ldb</i>, <i>n</i>) for one-based indexing, DIMENSION (<i>m</i>, <i>ldb</i>) for zero-based indexing. On entry with <i>transa</i>= 'N' or 'n', the leading <i>n</i>-by-<i>k</i> block part of the array <i>b</i> must contain the matrix <i>B</i>, otherwise the leading <i>m</i>-by-<i>n</i> block part of the array <i>b</i> must contain the matrix <i>B</i>.

<i>ldb</i>	INTEGER. Specifies the first dimension (in blocks) of <i>b</i> as declared in the calling (sub)program.
<i>beta</i>	REAL*8. Specifies the scalar <i>beta</i> .
<i>c</i>	REAL*8. Array, DIMENSION (<i>ldc</i> , <i>n</i>) for one-based indexing, DIMENSION (<i>k</i> , <i>ldc</i>) for zero-based indexing. On entry, the leading <i>m</i> -by- <i>n</i> block part of the array <i>c</i> must contain the matrix <i>C</i> , otherwise the leading <i>n</i> -by- <i>k</i> block part of the array <i>c</i> must contain the matrix <i>C</i> .
<i>ldc</i>	INTEGER. Specifies the first dimension (in blocks) of <i>c</i> as declared in the calling (sub)program.

Output Parameters

<i>c</i>	Overwritten by the matrix ($\alpha * A * B + \beta * C$) or ($\alpha * A' * B + \beta * C$).

Interfaces

FORTRAN 77:

```

SUBROUTINE mkl_dbsrmm(transa, m, n, k, lb, alpha, matdescra, val,
    indx, pntreb, pntre, b, ldb, beta, c, ldc)
    CHARACTER*1    transa
    CHARACTER       matdescra(*)
    INTEGER         m, n, k, ld, ldb, ldc
    INTEGER         indx(*), pntreb(m), pntre(m)
    REAL*8          alpha, beta
    REAL*8          val(*), b(ldb,*), c(ldc,*)

```

C:

```

void mkl_dbsrmm(char *transa, int *m, int *n, int *k, int *lb,
    double *alpha, char *matdescra, double *val, int *indx, int *pntreb,
    int *pntre, double *b, int *ldb, double *beta, double *c, int *ldc,);

```

mkl_dcscmm

Computes matrix-matrix product of a sparse matrix stored in the CSC format.

Syntax

Fortran:

```
call mkl_dcscmm(transa, m, n, k, alpha, matdescra, val, indx, pntrb, pntre,
b, ldb, beta, c, ldc)
```

C:

```
mkl_dcscmm(&transa, &m, &n, &k, &alpha, matdescra, val, indx, pntrb, pntre,
b, &ldb, &beta, c, &ldc);
```

Description

The `mkl_dcscmm` routine performs a matrix-matrix operation defined as

$$C := \alpha A * B + \beta C$$

or

$$C := \alpha A' * B + \beta C,$$

where:

alpha and *beta* are scalars,

B and *C* are dense matrices, *A* is an *m*-by-*k* sparse matrix in compressed sparse column (CSC) format, *A'* is the transpose of *A*.



NOTE. This routine supports CSC format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

transa CHARACTER*1. Specifies the operation.
If *transa* = 'N' or 'n', then $C := \alpha A * B + \beta C$

	If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $C := \alpha * A' * B + \beta * C$,
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>C</i> .
<i>k</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>alpha</i>	REAL*8. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6.
<i>val</i>	REAL*8. Array containing non-zero elements of the matrix <i>A</i>. For one-based indexing its length is $pntre(k) - pntrb(1)$. For zero-based indexing its length is $pntre(m-1) - pntrb(0)$. Refer to <i>values</i> array description in CSC Format for more details.
<i>indx</i>	INTEGER. Array containing the row indices for each non-zero element of the matrix <i>A</i>. Its length is equal to length of the <i>val</i> array. Refer to <i>rows</i> array description in CSC Format for more details.
<i>pntrb</i>	INTEGER. Array of length <i>k</i>. For one-based indexing this array contains column indices, such that $pntrb(i) - pntrb(1) + 1$ is the first index of column <i>i</i> in the arrays <i>val</i> and <i>indx</i>. For zero-based indexing this array contains column indices, such that $pntrb(i) - pntrb(0)$ is the first index of column <i>i</i> in the arrays <i>val</i> and <i>indx</i>. Refer to <i>pointerb</i> array description in CSC Format for more details.
<i>pntre</i>	INTEGER. Array of length <i>k</i>. For one-based indexing this array contains column indices, such that $pntre(i) - pntrb(1)$ is the last index of column <i>i</i> in the arrays <i>val</i> and <i>indx</i>.

For zero-based indexing this array contains column indices, such that $\text{pntrc}(i) - \text{pntrb}(1) - 1$ is the last index of column i in the arrays val and indx .

Refer to *pointerE* array description in [CSC Format](#) for more details.

b	REAL*8. Array, DIMENSION(ldb, n) for one-based indexing, and (m, ldb) for zero-based indexing. On entry with $\text{transa} = 'N'$ or $'n'$, the leading k-by-n part of the array b must contain the matrix B, otherwise the leading m-by-n part of the array b must contain the matrix B.
ldb	INTEGER. Specifies the first dimension of b for one-based indexing, and the second dimension of b for zero-based indexing, as declared in the calling (sub)program.
β	REAL*8. Specifies the scalar β.
c	REAL*8. Array, DIMENSION(ldc, n) for one-based indexing, and (m, ldc) for zero-based indexing. On entry, the leading m-by-n part of the array c must contain the matrix C, otherwise the leading k-by-n part of the array c must contain the matrix C.
ldc	INTEGER. Specifies the first dimension of c for one-based indexing, and the second dimension of c for zero-based indexing, as declared in the calling (sub)program.

Output Parameters

c	Overwritten by the matrix $(\alpha * A * B + \beta * C)$ or $(\alpha * A' * B + \beta * C)$.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_dcscmm(transa, m, n, k, alpha, matdescra, val, indx,
    pntrb, pntre, b, ldb, beta, c, ldc)
    CHARACTER*1    transa
    CHARACTER       matdescra(*)
    INTEGER        m, n, k, ldb, ldc
    INTEGER        indx(*), pntrb(k), pntre(k)
    REAL*8         alpha, beta
    REAL*8         val(*), b(ldb,*), c(ldc,*)
```

C:

```
void mkl_dcscmm(char *transa, int *m, int *n, int *k,
    double *alpha, char *matdescra, double *val, int *indx,
    int *pntrb, int *pntre, double *b, int *ldb,
    double *beta, double *c, int *ldc);
```

mkl_dcoomm

Computes matrix-matrix product of a sparse matrix stored in the coordinate format.

Syntax

Fortran:

```
call mkl_dcoomm(transa, m, n, k, alpha, matdescra, val, rowind, colind, nnz,
    b, ldb, beta, c, ldc)
```

C:

```
mkl_dcoomm(&transa, &m, &n, &k, &alpha, matdescra, val, rowind, colind, &nnz,
    b, &ldb, &beta, c, &ldc);
```

Description

The `mk1_dcoomm` routine performs a matrix-matrix operation defined as

$$C := \alpha * A * B + \beta * C$$

or

$$C := \alpha * A' * B + \beta * C,$$

where:

alpha and *beta* are scalars,

B and *C* are dense matrices, *A* is an *m*-by-*k* sparse matrix in the coordinate format, *A'* is the transpose of *A*.



NOTE. This routine supports coordinate format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then $C := \alpha * A * B + \beta * C$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $C := \alpha * A' * B + \beta * C$,
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>C</i> .
<i>k</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>alpha</i>	REAL*8. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
<i>val</i>	REAL*8. Array of length <i>nnz</i> , contains non-zero elements of the matrix <i>A</i> in the arbitrary order. Refer to <i>values</i> array description in Coordinate Format for more details.

<i>rowind</i>	INTEGER. Array of length <i>nnz</i>, contains the row indices for each non-zero element of the matrix <i>A</i>. Refer to <i>rows</i> array description in Coordinate Format for more details.
<i>colind</i>	INTEGER. Array of length <i>nnz</i>, contains the column indices for each non-zero element of the matrix <i>A</i>. Refer to <i>columns</i> array description in Coordinate Format for more details.
<i>nnz</i>	INTEGER. Specifies the number of non-zero element of the matrix <i>A</i>. Refer to <i>nnz</i> description in Coordinate Format for more details.
<i>b</i>	REAL*8. Array, DIMENSION (<i>ldb</i>, <i>n</i>) for one-based indexing, and (<i>m</i>, <i>ldb</i>) for zero-based indexing. On entry with <i>transa</i> = 'N' or 'n', the leading <i>k</i>-by-<i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i>, otherwise the leading <i>m</i>-by-<i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i>.
<i>ldb</i>	INTEGER. Specifies the first dimension of <i>b</i> for one-based indexing, and the second dimension of <i>b</i> for zero-based indexing, as declared in the calling (sub)program.
<i>beta</i>	REAL*8. Specifies the scalar <i>beta</i>.
<i>c</i>	REAL*8. Array, DIMENSION (<i>ldc</i>, <i>n</i>) for one-based indexing, and (<i>m</i>, <i>ldc</i>) for zero-based indexing. On entry, the leading <i>m</i>-by-<i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i>, otherwise the leading <i>k</i>-by-<i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i>.
<i>ldc</i>	INTEGER. Specifies the first dimension of <i>c</i> for one-based indexing, and the second dimension of <i>c</i> for zero-based indexing, as declared in the calling (sub)program.

Output Parameters

<i>c</i>	Overwritten by the matrix ($\alpha * A * B + \beta * C$) or ($\alpha * A' * B + \beta * C$).

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_dcoomm(transa, m, n, k, alpha, matdescra, val,
    rowind, colind, nnz, b, ldb, beta, c, ldc)
    CHARACTER*1    transa
    CHARACTER       matdescra(*)
    INTEGER         m, n, k, ldb, ldc, nnz
    INTEGER         rowind(*), colind(*)
    REAL*8          alpha, beta
    REAL*8          val(*), b(ldb,*), c(ldc,*)
```

C:

```
void mkl_dcoomm(char *transa, int *m, int *n, int *k,
    double *alpha,
    char *matdescra, double *val, int *rowind, int *colind, int *nnz,
    double *b, int *ldb, double *beta, double *c, int *ldc);
```

mkl_dcsrsm

Solves a system of linear matrix equations for a sparse matrix in the CSR format.

Syntax

Fortran:

```
call mkl_dcsrsm(transa, m, n, alpha, matdescra, val, indx, pntrb, pntre, b,
    ldb, c, ldc)
```

C:

```
mkl_dcsrsm(&transa, &m, &n, &alpha, matdescra, val, indx, pntrb, pntre, b,
    &ldb, c, &ldc);
```

Description

The `mkl_dcsrsm` routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the CSR format:

$$C := \alpha * \text{inv}(A) * B$$

or

$$C := \alpha * \text{inv}(A') * B,$$

where:

α is scalar, B and C are dense matrices, A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .



NOTE. This routine supports CSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>transa</i>	CHARACTER*1. Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then $C := \alpha * \text{inv}(A) * B$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $C := \alpha * \text{inv}(A') * B,$
<i>m</i>	INTEGER. Number of columns of the matrix A .
<i>n</i>	INTEGER. Number of columns of the matrix C .
<i>alpha</i>	REAL*8. Specifies the scalar α .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
<i>val</i>	REAL*8. Array containing non-zero elements of the matrix A . For one-based indexing its length is $\text{pntrc}(m) - \text{pntrb}(1)$. For zero-based indexing its length is $\text{pntrc}(m-1) - \text{pntrb}(0)$.

Refer to *values* array description in [CSR Format](#) for more details.

indx INTEGER. Array containing the column indices for each non-zero element of the matrix *A*. Its length is equal to length of the *val* array.

Refer to *columns* array description in [CSR Format](#) for more details.



NOTE. Column indices must be sorted in increasing order for each row.

pntrb INTEGER. Array of length *m*.

For one-based indexing this array contains row indices, such that $pntrb(i) - pntrb(1) + 1$ is the first index of row *i* in the arrays *val* and *indx*.

For zero-based indexing this array contains row indices, such that $pntrb(i) - pntrb(0)$ is the first index of row *i* in the arrays *val* and *indx*. Refer to *pointerb* array description in [CSR Format](#) for more details.

pntrr INTEGER. Array of length *m*.

For one-based indexing this array contains row indices, such that $pntrr(i) - pntrb(1)$ is the last index of row *i* in the arrays *val* and *indx*.

For zero-based indexing this array contains row indices, such that $pntrr(i) - pntrb(0) - 1$ is the last index of row *i* in the arrays *val* and *indx*. Refer to *pointerE* array description in [CSR Format](#) for more details.

b REAL*8.

Array, DIMENSION (*ldb*, *n*) for one-based indexing, and (*m*, *ldb*) for zero-based indexing.

On entry the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.

ldb INTEGER. Specifies the first dimension of *b* for one-based indexing, and the second dimension of *b* for zero-based indexing, as declared in the calling (sub)program.

ldc INTEGER. Specifies the first dimension of *c* for one-based indexing, and the second dimension of *c* for zero-based indexing, as declared in the calling (sub)program.

Output Parameters

c REAL*8.
 Array, DIMENSION (*ldc*, *n*) for one-based indexing, and (*m*, *ldc*) for zero-based indexing.
 The leading *m*-by-*n* part of the array *c* contains the output matrix *C*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_dcsrsm(transa, m, n, alpha, matdescra, val, indx,
  pntreb, pntre, b, ldb, c, ldc)
  CHARACTER*1  transa
  CHARACTER    matdescra(*)
  INTEGER      m, n, ldb, ldc
  INTEGER      indx(*), pntreb(m), pntre(m)
  REAL*8       alpha
  REAL*8       val(*), b(ldb,*), c(ldc,*)
```

C:

```
void mkl_dcsrsm(char *transa, int *m, int *n, double *alpha,
  char *matdescra, double *val, int *indx, int *pntreb,
  int *pntre, double *b, int *ldb, double *c, int *ldc);
```


mkl_dcscsm

Solves a system of linear matrix equations for a sparse matrix in the CSC format.

Syntax

Fortran:

```
call mkl_dcscsm(transa, m, n, alpha, matdescra, val, indx, pntrb, pntre, b,
ldb, c, ldc)
```

C:

```
mkl_dcscsm(&transa, &m, &n, &alpha, matdescra, val, indx, pntrb, pntre, b,
&ldb, c, &ldc);
```

Description

The `mkl_dcscsm` routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the CSC format:

$$C := \alpha * \text{inv}(A) * B$$

or

$$C := \alpha * \text{inv}(A') * B,$$

where:

α is scalar, B and C are dense matrices, A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .



NOTE. This routine supports CSC format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

transa CHARACTER*1. Specifies the system of equations.
If *transa* = 'N' or 'n', then $C := \alpha * \text{inv}(A) * B$

	If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $C := \alpha * \text{inv}(A') * B$,
<i>m</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>C</i> .
<i>alpha</i>	REAL*8. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
<i>val</i>	REAL*8. Array containing non-zero elements of the matrix <i>A</i> . For one-based indexing its length is $\text{pntrb}(k) - \text{pntrb}(1)$. For zero-based indexing its length is $\text{pntrb}(m-1) - \text{pntrb}(0)$. Refer to <i>values</i> array description in CSC Format for more details.
<i>indx</i>	INTEGER. Array containing the row indices for each non-zero element of the matrix <i>A</i> . Its length is equal to length of the <i>val</i> array. Refer to <i>rows</i> array description in CSC Format for more details.



NOTE. Row indices must be sorted in increasing order for each column.

<i>pntrb</i>	INTEGER. Array of length <i>m</i> . For one-based indexing this array contains column indices, such that $\text{pntrb}(i) - \text{pntrb}(1) + 1$ is the first index of column <i>i</i> in the arrays <i>val</i> and <i>indx</i> . For zero-based indexing this array contains column indices, such that $\text{pntrb}(i) - \text{pntrb}(0)$ is the first index of column <i>i</i> in the arrays <i>val</i> and <i>indx</i> . Refer to <i>pointerb</i> array description in CSC Format for more details.
<i>pntrb</i>	INTEGER. Array of length <i>m</i> .

For one-based indexing this array contains column indices, such that $\text{pntrc}(i) - \text{pntrb}(1)$ is the last index of column i in the arrays val and indx .

For zero-based indexing this array contains column indices, such that $\text{pntrc}(i) - \text{pntrb}(1) - 1$ is the last index of column i in the arrays val and indx .

Refer to *pointerE* array description in [CSC Format](#) for more details.

b

REAL*8.

Array, DIMENSION (ldb, n) for one-based indexing, and (m, ldb) for zero-based indexing.

On entry the leading m -by- n part of the array b must contain the matrix B .

ldb

INTEGER. Specifies the first dimension of b for one-based indexing, and the second dimension of b for zero-based indexing, as declared in the calling (sub)program.

ldc

INTEGER. Specifies the first dimension of c for one-based indexing, and the second dimension of c for zero-based indexing, as declared in the calling (sub)program.

Output Parameters

c

REAL*8.

Array, DIMENSION (ldc, n) for one-based indexing, and (m, ldc) for zero-based indexing.

The leading m -by- n part of the array c contains the output matrix C .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_dcscsm(transa, m, n, alpha, matdescra, val, indx,  
  pntrb, pntre, b, ldb, c, ldc)  
  
  CHARACTER*1  transa  
  
  CHARACTER    matdescra(*)  
  
  INTEGER      m, n, ldb, ldc  
  
  INTEGER      indx(*), pntrb(m), pntre(m)  
  
  REAL*8       alpha  
  
  REAL*8       val(*), b(ldb,*), c(ldc,*)
```

C:

```
void mkl_dcscsm(char *transa, int *m, int *n, double *alpha,  
  char *matdescra, double *val, int *indx, int *pntrb,  
  int *pntre, double *b, int *ldb, double *c, int *ldc);
```

mkl_dcoosm

*Solves a system of linear matrix equations for a
sparse matrix in the coordinate format.*

Syntax

Fortran:

```
call mkl_dcoosm(transa, m, n, alpha, matdescra, val, rowind, colind, nnz, b,  
  ldb, c, ldc)
```

C:

```
mkl_dcoosm(&transa, &m, &n, &alpha, matdescra, val, rowind, colind, &nnz, b,  
  &ldb, c, &ldc);
```

Description

The `mkl_dcoosm` routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the coordinate format:

`C := alpha*inv(A)*B`

or

`C := alpha*inv(A')*B,`

where:

alpha is scalar, *B* and *C* are dense matrices, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.



NOTE. This routine supports coordinate format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

<i>transa</i>	CHARACTER*1. Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then the matrix-matrix product is computed as <code>C := alpha*inv(A)*B</code> If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as <code>C := alpha*inv(A')*B,</code>
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>C</i> .
<i>alpha</i>	REAL*8. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
<i>val</i>	REAL*8. Array of length <i>nnz</i> , contains non-zero elements of the matrix <i>A</i> in the arbitrary order. Refer to <i>values</i> array description in Coordinate Format for more details.

<i>rowind</i>	INTEGER. Array of length <i>nnz</i>, contains the row indices for each non-zero element of the matrix <i>A</i>. Refer to <i>rows</i> array description in Coordinate Format for more details.
<i>colind</i>	INTEGER. Array of length <i>nnz</i>, contains the column indices for each non-zero element of the matrix <i>A</i>. Refer to <i>columns</i> array description in Coordinate Format for more details.
<i>nnz</i>	INTEGER. Specifies the number of non-zero element of the matrix <i>A</i>. Refer to <i>nnz</i> description in Coordinate Format for more details.
<i>b</i>	REAL*8. Array, DIMENSION (<i>ldb</i>, <i>n</i>) for one-based indexing, and (<i>m</i>, <i>ldb</i>) for zero-based indexing. Before entry the leading <i>m</i>-by-<i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i>.
<i>ldb</i>	INTEGER. Specifies the first dimension of <i>b</i> for one-based indexing, and the second dimension of <i>b</i> for zero-based indexing, as declared in the calling (sub)program.
<i>ldc</i>	INTEGER. Specifies the first dimension of <i>c</i> for one-based indexing, and the second dimension of <i>c</i> for zero-based indexing, as declared in the calling (sub)program.

Output Parameters

<i>c</i>	REAL*8. Array, DIMENSION (<i>ldc</i>, <i>n</i>) for one-based indexing, and (<i>m</i>, <i>ldc</i>) for zero-based indexing. The leading <i>m</i>-by-<i>n</i> part of the array <i>c</i> contains the output matrix <i>C</i>.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_dcoosm(transa, m, n, alpha, matdescra, val, rowind, colind, nnz, b, ldb, c,
  ldc)

  CHARACTER*1    transa

  CHARACTER      matdescra(*)

  INTEGER        m, n, ldb, ldc, nnz

  INTEGER        rowind(*), colind(*)

  REAL*8         alpha

  REAL*8         val(*), b(ldb,*), c(ldc,*)
```

C:

```
void mkl_dcoosm(char *transa, int *m, int *n, double *alpha, char *matdescra,
  double *val, int *rowind, int *colind, int *nnz, double *b, int *ldb, double *c, int *ldc);
```

mkl_dbsrsm

Solves a system of linear matrix equations for a sparse matrix in the BSR format.

Syntax

Fortran:

```
call mkl_dcsrsm(transa, m, n, lb, alpha, matdescra, val, indx, pntreb, pntre,
  b, ldb, c, ldc)
```

C:

```
mkl_dcsrsm(&transa, &m, &n, &lb, &alpha, matdescra, val, indx, pntreb, pntre,
  b, &ldb, c, &ldc);
```

Description

The `mkl_dbsrsm` routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the BSR format:

$$C := \alpha \cdot \text{inv}(A) \cdot B$$

or

$$C := \alpha * \text{inv}(A') * B,$$

where:

α is scalar, B and C are dense matrices, A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .



NOTE. This routine supports BSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then the matrix-matrix product is computed as $C := \alpha * \text{inv}(A) * B$. If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $C := \alpha * \text{inv}(A') * B$.
<i>m</i>	INTEGER. Number of block columns of the matrix A .
<i>n</i>	INTEGER. Number of columns of the matrix C .
<i>lb</i>	INTEGER. Size of the block in the matrix A .
<i>alpha</i>	REAL*8. Specifies the scalar α .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
<i>val</i>	REAL*8. Array containing elements of non-zero blocks of the matrix A . Its length is equal to the <i>ABAB</i> number <i>ABAB</i> of non-zero blocks in the matrix A multiplied by $lb * lb$. Refer to the <i>values</i> array description in BSR Format for more details.

<i>indx</i>	INTEGER. Array containing the column indices for each non-zero block in the matrix <i>A</i> . Its length is equal to the number of non-zero blocks in the matrix <i>A</i> . Refer to the <i>columns</i> array description in BSR Format for more details.
<i>pntrb</i>	INTEGER. Array of length <i>m</i>. For one-based indexing: this array contains row indices, such that $pntrb(i) - pntrb(1) + 1$ is the first index of block row <i>i</i> in the array <i>indx</i>. For zero-based indexing: this array contains row indices, such that $pntrb(i) - pntrb(0)$ is the first index of block row <i>i</i> in the array <i>indx</i>. Refer to <i>pointerB</i> array description in BSR Format for more details.
<i>pntre</i>	INTEGER. Array of length <i>m</i>. For one-based indexing this array contains row indices, such that $pntre(i) - pntrb(1)$ is the last index of block row <i>i</i> in the array <i>indx</i>. For zero-based indexing this array contains row indices, such that $pntre(i) - pntrb(0) - 1$ is the last index of block row <i>i</i> in the array <i>indx</i>. Refer to <i>pointerE</i> array description in BSR Format for more details.
<i>b</i>	REAL*8. Array, DIMENSION (<i>ldb</i>, <i>n</i>) for one-based indexing, DIMENSION (<i>m</i>, <i>ldb</i>) for zero-based indexing. On entry the leading <i>m</i>-by-<i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i>.
<i>ldb</i>	INTEGER. Specifies the first dimension (in blocks) of <i>b</i> as declared in the calling (sub)program.
<i>ldc</i>	INTEGER. Specifies the first dimension (in blocks) of <i>c</i> as declared in the calling (sub)program.

Output Parameters

<i>c</i>	REAL*8. Array, DIMENSION (<i>ldc</i>, <i>n</i>) for one-based indexing, DIMENSION (<i>m</i>, <i>ldc</i>) for zero-based indexing.

The leading m -by- n part of the array c contains the output matrix C .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_dbsrsm(transa, m, n, lb, alpha, matdescra, val, indx, pntrb, pntre, b, ldb,
c, ldc)

  CHARACTER*1  transa
  CHARACTER    matdescra(*)
  INTEGER      m, n, lb, ldb, ldc
  INTEGER      indx(*), pntrb(m), pntre(m)
  REAL*8       alpha
  REAL*8       val(*), b(ldb,*), c(ldc,*)
```

C:

```
void mkl_dbsrsm(char *transa, int *m, int *n, int *lb, double *alpha, char *matdescra,
double *val, int *indx, int *pntrb, int *pntre, double *b, int *ldb, double *c, int *ldc);
```

mkl_ddiamv

Computes matrix - vector product for a sparse matrix in the diagonal format with one-based indexing.

Syntax

Fortran:

```
call mkl_ddiamv(transa, m, k, alpha, matdescra, val, lval, idiag, ndiag, x,
beta, y)
```

C:

```
mkl_ddiamv(&transa, &m, &k, &alpha, matdescra, val, &lval, idiag, &ndiag, x,
&beta, y);
```

Description

The `mkl_ddiamv` routine performs a matrix-vector operation defined as

$$y := \alpha * A * x + \beta * y$$

or

$$y := \alpha * A' * x + \beta * y,$$

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* sparse matrix stored in the diagonal format, *A'* is the transpose of *A*.



NOTE. This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then $y := \alpha * A * x + \beta * y$, If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha * A' * x + \beta * y$.
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>k</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>alpha</i>	REAL*8. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
<i>val</i>	REAL*8. Two-dimensional array of size <i>lval</i> by <i>ndiag</i> , contains non-zero diagonals of the matrix <i>A</i> . Refer to <i>values</i> array description in Diagonal Storage Scheme for more details.

<i>lval</i>	INTEGER. Leading dimension of <i>val</i> , $lval \geq m$. Refer to <i>lval</i> description in Diagonal Storage Scheme for more details.
<i>idiag</i>	INTEGER. Array of length <i>ndiag</i> , contains the distances between main diagonal and each non-zero diagonals in the matrix <i>A</i> . Refer to <i>distance</i> array description in Diagonal Storage Scheme for more details.
<i>ndiag</i>	INTEGER. Specifies the number of non-zero diagonals of the matrix <i>A</i> .
<i>x</i>	REAL*8. Array, DIMENSION at least <i>k</i> if <i>transa</i> = 'N' or 'n', and at least <i>m</i> otherwise. On entry, the array <i>x</i> must contain the vector <i>x</i> .
<i>beta</i>	REAL*8. Specifies the scalar <i>beta</i> .
<i>y</i>	REAL*8. Array, DIMENSION at least <i>m</i> if <i>transa</i> = 'N' or 'n', and at least <i>k</i> otherwise. On entry, the array <i>y</i> must contain the vector <i>y</i> .

Output Parameters

<i>y</i>	Overwritten by the updated vector <i>y</i> .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_ddiamv(transa, m, k, alpha, matdescra, val, lval, idiag,
ndiag, x, beta, y)
  CHARACTER*1  transa
  CHARACTER    matdescra(*)
  INTEGER      m, k, lval, ndiag
  INTEGER      idiag(*)
  REAL*8       alpha, beta
  REAL*8       val(lval,*), x(*), y(*)
```

C:

```
void mkl_ddiamv(char *transa, int *m, int *k, double *alpha,
char *matdescra, double *val, int *lval, int *idiag,
int *ndiag, double *x, double *beta, double *y);
```

mkl_dskymv

Computes matrix - vector product for a sparse matrix in the skyline storage format with one-based indexing.

Syntax

Fortran:

```
call mkl_dskymv(transa, m, k, alpha, matdescra, val, pntr, x, beta, y)
```

C:

```
mkl_dskymv(&transa, &m, &k, &alpha, matdescra, val, pntr, x, &beta, y);
```

Description

The `mkl_dskymv` routine performs a matrix-vector operation defined as

$y := \alpha A x + \beta y$

or

$y := \alpha * A' * x + \beta * y,$

where:

α and β are scalars,

x and y are vectors,

A is an m -by- k sparse matrix stored using the skyline storage scheme, A' is the transpose of A .



NOTE. This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then $y := \alpha * A * x + \beta * y$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha * A' * x + \beta * y,$
<i>m</i>	INTEGER. Number of rows of the matrix A .
<i>k</i>	INTEGER. Number of columns of the matrix A .
<i>alpha</i>	REAL*8. Specifies the scalar α .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .



NOTE. General matrices (*matdescra* (1)='G') is not supported.

<i>val</i>	REAL*8. Array containing the set of elements of the matrix A in the skyline profile form. If <i>matdescra</i> (2)= 'L', then <i>val</i> contains elements from the low triangle of the matrix A .

	If <code>matdescrsa(2) = 'U'</code>, then <code>val</code> contains elements from the upper triangle of the matrix <code>A</code>. Refer to <code>values</code> array description in Skyline Storage Scheme for more details.
<code>pntr</code>	INTEGER. Array of length $(m+m)$ for lower triangle, and $(k+k)$ for upper triangle. It contains the indices specifying in the <code>val</code> the positions of the first element in each row (column) of the matrix <code>A</code>. Refer to <code>pointers</code> array description in Skyline Storage Scheme for more details.
<code>x</code>	REAL*8. Array, DIMENSION at least k if <code>transa = 'N'</code> or <code>'n'</code> and at least m otherwise. On entry, the array <code>x</code> must contain the vector <code>x</code>.
<code>beta</code>	REAL*8. Specifies the scalar <code>beta</code>.
<code>y</code>	REAL*8. Array, DIMENSION at least m if <code>transa = 'N'</code> or <code>'n'</code> and at least k otherwise. On entry, the array <code>y</code> must contain the vector <code>y</code>.

Output Parameters

<code>y</code>	Overwritten by the updated vector <code>y</code> .

Interfaces

FORTRAN 77:

```

SUBROUTINE mkl_dskymv(transa, m, k, alpha, matdescra, val, pntr, x, beta, y)
  CHARACTER*1  transa
  CHARACTER    matdescra(*)
  INTEGER      m, k
  INTEGER      pntr(*)
  REAL*8       alpha, beta
  REAL*8       val(*), x(*), y(*)

```

C:

```
void mkl_dskymv (char *transa, int *m, int *k, double *alpha, char *matdescra,  
                double *val, int *pntr, double *x, double *beta, double *y);
```

mkl_ddiasv

Solves a system of linear equations for a sparse matrix in the diagonal format with one-based indexing .

Syntax

Fortran:

```
call mkl_ddiasv(transa, m, alpha, matdescra, val, lval, iddiag, ndiag, x, y)
```

C:

```
mkl_ddiasv(&transa, &m, &alpha, matdescra, val, &lval, iddiag, &ndiag, x, y);
```

Description

The `mkl_ddiasv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the diagonal format:

$y := \alpha * \text{inv}(A) * x$

or

$y := \alpha * \text{inv}(A') * x,$

where:

α is scalar, x and y are vectors, A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .



NOTE. This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>transa</i>	CHARACTER*1. Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then $y := \alpha * \text{inv}(A) * x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha * \text{inv}(A') * x$,
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>alpha</i>	REAL*8. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
<i>val</i>	REAL*8. Two-dimensional array of size <i>lval</i> by <i>ndiag</i> , contains non-zero diagonals of the matrix <i>A</i> . Refer to <i>values</i> array description in Diagonal Storage Scheme for more details.
<i>lval</i>	INTEGER. Leading dimension of <i>val</i> , $lval \geq m$. Refer to <i>lval</i> description in Diagonal Storage Scheme for more details.
<i>idiag</i>	INTEGER. Array of length <i>ndiag</i> , contains the distances between main diagonal and each non-zero diagonals in the matrix <i>A</i> .



NOTE. All elements of this array must be sorted in increasing order.

Refer to *distance* array description in [Diagonal Storage Scheme](#) for more details.

<i>ndiag</i>	INTEGER. Specifies the number of non-zero diagonals of the matrix <i>A</i> .
<i>x</i>	REAL*8. Array, DIMENSION at least <i>m</i> . On entry, the array <i>x</i> must contain the vector <i>x</i> . The elements are accessed with unit increment.
<i>y</i>	REAL*8. Array, DIMENSION at least <i>m</i> . On entry, the array <i>y</i> must contain the vector <i>y</i> . The elements are accessed with unit increment.

Output Parameters

y Contains solution vector x .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_ddiasv(transa, m, alpha, matdescra, val, lval, idiag, ndiag, x, y)
    CHARACTER*1    transa
    CHARACTER       matdescra(*)
    INTEGER         m, lval, ndiag
    INTEGER         idiag(*)
    REAL*8          alpha
    REAL*8          val(lval,*), x(*), y(*)
```

C:

```
void mkl_ddiasv(char *transa, int *m, double *alpha, char *matdescra,
    double *val, int *lval, int *idiag, int *ndiag, double *x, double *y);
```

mkl_dskysv

Solves a system of linear equations for a sparse matrix in the skyline format with one-based indexing .

Syntax

Fortran:

```
call mkl_dskysv(transa, m, alpha, matdescra, val, pntr, x, y)
```

C:

```
mkl_dskysv(&transa, &m, &alpha, matdescra, val, pntr, x, y);
```

Description

The `mkl_dskysv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the skyline storage format:

`y := alpha*inv(A)*x`

or

`y := alpha*inv(A')*x,`

where:

alpha is scalar, *x* and *y* are vectors, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.



NOTE. This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>transa</i>	CHARACTER*1. Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then <code>y := alpha*inv(A)*x</code> If <i>transa</i> = 'T' or 't' or 'C' or 'c', then <code>y := alpha*inv(A')* x,</code>
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>alpha</i>	REAL*8. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .



NOTE. General matrices (*matdescra* (1)='G') is not supported.

<i>val</i>	REAL*8. Array containing the set of elements of the matrix <i>A</i> in the skyline profile form.

	If <code>matdescrsa(2) = 'L'</code>, then <code>val</code> contains elements from the low triangle of the matrix <code>A</code>. If <code>matdescrsa(2) = 'U'</code>, then <code>val</code> contains elements from the upper triangle of the matrix <code>A</code>. Refer to <code>values</code> array description in Skyline Storage Scheme for more details.
<code>pntr</code>	INTEGER. Array of length $(m+m)$ for lower triangle, and $(k+k)$ for upper triangle. It contains the indices specifying in the <code>val</code> the positions of the first element in each row (column) of the matrix <code>A</code>. Refer to <code>pointers</code> array description in Skyline Storage Scheme for more details.
<code>x</code>	REAL*8. Array, DIMENSION at least <code>m</code>. On entry, the array <code>x</code> must contain the vector <code>x</code>. The elements are accessed with unit increment.
<code>y</code>	REAL*8. Array, DIMENSION at least <code>m</code>. On entry, the array <code>y</code> must contain the vector <code>y</code>. The elements are accessed with unit increment.

Output Parameters

<code>y</code>	Contains solution vector <code>x</code> .

Interfaces

FORTRAN 77:

```

SUBROUTINE mkl_dskysv(transa, m, alpha, matdescra, val, pntr, x, y)

  CHARACTER*1    transa
  CHARACTER      matdescra(*)
  INTEGER        m
  INTEGER        pntr(*)
  REAL*8         alpha
  REAL*8         val(*), x(*), y(*)

```

C:

```
void mkl_dskysv(char *transa, int *m, double *alpha, char *matdescra,
               double *val, int *pntr, double *x, double *y);
```

mkl_ddiamm

Computes matrix-matrix product of a sparse matrix stored in the diagonal format with one-based indexing.

Syntax**Fortran:**

```
call mkl_ddiamm(transa, m, n, k, alpha, matdescra, val, lval, idiag, ndiag,
               b, ldb, beta, c, ldc)
```

C:

```
mkl_ddiamm(&transa, &m, &n, &k, &alpha, matdescra, val, &lval, idiag, &ndiag,
           b, &ldb, &beta, c, &ldc);
```

Description

The `mkl_ddiamm` routine performs a matrix-matrix operation defined as

$$C := \alpha A * B + \beta A * C$$

or

$$C := \alpha A' * B + \beta A' * C,$$

where:

alpha and *beta* are scalars,

B and *C* are dense matrices, *A* is an *m*-by-*k* sparse matrix in the diagonal format, *A'* is the transpose of *A*.



NOTE. This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then $C := \alpha * A * B + \beta * C$, If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $C := \alpha * A' * B + \beta * C$.
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>C</i> .
<i>k</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>alpha</i>	REAL*8. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
<i>val</i>	REAL*8. Two-dimensional array of size <i>lval</i> by <i>ndiag</i> , contains non-zero diagonals of the matrix <i>A</i> . Refer to <i>values</i> array description in Diagonal Storage Scheme for more details.
<i>lval</i>	INTEGER. Leading dimension of <i>val</i> , $lval \geq m$. Refer to <i>lval</i> description in Diagonal Storage Scheme for more details.
<i>idiag</i>	INTEGER. Array of length <i>ndiag</i> , contains the distances between main diagonal and each non-zero diagonals in the matrix <i>A</i> . Refer to <i>distance</i> array description in Diagonal Storage Scheme for more details.
<i>ndiag</i>	INTEGER. Specifies the number of non-zero diagonals of the matrix <i>A</i> .
<i>b</i>	REAL*8. Array, DIMENSION (<i>ldb</i> , <i>n</i>). On entry with <i>transa</i> = 'N' or 'n', the leading <i>k</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i> , otherwise the leading <i>m</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i> .

<i>ldb</i>	INTEGER. Specifies the first dimension of <i>b</i> as declared in the calling (sub)program.
<i>beta</i>	REAL*8. Specifies the scalar <i>beta</i> .
<i>c</i>	REAL*8. Array, DIMENSION (<i>ldc</i> , <i>n</i>). On entry, the leading <i>m</i> -by- <i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i> , otherwise the leading <i>k</i> -by- <i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i> .
<i>ldc</i>	INTEGER. Specifies the first dimension of <i>c</i> as declared in the calling (sub)program.

Output Parameters

<i>c</i>	Overwritten by the matrix ($\alpha * A * B + \beta * C$) or ($\alpha * A' * B + \beta * C$).

Interfaces

FORTRAN 77:

```

SUBROUTINE mkl_ddiamm(transa, m, n, k, alpha, matdescra, val, lval,
    idiag, ndiag, b, ldb, beta, c, ldc)
    CHARACTER*1    transa
    CHARACTER       matdescra(*)
    INTEGER         m, n, k, ldb, ldc, lval, ndiag
    INTEGER         idiag(*)
    REAL*8          alpha, beta
    REAL*8          val(lval,*), b(ldb,*), c(ldc,*)

```

C:

```

void mkl_ddiamm(char *transa, int *m, int *n, int *k, double *alpha,
    char *matdescra, double *val, int *lval, int *idiag, int *ndiag,
    double *b, int *ldb, double *beta, double *c, int *ldc);

```

mkl_dskymm

Computes matrix-matrix product of a sparse matrix stored using the skyline storage scheme with one-based indexing .

Syntax

Fortran:

```
call mkl_dskymm(transa, m, n, k, alpha, matdescra, val, pntr, b, ldb, beta,
c, ldc)
```

C:

```
mkl_dskymm(&transa, &m, &n, &k, &alpha, matdescra, val, pntr, b, &ldb, &beta,
c, &ldc);
```

Description

The `mkl_dskymm` routine performs a matrix-matrix operation defined as

$$C := \alpha * A * B + \beta * C$$

or

$$C := \alpha * A' * B + \beta * C,$$

where:

alpha and *beta* are scalars,

B and *C* are dense matrices, *A* is an *m*-by-*k* sparse matrix in the skyline storage format, *A'* is the transpose of *A*.



NOTE. This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then $C := \alpha * A * B + \beta * C$,

If *transa* = 'T' or 't' or 'C' or 'c', then $C := \alpha * A' * B + \beta * C$,

m INTEGER. Number of rows of the matrix *A*.

n INTEGER. Number of columns of the matrix *C*.

k INTEGER. Number of columns of the matrix *A*.

alpha REAL*8. Specifies the scalar *alpha*.

matdescra CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the [Table 2-6](#).



NOTE. General matrices (*matdescra*(1)='G') is not supported.

val REAL*8. Array containing the set of elements of the matrix *A* in the skyline profile form.
 If *matdescrsa*(2) = 'L', then *val* contains elements from the low triangle of the matrix *A*.
 If *matdescrsa*(2) = 'U', then *val* contains elements from the upper triangle of the matrix *A*.
 Refer to *values* array description in [Skyline Storage Scheme](#) for more details.

pntr INTEGER. Array of length (*m*+*m*) for lower triangle, and (*k*+*k*) for upper triangle.
 It contains the indices specifying in the *val* the positions of the first element in each row (column) of the matrix *A*. Refer to *pointers* array description in [Skyline Storage Scheme](#) for more details.

b REAL*8.
 Array, DIMENSION (*ldb*, *n*).
 On entry with *transa* = 'N' or 'n', the leading *k*-by-*n* part of the array *b* must contain the matrix *B*, otherwise the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.

ldb INTEGER. Specifies the first dimension of *b* as declared in the calling (sub)program.

<i>beta</i>	REAL*8. Specifies the scalar <i>beta</i> .
<i>c</i>	REAL*8. Array, DIMENSION (<i>ldc</i> , <i>n</i>). On entry, the leading <i>m</i> -by- <i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i> , otherwise the leading <i>k</i> -by- <i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i> .
<i>ldc</i>	INTEGER. Specifies the first dimension of <i>c</i> as declared in the calling (sub)program.

Output Parameters

<i>c</i>	Overwritten by the matrix ($\alpha * A * B + \beta * C$) or ($\alpha * A' * B + \beta * C$).

Interfaces

FORTRAN 77:

```

SUBROUTINE mkl_dskymm(transa, m, n, k, alpha, matdescra, val, pntr, b,
ldb, beta, c, ldc)
  CHARACTER*1  transa
  CHARACTER    matdescra(*)
  INTEGER      m, n, k, ldb, ldc
  INTEGER      pntr(*)
  REAL*8       alpha, beta
  REAL*8       val(*), b(ldb,*), c(ldc,*)

```

C:

```

void mkl_dskymm(char *transa, int *m, int *n, int *k, double *alpha,
char *matdescra, double *val, int *pntr, double *b, int *ldb,
double *beta, double *c, int *ldc);

```

mkl_ddiasm

Solves a system of linear matrix equations for a sparse matrix in the diagonal format with one-based indexing .

Syntax

Fortran:

```
call mkl_ddiasm(transa, m, n, alpha, matdescra, val, lval, idiag, ndiag, b,
ldb, c, ldc)
```

C:

```
mkl_ddiasm(&transa, &m, &n, &alpha, matdescra, val, &lval, idiag, &ndiag, b,
&ldb, c, &ldc);
```

Description

The `mkl_ddiasm` routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the diagonal format:

$$C := \alpha \cdot \text{inv}(A) \cdot B$$

or

$$C := \alpha \cdot \text{inv}(A') \cdot B,$$

where:

α is scalar, B and C are dense matrices, A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .



NOTE. This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

transa CHARACTER*1. Specifies the system of linear equations.
If *transa* = 'N' or 'n', then $C := \alpha \cdot \text{inv}(A) \cdot B$,

	If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $C := \alpha * \text{inv}(A') * B$.
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>C</i> .
<i>alpha</i>	REAL*8. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
<i>val</i>	REAL*8. Two-dimensional array of size <i>lval</i> by <i>ndiag</i> , contains non-zero diagonals of the matrix <i>A</i> . Refer to <i>values</i> array description in Diagonal Storage Scheme for more details.
<i>lval</i>	INTEGER. Leading dimension of <i>val</i> , $lval \geq m$. Refer to <i>lval</i> description in Diagonal Storage Scheme for more details.
<i>idiag</i>	INTEGER. Array of length <i>ndiag</i> , contains the distances between main diagonal and each non-zero diagonals in the matrix <i>A</i> .



NOTE. All elements of this array must be sorted in increasing order.

	Refer to <i>distance</i> array description in Diagonal Storage Scheme for more details.
<i>ndiag</i>	INTEGER. Specifies the number of non-zero diagonals of the matrix <i>A</i> .
<i>b</i>	REAL*8. Array, DIMENSION (<i>ldb</i> , <i>n</i>). On entry the leading <i>m</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i> .
<i>ldb</i>	INTEGER. Specifies the first dimension of <i>b</i> as declared in the calling (sub)program.
<i>ldc</i>	INTEGER. Specifies the first dimension of <i>c</i> as declared in the calling (sub)program.

Output Parameters

c REAL*8.
 Array, DIMENSION (*ldc*, *n*).
 The leading *m*-by-*n* part of the array *c* contains the matrix *C*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_ddiasm(transa, m, n, alpha, matdescra, val, lval, idiag,
ndiag, b, ldb, c, ldc)

  CHARACTER*1  transa
  CHARACTER    matdescra(*)
  INTEGER      m, n, ldb, ldc, lval, ndiag
  INTEGER      idiag(*)
  REAL*8       alpha
  REAL*8       val(lval,*), b(ldb,*), c(ldc,*)
```

C:

```
void mkl_ddiasm(char *transa, int *m, int *n, double *alpha,
char *matdescra, double *val, int *lval, int *idiag, int *ndiag,
double *b, int *ldb, double *c, int *ldc);
```

mkl_dskysm

Solves a system of linear matrix equations for a sparse matrix stored using the skyline storage scheme with one-based indexing .

Syntax

Fortran:

```
call mkl_dskysm(transa, m, n, alpha, matdescra, val, pntr, b, ldb, c, ldc)
```

C:

```
mkl_dskysm(&transa, &m, &n, &alpha, matdescra, val, pntr, b, &ldb, c, &ldc);
```

Description

The `mkl_dskysm` routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the skyline storage format:

$$C := \alpha * \text{inv}(A) * B$$

or

$$C := \alpha * \text{inv}(A') * B,$$

where:

alpha is scalar, *B* and *C* are dense matrices, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.



NOTE. This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section “**Interfaces**” below.

<i>transa</i>	CHARACTER*1. Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then $C := \alpha * \text{inv}(A) * B$, If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $C := \alpha * \text{inv}(A') * B$,
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>C</i> .
<i>alpha</i>	REAL*8. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .



NOTE. General matrices (*matdescra* (1)='G') is not supported.

<i>val</i>	REAL*8. Array containing the set of elements of the matrix <i>A</i> in the skyline profile form. If <i>matdescrsa</i>(2) = 'L', then <i>val</i> contains elements from the low triangle of the matrix <i>A</i>. If <i>matdescrsa</i>(2) = 'U', then <i>val</i> contains elements from the upper triangle of the matrix <i>A</i>. Refer to <i>values</i> array description in Skyline Storage Scheme for more details.
<i>pntr</i>	INTEGER. Array of length $(m+m)$. It contains the indices specifying in the <i>val</i> the positions of the first non-zero element of each <i>i</i>-row (column) of the matrix <i>A</i> such that <i>pointers</i>(<i>i</i>) - <i>pointers</i>(1)+1. Refer to <i>pointers</i> array description in Skyline Storage Scheme for more details.
<i>b</i>	REAL*8. Array, DIMENSION (<i>ldb</i>, <i>n</i>). On entry the leading <i>m</i>-by-<i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i>.
<i>ldb</i>	INTEGER. Specifies the first dimension of <i>b</i> as declared in the calling (sub)program.
<i>ldc</i>	INTEGER. Specifies the first dimension of <i>c</i> as declared in the calling (sub)program.

Output Parameters

<i>c</i>	REAL*8. Array, DIMENSION (<i>ldc</i>, <i>n</i>). The leading <i>m</i>-by-<i>n</i> part of the array <i>c</i> contains the matrix <i>C</i>.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_dskysm(transa, m, n, alpha, matdescra, val, pntr, b, ldb, c, ldc)
  CHARACTER*1  transa
  CHARACTER    matdescra(*)
  INTEGER      m, n, ldb, ldc
  INTEGER      pntr(*)
  REAL*8       alpha
  REAL*8       val(*), b(ldb,*), c(ldc,*)
```

C:

```
void mkl_dskysm(char *transa, int *m, int *n, double *alpha, char *matdescra,
  double *val, int *pntr, double *b, int *ldb, double *c, int *ldc,);
```


LAPACK Routines: Linear Equations

3

This chapter describes the Intel® Math Kernel Library implementation of routines from the LAPACK package that are used for solving systems of linear equations and performing a number of related computational tasks. The library includes LAPACK routines for both real and complex data. Routines are supported for systems of equations with the following types of matrices:

- general
- banded
- symmetric or Hermitian positive-definite (both full and packed storage)
- symmetric or Hermitian positive-definite banded
- symmetric or Hermitian indefinite (both full and packed storage)
- symmetric or Hermitian indefinite banded
- triangular (both full and packed storage)
- triangular banded
- tridiagonal.

For each of the above matrix types, the library includes routines for performing the following computations:

- factoring the matrix (except for triangular matrices)
- equilibrating the matrix
- solving a system of linear equations
- estimating the condition number of a matrix
- refining the solution of linear equations and computing its error bounds
- inverting the matrix.

To solve a particular problem, you can call two or more [computational routines](#) or call a corresponding [driver routine](#) that combines several tasks in one call, such as `?gesv` for factoring and solving. For example, to solve a system of linear equations with a general matrix, call `?getrf` (*LU* factorization) and then `?getrs` (computing the solution). Then, call `?gerfs` to refine the solution and get the error bounds. Alternatively, use the driver routine `?gesvx` that performs all these tasks in one call.



WARNING. LAPACK routines expect that input matrices do not contain `INF` or `NaN` values. When input data is inappropriate for LAPACK, problems may arise, including possible hangs.

Starting from release 8.0, Intel MKL along with Fortran 77 interface to LAPACK computational and driver routines also supports Fortran 95 interface which uses simplified routine calls with shorter argument lists. The syntax section of the routine description gives the calling sequence for Fortran 95 interface immediately after Fortran 77 calls.

Routine Naming Conventions

To call each routine introduced in this chapter from the Fortran 77 program, you can use the LAPACK name.

LAPACK names are listed in [Table 3-1](#) and [Table 3-2](#), and have the structure `?yyzzz` or `?yyzz`, which is described below.

The initial symbol `?` indicates the data type:

s	real, single precision
c	complex, single precision
d	real, double precision
z	complex, double precision

The second and third letters `yy` indicate the matrix type and storage scheme:

ge	general
gb	general band
gt	general tridiagonal
po	symmetric or Hermitian positive-definite
pp	symmetric or Hermitian positive-definite (packed storage)
pb	symmetric or Hermitian positive-definite band
pt	symmetric or Hermitian positive-definite tridiagonal
sy	symmetric indefinite
sp	symmetric indefinite (packed storage)
he	Hermitian indefinite
hp	Hermitian indefinite (packed storage)
tr	triangular
tp	triangular (packed storage)
tb	triangular band

For computational routines, the last three letters `zzz` indicate the computation performed:

trf	form a triangular matrix factorization

<code>trs</code>	solve the linear system with a factored matrix
<code>con</code>	estimate the matrix condition number
<code>rfs</code>	refine the solution and compute error bounds
<code>tri</code>	compute the inverse matrix using the factorization
<code>equ</code>	equilibrate a matrix.

For example, the `sgetrf` routine performs the triangular factorization of general real matrices in single precision; the corresponding routine for complex matrices is `cgetrf`.

For driver routines, the names can end with `-sv` (meaning a *simple* driver), or with `-svx` (meaning an *expert* driver).

Names of the LAPACK computational and driver routines for Fortran 95 interface in Intel MKL are the same as Fortran 77 names but without the first letter that indicates the data type. For example, the name of the routine that performs triangular factorization of general real matrices in Fortran 95 interface is `getrf`. Different data types are handled through defining a specific internal parameter that refers to a module block with named constants for single and double precision.

Fortran 95 Interface Conventions

Fortran 95 interface to LAPACK is implemented through wrappers that call respective Fortran 77 routines. This interface uses such features of Fortran 95 as assumed-shape arrays and optional arguments to provide simplified calls to LAPACK routines with fewer arguments.



NOTE. For LAPACK, Intel MKL offers two types of Fortran 95 interfaces:

- using `mkl_lapack.fi` only through `include 'mkl_lapack.fi'` statement. Such interfaces allow you to make use of the original LAPACK routines with all their arguments
 - using `mkl_lapack.f90` that includes improved interfaces described below. This file is used to generate the module files `mkl95_lapack.mod` and `mkl95_precision.mod`. See also section “Fortran 95 interfaces and wrappers to LAPACK and BLAS” of *Intel® MKL User's Guide* for details. The module files are used to process the FORTRAN use clauses referencing the LAPACK interface: `use mkl95_lapack` and `use mkl95_precision`.
-

The main conventions for Fortran 95 interface are as follows:

- The names of arguments used in Fortran 95 call are typically the same as for the respective generic (Fortran 77) interface. However, to reduce the number of argument names used in the library, Fortran 95 interface uses the following identity settings of formal argument names:

Generic Argument Name	Fortran 95 Argument Name
<i>ap</i>	<i>a</i>
<i>ab</i>	<i>a</i>
<i>afb</i>	<i>af</i>
<i>afp</i>	<i>af</i>
<i>bp</i>	<i>b</i>
<i>bb</i>	<i>b</i>
<i>selctg</i>	<i>select</i>

Note that these name changes of formal arguments have no impact on program semantics and follow the unification conventions.

- Input arguments such as array dimensions are not required in Fortran 95 and are skipped from the calling sequence. Array dimensions are reconstructed from the user data that must exactly follow the required array shape.

Another type of generic arguments that are skipped in Fortran 95 interface are arguments that represent workspace arrays (such as *work*, *rwork*, and so on). The only exception are cases when workspace arrays return significant information on output.

An argument can also be skipped if its value is completely defined by the presence or absence of another argument in the calling sequence, and the restored value is the only meaningful value for the skipped argument.

- Some generic arguments are declared as optional in Fortran 95 interface and may or may not be present in the calling sequence. An argument can be declared optional if it satisfies one of the following conditions:
 - If the argument value is completely defined by the presence or absence of another argument in the calling sequence, it can be declared as optional. The difference from the skipped argument in this case is that the optional argument can have some meaningful values that are distinct from the value reconstructed by default. For example, if some

argument (like *jobz*) can take only two values and one of these values directly implies the use of another argument, then the value of *jobz* can be uniquely reconstructed from the actual presence or absence of this second argument, and *jobz* can be omitted.

- If an input argument can take only a few possible values, it can be declared as optional. The default value of such argument is typically set as the first value in the list and all exceptions to this rule are explicitly stated in the routine description.
- If an input argument has a natural default value, it can be declared as optional. The default value of such optional argument is set to its natural default value.
- Argument *info* is declared as optional in Fortran 95 interface. If it is present in the calling sequence, the value assigned to *info* is interpreted as follows:
 - If this value is more than -1000, its meaning is the same as in Fortran 77 routine.
 - If this value is equal to -1000, it means that there is not enough work memory.
 - If this value is equal to -1001, incompatible arguments are present in the calling sequence.
- Optional arguments are given in square brackets in Fortran 95 call syntax.

“Fortran 95 Notes” subsection at the end of the topic describing each routine details concrete rules for reconstructing the values of omitted optional parameters.

Intel® MKL Fortran 95 Interfaces for LAPACK Routines vs. Netlib Implementation

The following list presents general digressions of the Intel MKL LAPACK95 implementation from the Netlib analog:

- Intel MKL Fortran 95 interfaces are provided for pure procedures.
- Names of interfaces do not contain the `LA_` prefix.
- An optional array argument always has the `target` attribute.
- Functionality of the Intel MKL LAPACK95 wrapper is close to the Fortran 77 original implementation in the `getrf`, `gbtrf`, and `potrf` interfaces.
- If *jobz* argument value specifies presence or absence of *z* argument, then *z* is always declared as optional and *jobz* is restored depending on whether *z* is present or not. It is not always so in the Netlib version (see “[Modified Netlib Interfaces](#)” in Appendix E).
- To avoid double error checking, processing of the *info* argument is limited to checking of the allocated memory and disarranging of optional arguments.
- If an argument that is present in the list of arguments completely defines another argument, the latter is always declared as optional.

You can transform an application that uses the Netlib LAPACK interfaces to ensure its work with the Intel MKL interfaces providing that:

- a.** The application is correct, that is, unambiguous, compiler-independent, and contains no errors.
- b.** Each routine name denotes only one specific routine. If any routine name in the application coincides with a name of the original Netlib routine (for example, after removing the `LA_` prefix) but denotes a routine different from the Netlib original routine, this name should be modified through context name replacement.

You should transform your application in the following cases (see Appendix E for specific differences of individual interfaces):

- When using the Netlib routines that differ from the Intel MKL routines only by the `LA_` prefix or in the array attribute `target`. The only transformation required in this case is context name replacement. See [“Interfaces Identical to Netlib”](#) in Appendix E for details.
- When using Netlib routines that differ from the Intel MKL routines by the `LA_` prefix, the `target` array attribute, and the names of formal arguments. In the case of positional passing of arguments, no additional transformation except context name replacement is required. In the case of the keywords passing of arguments, in addition to the context name replacement the names of mismatching keywords should also be modified. See [“Interfaces with Replaced Argument Names”](#) in Appendix E for details.
- When using the Netlib routines that differ from the respective Intel MKL routines by the `LA_` prefix, the `target` array attribute, sequence of the arguments, arguments missing in Intel MKL but present in Netlib and, vice versa, present in Intel MKL but missing in Netlib. Remove the differences in the sequence and range of the arguments in process of all the transformations when you use the Netlib routines specified by this bullet and the preceding bullet. See [“Modified Netlib Interfaces”](#) in Appendix E for details.
- When using the `getrf`, `gbtrf`, and `potrf` interfaces, that is, new functionality implemented in Intel MKL but unavailable in the Netlib source. To override the differences, build the desired functionality explicitly with the Intel MKL means or create a new subroutine with the new functionality, using specific MKL interfaces corresponding to LAPACK 77 routines. You can call the LAPACK 77 routines directly but using the new Intel MKL interfaces is preferable. See [“Interfaces Absent From Netlib”](#) and [“Interfaces of New Functionality”](#) in Appendix E for details.

Note that if the transformed application calls `getrf`, `gbtrf` or `potrf` without controlling arguments `rcond` and `norm`, just context name replacement is enough in modifying the calls into the Intel MKL interfaces, as described in the first bullet above. The Netlib functionality is preserved in such cases.

- When using the Netlib auxiliary routines. In this case, call a corresponding subroutine directly, using the MKL LAPACK 77 interfaces.

Transform your application as follows:

- 1.** Make sure conditions a. and b. are met.

2. Select Netlib LAPACK 95 calls. For each call do the following:
 - Select the type of digression and do the required transformations.
 - Revise results to eliminate unneeded code or data, which may appear after several identical calls.
3. Make sure the transformations are correct and complete.

Matrix Storage Schemes

LAPACK routines use the following matrix storage schemes:

- *Full storage*: a matrix A is stored in a two-dimensional array a , with the matrix element a_{ij} stored in the array element $a(i, j)$.
- *Packed storage* scheme allows you to store symmetric, Hermitian, or triangular matrices more compactly: the upper or lower triangle of the matrix is packed by columns in a one-dimensional array.
- *Band storage*: an m -by- n band matrix with kl sub-diagonals and ku superdiagonals is stored compactly in a two-dimensional array ab with $kl+ku+1$ rows and n columns. Columns of the matrix are stored in the corresponding columns of the array, and *diagonals* of the matrix are stored in rows of the array.

In Chapters 4 and 5, arrays that hold matrices in packed storage have names ending in p ; arrays with matrices in band storage have names ending in b .

For more information on matrix storage schemes, see “[Matrix Arguments](#)” in Appendix B.

Mathematical Notation

Descriptions of LAPACK routines use the following notation:

$Ax = b$	A system of linear equations with an n -by- n matrix $A = \{a_{ij}\}$, a right-hand side vector $b = \{b_i\}$, and an unknown vector $x = \{x_i\}$.
$AX = B$	A set of systems with a common matrix A and multiple right-hand sides. The columns of B are individual right-hand sides, and the columns of X are the corresponding solutions.
$ x $	the vector with elements $ x_i $ (absolute values of x_i).
$ A $	the matrix with elements $ a_{ij} $ (absolute values of a_{ij}).
$\ x\ _\infty = \max_i x_i $	The <i>infinity-norm</i> of the vector x .

$$\|A\|_{\infty} = \max_j \sum_i |a_{ij}|$$

The *infinity-norm* of the matrix A .

$$\|A\|_1 = \max_i \sum_j |a_{ij}|$$

The *one-norm* of the matrix A . $\|A\|_1 = \|A^T\|_{\infty} = \|A^H\|_{\infty}$

$$\kappa(A) = \|A\| \|A^{-1}\|$$

The *condition number* of the matrix A .

Error Analysis

In practice, most computations are performed with rounding errors. Besides, you often need to solve a system $Ax = b$, where the data (the elements of A and b) are not known exactly. Therefore, it is important to understand how the data errors and rounding errors can affect the solution x .

Data perturbations. If x is the exact solution of $Ax = b$, and $x + \delta x$ is the exact solution of a perturbed problem $(A + \delta A)x = (b + \delta b)$, then

$$\frac{\|\delta x\|}{\|x\|} \leq \kappa(A) \left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|b\|} \right),$$

where

$$\kappa(A) = \|A\| \|A^{-1}\|.$$

In other words, relative errors in A or b may be amplified in the solution vector x by a factor $\kappa(A) = \|A\| \|A^{-1}\|$ called the *condition number* of A .

Rounding errors have the same effect as relative perturbations $c(n)\varepsilon$ in the original data.

Here ε is the *machine precision*, and $c(n)$ is a modest function of the matrix order n . The corresponding solution error is

$$\|\delta x\| / \|x\| \leq c(n) \kappa(A) \varepsilon. \text{ (The value of } c(n) \text{ is seldom greater than } 10n.)$$

Thus, if your matrix A is *ill-conditioned* (that is, its condition number $\kappa(A)$ is very large), then the error in the solution x is also large; you may even encounter a complete loss of precision. LAPACK provides routines that allow you to estimate $\kappa(A)$ (see [Routines for Estimating the Condition Number](#)) and also give you a more precise estimate for the actual solution error (see [Refining the Solution and Estimating Its Error](#)).

Computational Routines

[Table 3-1](#) lists the LAPACK computational routines (Fortran 77 and Fortran 95 interfaces) for factorizing, equilibrating, and inverting *real* matrices, estimating their condition numbers, solving systems of equations with real matrices, refining the solution, and estimating its error. [Table 3-2](#) lists similar routines for *complex* matrices. Respective routine names in Fortran 95 interface are without the first symbol (see [Routine Naming Conventions](#)).

Table 3-1 Computational Routines for Systems of Equations with Real Matrices

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
general	?getrf	?geequ	?getrs	?gecon	?gerfs	?getri
general band	?gbtrf	?gbequ	?gbtrs	?gbcon	?gbrfs	
general tridiagonal	?gttrf		?gttrs	?gtcon	?gtrfs	
symmetric positive-definite	?potrf	?poequ	?potrs	?pocon	?porfs	?potri
symmetric positive-definite, packed storage	?pptrf	?ppequ	?pptrs	?ppcon	?pprfs	?pptri
symmetric positive-definite, band	?pbtrf	?pbequ	?pbtrs	?pbcon	?pbrfs	
symmetric positive-definite, tridiagonal	?pttrf		?pttrs	?ptcon	?ptrfs	
symmetric indefinite	?sytrf		?sytrs	?sycon	?syrrfs	?sytri

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
symmetric indefinite, packed storage	<code>?sptrf</code>		<code>?sptrs</code>	<code>?spcon</code>	<code>?sprfs</code>	<code>?sptri</code>
triangular			<code>?trtrs</code>	<code>?trcon</code>	<code>?trrfs</code>	<code>?trtri</code>
triangular, packed storage			<code>?tptrs</code>	<code>?tpcon</code>	<code>?tprfs</code>	<code>?tptri</code>
triangular band			<code>?tbtrs</code>	<code>?tbcon</code>	<code>?tbrfs</code>	

In the table above, ? denotes s (single precision) or d (double precision) for Fortran 77 interface.

Table 3-2 Computational Routines for Systems of Equations with Complex Matrices

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
general	<code>?getrf</code>	<code>?geequ</code>	<code>?getrs</code>	<code>?gecon</code>	<code>?gerfs</code>	<code>?getri</code>
general band	<code>?gbtrf</code>	<code>?gbequ</code>	<code>?gbtrs</code>	<code>?gbcon</code>	<code>?gbrfs</code>	
general tridiagonal	<code>?gttrf</code>		<code>?gttrs</code>	<code>?gtcon</code>	<code>?gtrfs</code>	
Hermitian positive-definite	<code>?potrf</code>	<code>?poequ</code>	<code>?potrs</code>	<code>?pocon</code>	<code>?porfs</code>	<code>?potri</code>
Hermitian positive-definite, packed storage	<code>?pptrf</code>	<code>?ppequ</code>	<code>?pptrs</code>	<code>?ppcon</code>	<code>?pprfs</code>	<code>?pptri</code>
Hermitian positive-definite, band	<code>?pbtrf</code>	<code>?pbequ</code>	<code>?pbtrs</code>	<code>?pbcon</code>	<code>?pbrfs</code>	
Hermitian positive-definite, tridiagonal	<code>?pttrf</code>		<code>?pttrs</code>	<code>?ptcon</code>	<code>?ptrfs</code>	
Hermitian indefinite	<code>?hetrf</code>		<code>?hetrs</code>	<code>?hecon</code>	<code>?herfs</code>	<code>?hetri</code>

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
symmetric indefinite	<code>?sytrf</code>		<code>?sytrs</code>	<code>?sycon</code>	<code>?syrfs</code>	<code>?sytri</code>
Hermitian indefinite, packed storage	<code>?hptrf</code>		<code>?hptrs</code>	<code>?hpcon</code>	<code>?hprfs</code>	<code>?hptri</code>
symmetric indefinite, packed storage	<code>?sptrf</code>		<code>?sptrs</code>	<code>?spcon</code>	<code>?sprfs</code>	<code>?sptri</code>
triangular			<code>?trtrs</code>	<code>?trcon</code>	<code>?trrfs</code>	<code>?trtri</code>
triangular, packed storage			<code>?tptrs</code>	<code>?tpcon</code>	<code>?tprfs</code>	<code>?tptri</code>
triangular band			<code>?tbtrs</code>	<code>?tbcon</code>	<code>?tbrfs</code>	

In the table above, `?` stands for `c` (single precision complex) or `z` (double precision complex) for Fortran 77 interface.

Routines for Matrix Factorization

This section describes the LAPACK routines for matrix factorization. The following factorizations are supported:

- *LU* factorization
- Cholesky factorization of real symmetric positive-definite matrices
- Cholesky factorization of Hermitian positive-definite matrices
- Bunch-Kaufman factorization of real and complex symmetric matrices
- Bunch-Kaufman factorization of Hermitian matrices.

You can compute:

- the *LU* factorization using full and band storage of matrices
- the Cholesky factorization using full, packed, and band storage
- the Bunch-Kaufman factorization using full and packed storage.

?getrf

Computes the LU factorization of a general m -by- n matrix.

Syntax

FORTRAN 77:

```
call sgetrf( m, n, a, lda, ipiv, info )
call dgetrf( m, n, a, lda, ipiv, info )
call cgetrf( m, n, a, lda, ipiv, info )
call zgetrf( m, n, a, lda, ipiv, info )
```

Fortran 95:

```
call getrf( a [,ipiv] [,info] )
```

Description

The routine computes the LU factorization of a general m -by- n matrix A as

$$A = P * L * U,$$

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U is upper triangular (upper trapezoidal if $m < n$). Usually A is square ($m = n$), and both L and U are triangular. The routine uses partial pivoting, with row interchanges.

Input Parameters

m	INTEGER. The number of rows in the matrix A ($m \geq 0$).
n	INTEGER. The number of columns in A ; $n \geq 0$.
a	REAL for sgetrf DOUBLE PRECISION for dgetrf COMPLEX for cgetrf DOUBLE COMPLEX for zgetrf. Array, DIMENSION ($lda, *$). Contains the matrix A . The second dimension of a must be at least $\max(1, n)$.
lda	INTEGER. The first dimension of array a .

Output Parameters

<i>a</i>	Overwritten by <i>L</i> and <i>U</i> . The unit diagonal elements of <i>L</i> are not stored.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, \min(m, n))$. The pivot indices: row <i>i</i> was interchanged with row <i>ipiv</i> (<i>i</i>).
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , <i>u</i> _{<i>ii</i>} is 0. The factorization has been completed, but <i>U</i> is exactly singular. Division by 0 will occur if you use the factor <i>U</i> for solving a system of linear equations.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `getrf` interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>m</i> , <i>n</i>).
<i>ipiv</i>	Holds the vector of length $\min(m, n)$.

Application Notes

The computed *L* and *U* are the exact factors of a perturbed matrix $A + E$, where

$$|E| \leq c(\min(m, n)) \varepsilon P|L||U|$$

$c(n)$ is a modest linear function of *n*, and ε is the machine precision.

The approximate number of floating-point operations for real flavors is

$(2/3)n^3$	If $m = n$,
$(1/3)n^2(3m-n)$	If $m > n$,
$(1/3)m^2(3n-m)$	If $m < n$.

The number of operations for complex flavors is four times greater.

After calling this routine with $m = n$, you can call the following:

<code>?getrs</code>	to solve $A^*X = B$ or $A^TX = B$ or $A^HX = B$
<code>?gecon</code>	to estimate the condition number of <i>A</i>

`?getri` to compute the inverse of A .

?gbtrf

Computes the LU factorization of a general m -by- n band matrix.

Syntax

FORTRAN 77:

```
call sgbtrf( m, n, kl, ku, ab, ldab, ipiv, info )
call dgbtrf( m, n, kl, ku, ab, ldab, ipiv, info )
call cgbtrf( m, n, kl, ku, ab, ldab, ipiv, info )
call zgbtrf( m, n, kl, ku, ab, ldab, ipiv, info )
```

Fortran 95:

```
call gbtrf( a [,kl] [,m] [,ipiv] [,info] )
```

Description

The routine forms the LU factorization of a general m -by- n band matrix A with kl non-zero subdiagonals and ku non-zero superdiagonals. Usually A is square ($m = n$), and then

$$A = P * L * U,$$

where P is a permutation matrix; L is lower triangular with unit diagonal elements and at most kl non-zero elements in each column; U is an upper triangular band matrix with $kl + ku$ superdiagonals. The routine uses partial pivoting, with row interchanges (which creates the additional kl superdiagonals in U).

Input Parameters

m	INTEGER. The number of rows in matrix A ($m \geq 0$).
n	INTEGER. The number of columns in matrix A ; $n \geq 0$.
kl	INTEGER. The number of subdiagonals within the band of A ; $kl \geq 0$.
ku	INTEGER. The number of superdiagonals within the band of A ; $ku \geq 0$.
ab	REAL for sgbtrf DOUBLE PRECISION for dgbtrf

COMPLEX for `cgbtrf`

DOUBLE COMPLEX for `zgbtrf`.

Array, DIMENSION ($ldab, *$). The array *ab* contains the matrix *A* in band storage (see [Matrix Storage Schemes](#)). The second dimension of *ab* must be at least $\max(1, n)$.

ldab INTEGER. The first dimension of the array *ab*. ($ldab \geq 2 * kl + ku + 1$)

Output Parameters

ab Overwritten by *L* and *U*. The diagonal and $kl + ku$ superdiagonals of *U* are stored in the first $1 + kl + ku$ rows of *ab*. The multipliers used to form *L* are stored in the next *kl* rows.

ipiv INTEGER.
Array, DIMENSION at least $\max(1, \min(m, n))$. The pivot indices: row *i* was interchanged with row *ipiv*(*i*).

info INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.
If *info* = *i*, u_{ii} is 0. The factorization has been completed, but *U* is exactly singular. Division by 0 will occur if you use the factor *U* for solving a system of linear equations.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gbtrf` interface are as follows:

a Stands for argument *ab* in Fortran 77 interface. Holds the array *A* of size $(2 * kl + ku + 1, n)$.

ipiv Holds the vector of length $\min(m, n)$.

kl If omitted, assumed $kl = ku$.

ku Restored as $ku = lda - 2 * kl - 1$.

m If omitted, assumed $m = n$.

Application Notes

The computed L and U are the exact factors of a perturbed matrix $A + E$, where

$$|E| \leq c(kl+ku+1) \varepsilon P|L||U|$$

$c(k)$ is a modest linear function of k , and ε is the machine precision.

The total number of floating-point operations for real flavors varies between approximately $2n(ku+1)kl$ and $2n(kl+ku+1)kl$. The number of operations for complex flavors is four times greater. All these estimates assume that kl and ku are much less than $\min(m, n)$.

After calling this routine with $m = n$, you can call the following routines:

<code>gbtrs</code>	to solve $A^*X = B$ or $A^T * X = B$ or $A^H * X = B$
<code>gbcon</code>	to estimate the condition number of A .

?gttrf

Computes the LU factorization of a tridiagonal matrix.

Syntax

FORTRAN 77:

```
call sgttrf( n, dl, d, du, du2, ipiv, info )
call dgttrf( n, dl, d, du, du2, ipiv, info )
call cgttrf( n, dl, d, du, du2, ipiv, info )
call zgttrf( n, dl, d, du, du2, ipiv, info )
```

Fortran 95:

```
call gttrf( dl, d, du, du2 [, ipiv] [,info] )
```

Description

The routine computes the LU factorization of a real or complex tridiagonal matrix A in the form $A = P^*L^*U$,

where P is a permutation matrix; L is lower bidiagonal with unit diagonal elements; and U is an upper triangular matrix with nonzeros in only the main diagonal and first two superdiagonals. The routine uses elimination with partial pivoting and row interchanges.

Input Parameters

n INTEGER. The order of the matrix *A*; $n \geq 0$.

dl, *d*, *du* REAL for sgttrf
 DOUBLE PRECISION for dgttrf
 COMPLEX for cgttrf
 DOUBLE COMPLEX for zgttrf.
 Arrays containing elements of *A*.
 The array *dl* of dimension $(n - 1)$ contains the subdiagonal elements of *A*.
 The array *d* of dimension *n* contains the diagonal elements of *A*.
 The array *du* of dimension $(n - 1)$ contains the superdiagonal elements of *A*.

Output Parameters

dl Overwritten by the $(n-1)$ multipliers that define the matrix *L* from the *LU* factorization of *A*.

d Overwritten by the *n* diagonal elements of the upper triangular matrix *U* from the *LU* factorization of *A*.

du Overwritten by the $(n-1)$ elements of the first superdiagonal of *U*.

du2 REAL for sgttrf
 DOUBLE PRECISION for dgttrf
 COMPLEX for cgttrf
 DOUBLE COMPLEX for zgttrf.
 Array, dimension $(n-2)$. On exit, *du2* contains $(n-2)$ elements of the second superdiagonal of *U*.

ipiv INTEGER.
 Array, dimension (n) . The pivot indices: row *i* was interchanged with row *ipiv*(*i*).

info INTEGER. If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.
 If *info* = *i*, u_{ii} is 0. The factorization has been completed, but *U* is exactly singular. Division by zero will occur if you use the factor *U* for solving a system of linear equations.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gttrf` interface are as follows:

<code>d1</code>	Holds the vector of length $(n-1)$.
<code>d</code>	Holds the vector of length (n) .
<code>du</code>	Holds the vector of length $(n-1)$.
<code>du2</code>	Holds the vector of length $(n-2)$.
<code>ipiv</code>	Holds the vector of length (n) .

Application Notes

<code>?gbtrs</code>	to solve $A * X = B$ or $A^T * X = B$ or $A^H * X = B$
<code>?gbcon</code>	to estimate the condition number of A .

?potrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix.

Syntax

FORTRAN 77:

```
call spotrf( uplo, n, a, lda, info )
call dpotrf( uplo, n, a, lda, info )
call cpotrf( uplo, n, a, lda, info )
call zpotrf( uplo, n, a, lda, info )
```

Fortran 95:

```
call potrf( a [, uplo] [,info] )
```

Description

This routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A :

$$A = U^H * U \quad \text{if } uplo = 'U'$$

$$A = L * L^H \quad \text{if } uplo = 'L',$$

where L is a lower triangular matrix and U is upper triangular.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored and how A is factored: If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular part of the matrix A , and A is factored as $U^H * U$. If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular part of the matrix A ; A is factored as $L * L^H$.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>a</i>	REAL for <code>spotrf</code> DOUBLE PRECISION for <code>dpotrf</code> COMPLEX for <code>cpotrf</code> DOUBLE COMPLEX for <code>zpotrf</code> . Array, DIMENSION (<i>lda</i> , *). The array <i>a</i> contains either the upper or the lower triangular part of the matrix A (see <i>uplo</i>). The second dimension of <i>a</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> .

Output Parameters

<i>a</i>	The upper or lower triangular part of <i>a</i> is overwritten by the Cholesky factor U or L , as specified by <i>uplo</i> .
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , the leading minor of order <i>i</i> (and therefore the matrix A itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix A .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `potrf` interface are as follows:

`a` Holds the matrix A of size (n, n) .
`uplo` Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If `uplo` = 'U', the computed factor U is the exact factor of a perturbed matrix $A + E$, where

$$|E| \leq c(n)\varepsilon |U^H| |U|, |e_{ij}| \leq c(n)\varepsilon \sqrt{a_{ii}a_{jj}}$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

A similar estimate holds for `uplo` = 'L'.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors or $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following routines:

`?potrs` to solve $A^*X = B$
`?pocon` to estimate the condition number of A
`?potri` to compute the inverse of A .

?pptrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix using packed storage.

Syntax

FORTRAN 77:

```
call spptrf( uplo, n, ap, info )
call dpptrf( uplo, n, ap, info )
call cpptrf( uplo, n, ap, info )
call zpptrf( uplo, n, ap, info )
```

Fortran 95:

```
call ppstrf( a [, uplo] [,info] )
```

Description

This routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite packed matrix A :

$$A = U^H * U \quad \text{if } uplo = 'U'$$

$$A = L * L^H \quad \text{if } uplo = 'L',$$

where L is a lower triangular matrix and U is upper triangular.

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
Indicates whether the upper or lower triangular part of A is packed in the array *ap*, and how A is factored:
If *uplo* = 'U', the array *ap* stores the upper triangular part of the matrix A , and A is factored as $U^H * U$.
If *uplo* = 'L', the array *ap* stores the lower triangular part of the matrix A ; A is factored as $L * L^H$.

n INTEGER. The order of matrix A ; $n \geq 0$.

ap REAL for spstrf
DOUBLE PRECISION for dpstrf
COMPLEX for cpstrf
DOUBLE COMPLEX for zpstrf.
Array, DIMENSION at least $\max(1, n(n+1)/2)$. The array *ap* contains either the upper or the lower triangular part of the matrix A (as specified by *uplo*) in packed storage (see [Matrix Storage Schemes](#)).

Output Parameters

ap The upper or lower triangular part of A in packed storage is overwritten by the Cholesky factor U or L , as specified by *uplo*.

info INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

If $info = i$, the leading minor of order i (and therefore the matrix A itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix A .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `pptrf` interface are as follows:

<code>a</code>	Stands for argument <code>ap</code> in Fortran 77 interface. Holds the array A of size $(n * (n+1) / 2)$.
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If `uplo = 'U'`, the computed factor U is the exact factor of a perturbed matrix $A + E$, where

$$|E| \leq c(n)\epsilon |U^H| |U|, |e_{ij}| \leq c(n)\epsilon \sqrt{a_{ii}a_{jj}}$$

$c(n)$ is a modest linear function of n , and ϵ is the machine precision.

A similar estimate holds for `uplo = 'L'`.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors and $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following routines:

<code>?pptrs</code>	to solve $A * X = B$
<code>?ppcon</code>	to estimate the condition number of A
<code>?pptri</code>	to compute the inverse of A .

?pbtrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite band matrix.

Syntax

FORTRAN 77:

```
call spbtrf( uplo, n, kd, ab, ldab, info )
call dpbtrf( uplo, n, kd, ab, ldab, info )
call cpbtrf( uplo, n, kd, ab, ldab, info )
call zpbtrf( uplo, n, kd, ab, ldab, info )
```

Fortran 95:

```
call pbtrf( a [, uplo] [,info] )
```

Description

This routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite band matrix A :

$A = U^T * U$ for real data, if $uplo = 'U'$

$A = U^H * U$ for complex data

$A = L * L^T$ for real data, if $uplo = 'L'$,

$A = L * L^H$ for complex data

where L is a lower triangular matrix and U is upper triangular.

Input Parameters

$uplo$	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored in the array ab , and how A is factored: If $uplo = 'U'$, the upper triangle of A is stored. If $uplo = 'L'$, the lower triangle of A is stored.
n	INTEGER. The order of matrix A ; $n \geq 0$.

<i>kd</i>	INTEGER. The number of superdiagonals or subdiagonals in the matrix A ($kd \geq 0$).
<i>ab</i>	REAL for <code>spbtrf</code> DOUBLE PRECISION for <code>dpbtrf</code> COMPLEX for <code>cpbtrf</code> DOUBLE COMPLEX for <code>zpbtrf</code> . Array, DIMENSION ($*$). The array <i>ap</i> contains either the upper or the lower triangular part of the matrix A (as specified by <i>uplo</i>) in band storage (see Matrix Storage Schemes). The second dimension of <i>ab</i> must be at least $\max(1, n)$.
<i>ldab</i>	INTEGER. The first dimension of the array <i>ab</i> . ($ldab \geq kd + 1$)

Output Parameters

<i>ap</i>	The upper or lower triangular part of A (in band storage) is overwritten by the Cholesky factor U or L , as specified by <i>uplo</i> .
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. If <i>info</i> = $-i$, the i -th parameter had an illegal value. If <i>info</i> = i , the leading minor of order i (and therefore the matrix A itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix A .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `pbtrf` interface are as follows:

<i>a</i>	Stands for argument <i>ab</i> in Fortran 77 interface. Holds the array A of size $(kd+1, n)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If *uplo* = 'U', the computed factor U is the exact factor of a perturbed matrix $A + E$, where

$$|E| \leq c(kd + 1)\varepsilon |U^H| |U|, |e_{ij}| \leq c(kd + 1)\varepsilon \sqrt{a_{ii}a_{jj}}$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

A similar estimate holds for $uplo = 'L'$.

The total number of floating-point operations for real flavors is approximately $n(kd+1)^2$. The number of operations for complex flavors is 4 times greater. All these estimates assume that kd is much less than n .

After calling this routine, you can call the following routines:

`?pbtrs` to solve $A^*X = B$
`?pbcon` to estimate the condition number of A .

?pttrf

Computes the factorization of a symmetric (Hermitian) positive-definite tridiagonal matrix.

Syntax

FORTRAN 77:

```
call spttrf( n, d, e, info )
call dpttrf( n, d, e, info )
call cpttrf( n, d, e, info )
call zpttrf( n, d, e, info )
```

Fortran 95:

```
call pttrf( d, e [,info] )
```

Description

This routine forms the factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite tridiagonal matrix A :

$$A = L^*D^*L',$$

where D is diagonal and L is unit lower bidiagonal. The factorization may also be regarded as having the form $A = U^*D^*U$, where D is unit upper bidiagonal.

Input Parameters

<i>n</i>	INTEGER. The order of the matrix <i>A</i> ; $n \geq 0$.
<i>d</i>	REAL for <code>spttrf</code> , <code>cpttrf</code> DOUBLE PRECISION for <code>dpttrf</code> , <code>zpttrf</code> . Array, dimension (<i>n</i>). Contains the diagonal elements of <i>A</i> .
<i>e</i>	REAL for <code>spttrf</code> DOUBLE PRECISION for <code>dpttrf</code> COMPLEX for <code>cpttrf</code> DOUBLE COMPLEX for <code>zpttrf</code> . Array, dimension (<i>n</i> - 1). Contains the subdiagonal elements of <i>A</i> .

Output Parameters

<i>d</i>	Overwritten by the <i>n</i> diagonal elements of the diagonal matrix <i>D</i> from the L^*D^*L' factorization of <i>A</i> .
<i>e</i>	Overwritten by the (<i>n</i> - 1) off-diagonal elements of the unit bidiagonal factor <i>L</i> or <i>U</i> from the factorization of <i>A</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , the leading minor of order <i>i</i> (and therefore the matrix <i>A</i> itself) is not positive-definite; if $i < n$, the factorization could not be completed, while if $i = n$, the factorization was completed, but $d(n) \leq 0$.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `pttrf` interface are as follows:

<i>d</i>	Holds the vector of length (<i>n</i>).
<i>e</i>	Holds the vector of length (<i>n</i> -1).

?sytrf

Computes the Bunch-Kaufman factorization of a symmetric matrix.

Syntax

FORTRAN 77:

```
call ssytrf( uplo, n, a, lda, ipiv, work, lwork, info )
call dsytrf( uplo, n, a, lda, ipiv, work, lwork, info )
call csytrf( uplo, n, a, lda, ipiv, work, lwork, info )
call zsytrf( uplo, n, a, lda, ipiv, work, lwork, info )
```

Fortran 95:

```
call sytrf( a [, uplo] [,ipiv] [, info] )
```

Description

This routine forms the Bunch-Kaufman factorization of a symmetric matrix:

if $uplo = 'U'$, $A = P * U * D * U^T * P^T$
if $uplo = 'L'$, $A = P * L * D * L^T * P^T$,

where A is the input matrix, P is a permutation matrix, U and L are upper and lower triangular matrices with unit diagonal, and D is a symmetric block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D .

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
Indicates whether the upper or lower triangular part of A is stored and how A is factored:
If $uplo = 'U'$, the array a stores the upper triangular part of the matrix A , and A is factored as $P * U * D * U^T * P^T$.
If $uplo = 'L'$, the array a stores the lower triangular part of the matrix A , and A is factored as $P * L * D * L^T * P^T$.

n INTEGER. The order of matrix A ; $n \geq 0$.

a REAL for ssytrf
DOUBLE PRECISION for dsytrf
COMPLEX for csytrf
DOUBLE COMPLEX for zsytrf.
Array, DIMENSION (*lda*, *). The array *a* contains either the upper or the lower triangular part of the matrix *A* (see *uplo*). The second dimension of *a* must be at least $\max(1, n)$.

lda INTEGER. The first dimension of *a*; at least $\max(1, n)$.

work Same type as *a*. A workspace array, dimension at least $\max(1, lwork)$.

lwork INTEGER. The size of the *work* array ($lwork \geq n$).
If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.
See [Application Notes](#) for the suggested value of *lwork*.

Output Parameters

a The upper or lower triangular part of *a* is overwritten by details of the block-diagonal matrix *D* and the multipliers used to obtain the factor *U* (or *L*).

work(1) If *info*=0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

ipiv INTEGER.
Array, DIMENSION at least $\max(1, n)$. Contains details of the interchanges and the block structure of *D*. If $ipiv(i) = k > 0$, then d_{ii} is a 1-by-1 block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column.
If $uplo = 'U'$ and $ipiv(i) = ipiv(i-1) = -m < 0$, then *D* has a 2-by-2 block in rows/columns *i* and *i-1*, and (*i-1*)-th row and column of *A* was interchanged with the *m*-th row and column.
If $uplo = 'L'$ and $ipiv(i) = ipiv(i+1) = -m < 0$, then *D* has a 2-by-2 block in rows/columns *i* and *i+1*, and (*i+1*)-th row and column of *A* was interchanged with the *m*-th row and column.

info INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

If $info = i$, d_{ii} is 0. The factorization has been completed, but D is exactly singular. Division by 0 will occur if you use D for solving a system of linear equations.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sytrf` interface are as follows:

<code>a</code>	holds the matrix A of size (n, n)
<code>ipiv</code>	holds the vector of length (n)
<code>uplo</code>	must be 'U' or 'L'. The default value is 'U'.

Application Notes

For better performance, try using $lwork = n * blocksize$, where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The 2-by-2 unit diagonal blocks and the unit diagonal elements of U and L are not stored. The remaining elements of U and L are stored in the corresponding columns of the array `a`, but additional row interchanges are required to recover U or L explicitly (which is seldom necessary).

If $ipiv(i) = i$ for all $i = 1 \dots n$, then all off-diagonal elements of U (L) are stored explicitly in the corresponding elements of the array `a`.

If `uplo = 'U'`, the computed factors U and D are the exact factors of a perturbed matrix $A + E$, where

$$|E| \leq c(n)\varepsilon |P||U||D||U^T|P^T$$

$c(n)$ is a modest linear function of n , and ε is the machine precision. A similar estimate holds for the computed L and D when `uplo = 'L'`.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors or $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following routines:

<code>?sytrs</code>	to solve $A^*X = B$
<code>?sycon</code>	to estimate the condition number of A
<code>?sytri</code>	to compute the inverse of A .

?hetrf

Computes the Bunch-Kaufman factorization of a complex Hermitian matrix.

Syntax

FORTRAN 77:

```
call chetrf( uplo, n, a, lda, ipiv, work, lwork, info )
call zhetrf( uplo, n, a, lda, ipiv, work, lwork, info )
```

Fortran 95:

```
call hetrf( a [, uplo] [,ipiv] [,info] )
```

Description

This routine forms the Bunch-Kaufman factorization of a Hermitian matrix:

if `uplo='U'`, $A = P^*U^*D^*U^H*P^T$

if `uplo='L'`, $A = P^*L^*D^*L^H*P^T$,

where A is the input matrix, P is a permutation matrix, U and L are upper and lower triangular matrices with unit diagonal, and D is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D .

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored and how A is factored: If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular part of the matrix A, and A is factored as $P*U*D*U^H*P^T$. If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular part of the matrix A, and A is factored as $P*L*D*L^H*P^T$.
<i>n</i>	INTEGER. The order of matrix A; $n \geq 0$.
<i>a, work</i>	COMPLEX for <code>chetr</code> DOUBLE COMPLEX for <code>zhetr</code>. Arrays, DIMENSION $a(lda, *)$, $work(*)$. The array <i>a</i> contains the upper or the lower triangular part of the matrix A (see <i>uplo</i>). The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i>(*) is a workspace array of dimension at least $\max(1, lwork)$.
<i>lda</i>	INTEGER. The first dimension of <i>a</i>; at least $\max(1, n)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array ($lwork \geq n$). If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <code>xerbla</code>. See Application Notes for the suggested value of <i>lwork</i>.

Output Parameters

<i>a</i>	The upper or lower triangular part of <i>a</i> is overwritten by details of the block-diagonal matrix D and the multipliers used to obtain the factor U (or L).
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i>(1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>ipiv</i>	INTEGER.

Array, `DIMENSION` at least $\max(1, n)$. Contains details of the interchanges and the block structure of D . If $ipiv(i) = k > 0$, then d_{ii} is a 1-by-1 block, and the i -th row and column of A was interchanged with the k -th row and column.

If $uplo = 'U'$ and $ipiv(i) = ipiv(i-1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i-1$, and the $(i-1)$ -th row and column of A was interchanged with the m -th row and column.

If $uplo = 'L'$ and $ipiv(i) = ipiv(i+1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i+1$, and the $(i+1)$ -th row and column of A was interchanged with the m -th row and column.

info

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, d_{ii} is 0. The factorization has been completed, but D is exactly singular. Division by 0 will occur if you use D for solving a system of linear equations.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hetrf` interface are as follows:

<i>a</i>	holds the matrix A of size (n, n)
<i>ipiv</i>	holds the vector of length (n)
<i>uplo</i>	must be 'U' or 'L'. The default value is 'U'.

Application Notes

This routine is suitable for Hermitian matrices that are not known to be positive-definite. If A is in fact positive-definite, the routine does not perform interchanges, and no 2-by-2 diagonal blocks occur in D .

For better performance, try using $lwork = n * blocksize$, where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The 2-by-2 unit diagonal blocks and the unit diagonal elements of *U* and *L* are not stored. The remaining elements of *U* and *L* are stored in the corresponding columns of the array *a*, but additional row interchanges are required to recover *U* or *L* explicitly (which is seldom necessary).

If *ipiv*(*i*) = *i* for all *i* = 1 . . . *n*, then all off-diagonal elements of *U* (*L*) are stored explicitly in the corresponding elements of the array *a*.

If *uplo* = 'U', the computed factors *U* and *D* are the exact factors of a perturbed matrix *A* + *E*, where

$$|E| \leq c(n) \varepsilon P |U| |D| |U^T| P^T$$

c(*n*) is a modest linear function of *n*, and ε is the machine precision.

A similar estimate holds for the computed *L* and *D* when *uplo* = 'L'.

The total number of floating-point operations is approximately $(4/3)n^3$.

After calling this routine, you can call the following routines:

<code>?hetrs</code>	to solve $A * X = B$
<code>?hecon</code>	to estimate the condition number of <i>A</i>
<code>?hetri</code>	to compute the inverse of <i>A</i> .

?sptf

Computes the Bunch-Kaufman factorization of a symmetric matrix using packed storage.

Syntax

FORTRAN 77:

```
call ssptf( uplo, n, ap, ipiv, info )
call dsptf( uplo, n, ap, ipiv, info )
call csptf( uplo, n, ap, ipiv, info )
call zsptf( uplo, n, ap, ipiv, info )
```

Fortran 95:

```
call sptrf( a [,uplo] [,ipiv] [,info] )
```

Description

This routine forms the Bunch-Kaufman factorization of a symmetric matrix A using packed storage:

if $uplo='U'$, $A = P*U*D*U^T*P^T$
 if $uplo='L'$, $A = P*L*D*L^T*P^T$,

where P is a permutation matrix, U and L are upper and lower triangular matrices with unit diagonal, and D is a symmetric block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D .

Input Parameters

$uplo$	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is packed in the array ap and how A is factored: If $uplo = 'U'$, the array ap stores the upper triangular part of the matrix A , and A is factored as $P*U*D*U^T*P^T$. If $uplo = 'L'$, the array ap stores the lower triangular part of the matrix A , and A is factored as $P*L*D*L^T*P^T$.
n	INTEGER. The order of matrix A ; $n \geq 0$.

ap REAL for `ssptf`
 DOUBLE PRECISION for `dsptf`
 COMPLEX for `csptf`
 DOUBLE COMPLEX for `zsptf`.
 Array, DIMENSION at least $\max(1, n(n+1)/2)$. The array *ap* contains the upper or the lower triangular part of the matrix *A* (as specified by *uplo*) in *packed storage* (see [Matrix Storage Schemes](#)).

Output Parameters

ap The upper or lower triangle of *A* (as specified by *uplo*) is overwritten by details of the block-diagonal matrix *D* and the multipliers used to obtain the factor *U* (or *L*).

ipiv INTEGER.
 Array, DIMENSION at least $\max(1, n)$. Contains details of the interchanges and the block structure of *D*. If $ipiv(i) = k > 0$, then d_{ii} is a 1-by-1 block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column.
 If $uplo = 'U'$ and $ipiv(i) = ipiv(i-1) = -m < 0$, then *D* has a 2-by-2 block in rows/columns *i* and *i-1*, and the (*i-1*)-th row and column of *A* was interchanged with the *m*-th row and column.
 If $uplo = 'L'$ and $ipiv(i) = ipiv(i+1) = -m < 0$, then *D* has a 2-by-2 block in rows/columns *i* and *i+1*, and the (*i+1*)-th row and column of *A* was interchanged with the *m*-th row and column.

info INTEGER. If $info = 0$, the execution is successful.
 If $info = -i$, the *i*-th parameter had an illegal value.
 If $info = i$, d_{ii} is 0. The factorization has been completed, but *D* is exactly singular. Division by 0 will occur if you use *D* for solving a system of linear equations.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sptf` interface are as follows:

a stands for argument *ap* in Fortran 77 interface. Holds the array *A* of size $(n * (n+1) / 2)$.

ipiv holds the vector of length (*n*).
uplo must be 'U' or 'L'. The default value is 'U'.

Application Notes

The 2-by-2 unit diagonal blocks and the unit diagonal elements of *U* and *L* are not stored. The remaining elements of *U* and *L* overwrite elements of the corresponding columns of the matrix *A*, but additional row interchanges are required to recover *U* or *L* explicitly (which is seldom necessary).

If *ipiv*(*i*) = *i* for all *i* = 1...*n*, then all off-diagonal elements of *U* (*L*) are stored explicitly in packed form.

If *uplo* = 'U', the computed factors *U* and *D* are the exact factors of a perturbed matrix *A* + *E*, where

$$|E| \leq c(n)\varepsilon P|U||D||U^T|P^T$$

c(*n*) is a modest linear function of *n*, and ε is the machine precision. A similar estimate holds for the computed *L* and *D* when *uplo* = 'L'.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors or $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following routines:

?sptsr to solve $A * X = B$
 ?spcon to estimate the condition number of *A*
 ?sptri to compute the inverse of *A*.

?hptrf

Computes the Bunch-Kaufman factorization of a complex Hermitian matrix using packed storage.

Syntax

FORTRAN 77:

```
call chptrf( uplo, n, ap, ipiv, info )
call zhptrf( uplo, n, ap, ipiv, info )
```

Fortran 95:

```
call hptrf( a [,uplo] [,ipiv] [,info] )
```

Description

This routine forms the Bunch-Kaufman factorization of a Hermitian matrix using packed storage:

if $uplo='U'$, $A = P*U*D*U^H*P^T$
 if $uplo='L'$, $A = P*L*D*L^H*P^T$,

where A is the input matrix, P is a permutation matrix, U and L are upper and lower triangular matrices with unit diagonal, and D is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D .

Input Parameters

$uplo$ CHARACTER*1. Must be 'U' or 'L'.
 Indicates whether the upper or lower triangular part of A is packed and how A is factored:
 If $uplo = 'U'$, the array ap stores the upper triangular part of the matrix A , and A is factored as $P*U*D*U^H*P^T$.
 If $uplo = 'L'$, the array ap stores the lower triangular part of the matrix A , and A is factored as $P*L*D*L^H*P^T$.

n INTEGER. The order of matrix A ; $n \geq 0$.

ap COMPLEX for `chptrf`
 DOUBLE COMPLEX for `zhptrf`.
 Array, DIMENSION at least $\max(1, n(n+1)/2)$. The array ap contains the upper or the lower triangular part of the matrix A (as specified by $uplo$) in *packed storage* (see [Matrix Storage Schemes](#)).

Output Parameters

ap The upper or lower triangle of A (as specified by $uplo$) is overwritten by details of the block-diagonal matrix D and the multipliers used to obtain the factor U (or L).

$ipiv$ INTEGER.

Array, `DIMENSION` at least $\max(1, n)$. Contains details of the interchanges and the block structure of D . If $ipiv(i) = k > 0$, then d_{ii} is a 1-by-1 block, and the i -th row and column of A was interchanged with the k -th row and column.

If $uplo = 'U'$ and $ipiv(i) = ipiv(i-1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i-1$, and the $(i-1)$ -th row and column of A was interchanged with the m -th row and column.

If $uplo = 'L'$ and $ipiv(i) = ipiv(i+1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i+1$, and the $(i+1)$ -th row and column of A was interchanged with the m -th row and column.

info

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, d_{ii} is 0. The factorization has been completed, but D is exactly singular. Division by 0 will occur if you use D for solving a system of linear equations.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hpztrf` interface are as follows:

<i>a</i>	Stands for argument <i>ap</i> in Fortran 77 interface. Holds the array A of size $(n * (n+1) / 2)$.
<i>ipiv</i>	Holds the vector of length (n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The 2-by-2 unit diagonal blocks and the unit diagonal elements of U and L are not stored. The remaining elements of U and L are stored in the corresponding columns of the array a , but additional row interchanges are required to recover U or L explicitly (which is seldom necessary).

If $ipiv(i) = i$ for all $i = 1 \dots n$, then all off-diagonal elements of U (L) are stored explicitly in the corresponding elements of the array a .

If $uplo = 'U'$, the computed factors U and D are the exact factors of a perturbed matrix $A + E$, where

$$|E| \leq c(n) \varepsilon_P |U| |D| |U^T| P^T$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

A similar estimate holds for the computed L and D when $uplo = 'L'$.

The total number of floating-point operations is approximately $(4/3)n^3$.

After calling this routine, you can call the following routines:

<code>?hptrs</code>	to solve $A^*X = B$
<code>?hpcon</code>	to estimate the condition number of A
<code>?hptri</code>	to compute the inverse of A .

Routines for Solving Systems of Linear Equations

This section describes the LAPACK routines for solving systems of linear equations. Before calling most of these routines, you need to factorize the matrix of your system of equations (see [Routines for Matrix Factorization](#) in this chapter). However, the factorization is not necessary if your system of equations has a triangular matrix.

?getrs

Solves a system of linear equations with an LU-factored square matrix, with multiple right-hand sides.

Syntax

FORTRAN 77:

```
call sgetrs( trans, n, nrhs, a, lda, ipiv, b, ldb, info )
call dgetrs( trans, n, nrhs, a, lda, ipiv, b, ldb, info )
call cgetrs( trans, n, nrhs, a, lda, ipiv, b, ldb, info )
call zgetrs( trans, n, nrhs, a, lda, ipiv, b, ldb, info )
```

Fortran 95:

```
call getrs( a, ipiv, b [, trans] [,info] )
```

Description

This routine solves for x the following systems of linear equations:

$A^*X = B$ if $trans = 'N'$,

$A^T * X = B$ if *trans* = 'T',
 $A^H * X = B$ if *trans* = 'C' (for complex matrices only).

Before calling this routine, you must call [?getrf](#) to compute the *LU* factorization of *A*.

Input Parameters

trans CHARACTER*1. Must be 'N' or 'T' or 'C'.
 Indicates the form of the equations:
 If *trans* = 'N', then $A * X = B$ is solved for *X*.
 If *trans* = 'T', then $A^T * X = B$ is solved for *X*.
 If *trans* = 'C', then $A^H * X = B$ is solved for *X*.

n INTEGER. The order of *A*; the number of rows in *B* ($n \geq 0$).

nrhs INTEGER. The number of right-hand sides; $nrhs \geq 0$.

a, b REAL for sgetrs
 DOUBLE PRECISION for dgetrs
 COMPLEX for cgetrs
 DOUBLE COMPLEX for zgetrs.
 Arrays: *a*(*lda*,*), *b*(*ldb*,*).
 The array *a* contains *LU* factorization of matrix *A* resulting from the call of [?getrf](#).
 The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.
 The second dimension of *a* must be at least $\max(1, n)$, the second dimension of *b* at least $\max(1, nrhs)$.

lda INTEGER. The first dimension of *a*; $lda \geq \max(1, n)$.

ldb INTEGER. The first dimension of *b*; $ldb \geq \max(1, n)$.

ipiv INTEGER.
 Array, DIMENSION at least $\max(1, n)$. The *ipiv* array, as returned by [?getrf](#).

Output Parameters

b Overwritten by the solution matrix *X*.

info INTEGER. If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `getrs` interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>nrhs</i>).
<i>ipiv</i>	Holds the vector of length (<i>n</i>).
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon P|L||U|$$

$c(n)$ is a modest linear function of *n*, and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \operatorname{cond}(A, x) \varepsilon$$

where $\operatorname{cond}(A, x) = \frac{\|A^{-1}\|_\infty \|A\|_\infty \|x\|_\infty}{\|x\|_\infty} \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\operatorname{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector *b* is $2n^2$ for real flavors and $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call [?gecon](#).

To refine the solution and estimate the error, call [?gerfs](#).

?gbtrs

Solves a system of linear equations with an LU-factored band matrix, with multiple right-hand sides.

Syntax

FORTRAN 77:

```
call sgbtrs( trans, n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call dgbtrs( trans, n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call cgbtrs( trans, n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call zgbtrs( trans, n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
```

Fortran 95:

```
call gbtrs( a, b, ipiv, [, kl] [, trans] [, info] )
```

Description

This routine solves for X the following systems of linear equations:

$A * X = B$ if $trans = 'N'$,
 $A^T * X = B$ if $trans = 'T'$,
 $A^H * X = B$ if $trans = 'C'$ (for complex matrices only).

Here A is an LU -factored general band matrix of order n with kl non-zero subdiagonals and ku nonzero superdiagonals. Before calling this routine, call [?gbtrf](#) to compute the LU factorization of A .

Input Parameters

<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'.
<i>n</i>	INTEGER. The order of A ; the number of rows in B ; $n \geq 0$.
<i>kl</i>	INTEGER. The number of subdiagonals within the band of A ; $kl \geq 0$.
<i>ku</i>	INTEGER. The number of superdiagonals within the band of A ; $ku \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ab, b</i>	REAL for sgbtrs

DOUBLE PRECISION for `dgbtrs`

COMPLEX for `cgbtrs`

DOUBLE COMPLEX for `zgbtrs`.

Arrays: `ab(ldab,*)`, `b(ldb,*)`.

The array `ab` contains the matrix A in *band storage* (see [Matrix Storage Schemes](#)).

The array `b` contains the matrix B whose columns are the right-hand sides for the systems of equations.

The second dimension of `ab` must be at least $\max(1, n)$, and the second dimension of `b` at least $\max(1, nrhs)$.

`ldab` INTEGER. The leading dimension of the array `ab`; $ldab \geq 2*kl + ku + 1$.

`ldb` INTEGER. The leading dimension of `b`; $ldb \geq \max(1, n)$.

`ipiv` INTEGER. Array, DIMENSION at least $\max(1, n)$. The `ipiv` array, as returned by `?gbtrf`.

Output Parameters

`b` Overwritten by the solution matrix X .

`info` INTEGER. If `info=0`, the execution is successful.
If `info = -i`, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gbtrs` interface are as follows:

`a` Stands for argument `ab` in Fortran 77 interface. Holds the array A of size $(2*kl+ku+1, n)$.

`b` Holds the matrix B of size $(n, nrhs)$.

`ipiv` Holds the vector of length $\min(m, n)$.

`kl` If omitted, assumed $kl = ku$.

`ku` Restored as $lda-2*kl-1$.

`trans` Must be 'N', 'C', or 'T'. The default value is 'N'.

Application Notes

For each right-hand side b , the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(kl + ku + 1)\varepsilon P|L||U|$$

$c(k)$ is a modest linear function of k , and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(kl + ku + 1) \text{cond}(A, x) \varepsilon$$

where $\text{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \| |A^{-1}| \|_\infty \| |A| \|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector is $2n(ku + 2kl)$ for real flavors. The number of operations for complex flavors is 4 times greater. All these estimates assume that kl and ku are much less than $\min(m, n)$.

To estimate the condition number $\kappa_\infty(A)$, call [?gbcon](#).

To refine the solution and estimate the error, call [?gbrfs](#).

?gttrs

Solves a system of linear equations with a tridiagonal matrix using the LU factorization computed by ?gttrf.

Syntax

FORTRAN 77:

```
call sgtrrs( trans, n, nrhs, dl, d, du, du2, ipiv, b, ldb, info )
call dgtrrs( trans, n, nrhs, dl, d, du, du2, ipiv, b, ldb, info )
call cgtrrs( trans, n, nrhs, dl, d, du, du2, ipiv, b, ldb, info )
call zgtrrs( trans, n, nrhs, dl, d, du, du2, ipiv, b, ldb, info )
```

Fortran 95:

```
call gttrs( dl, d, du, du2, b, ipiv [, trans] [,info] )
```

Description

This routine solves for x the following systems of linear equations with multiple right hand sides:

$A * X = B$ if $trans = 'N'$,
 $A^T * X = B$ if $trans = 'T'$,
 $A^H * X = B$ if $trans = 'C'$ (for complex matrices only).

Before calling this routine, you must call [?gttrf](#) to compute the LU factorization of A .

Input Parameters

<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If $trans = 'N'$, then $A * X = B$ is solved for x . If $trans = 'T'$, then $A^T * X = B$ is solved for x . If $trans = 'C'$, then $A^H * X = B$ is solved for x .
<i>n</i>	INTEGER. The order of A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, that is, the number of columns in B ; $nrhs \geq 0$.

dl,d,du,du2,b REAL for sgttrs
 DOUBLE PRECISION for dgttrs
 COMPLEX for cgttrs
 DOUBLE COMPLEX for zgttrs.
Arrays: *dl*(*n - 1*), *d*(*n*), *du*(*n - 1*), *du2*(*n - 2*), *b*(*ldb,nrhs*).
 The array *dl* contains the (*n - 1*) multipliers that define the matrix *L* from the *LU* factorization of *A*.
 The array *d* contains the *n* diagonal elements of the upper triangular matrix *U* from the *LU* factorization of *A*.
 The array *du* contains the (*n - 1*) elements of the first superdiagonal of *U*.
 The array *du2* contains the (*n - 2*) elements of the second superdiagonal of *U*.
 The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.

ldb INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

ipiv INTEGER. Array, DIMENSION (*n*). The *ipiv* array, as returned by [?gttrf](#).

Output Parameters

b Overwritten by the solution matrix *x*.

info INTEGER. If *info*=0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *gttrs* interface are as follows:

dl Holds the vector of length (*n-1*).

d Holds the vector of length (*n*).

du Holds the vector of length (*n-1*).

du2 Holds the vector of length (*n-2*).

b Holds the matrix *B* of size (*n,nrhs*).

ipiv Holds the vector of length (n).
trans Must be 'N', 'C', or 'T'. The default value is 'N'.

Application Notes

For each right-hand side b , the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon P|L||U|$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \text{cond}(A, x) \varepsilon$$

where $\text{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \| |A^{-1}| \|_\infty \| |A| \|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector b is $7n$ (including n divisions) for real flavors and $34n$ (including $2n$ divisions) for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call [?gtcon](#).

To refine the solution and estimate the error, call [?gtrfs](#).

?potrs

Solves a system of linear equations with a Cholesky-factored symmetric (Hermitian) positive-definite matrix.

Syntax

FORTRAN 77:

```
call spotrs( uplo, n, nrhs, a, lda, b, ldb, info )
call dpotrs( uplo, n, nrhs, a, lda, b, ldb, info )
call cpotrs( uplo, n, nrhs, a, lda, b, ldb, info )
call zpotrs( uplo, n, nrhs, a, lda, b, ldb, info )
```

Fortran 95:

```
call potrs( a, b [,uplo] [, info] )
```

Description

This routine solves for X the system of linear equations $A^*X = B$ with a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A , given the Cholesky factorization of A :

$A = U^T * U$ for real data, If $uplo = 'U'$

$A = U^H * U$ for complex

data

$A = L * L^T$ for real data, If $uplo = 'L'$,

$A = L * L^H$ for complex

data

where L is a lower triangular matrix and U is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix B .

Before calling this routine, you must call [?potrf](#) to compute the Cholesky factorization of A .

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
Indicates how the input matrix A has been factored:
If $uplo = 'U'$, the upper triangle of A is stored.

	If <i>uplo</i> = 'L', the lower triangle of <i>A</i> is stored.
<i>n</i>	INTEGER. The order of matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides ($nrhs \geq 0$).
<i>a</i> , <i>b</i>	REAL for <i>spotrs</i> DOUBLE PRECISION for <i>dpotrs</i> COMPLEX for <i>cpotrs</i> DOUBLE COMPLEX for <i>zpotrs</i> . Arrays: <i>a</i> (<i>lda</i> ,*), <i>b</i> (<i>ldb</i> ,*). The array <i>a</i> contains the factor <i>U</i> or <i>L</i> (see <i>uplo</i>). The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations. The second dimension of <i>a</i> must be at least $\max(1, n)$, the second dimension of <i>b</i> at least $\max(1, nrhs)$.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; $ldb \geq \max(1, n)$.

Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *potrs* interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>nrhs</i>).
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If `uplo = 'U'`, the computed solution for each right-hand side b is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon |U^H||U|$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

A similar estimate holds for `uplo = 'L'`. If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \text{cond}(A, x) \varepsilon$$

where $\text{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \| |A^{-1}| \|_\infty \| |A| \|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$. The approximate number of floating-point operations for one right-hand side vector b is $2n^2$ for real flavors and $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call `?pocon`.

To refine the solution and estimate the error, call `?porfs`.

?pptrs

Solves a system of linear equations with a packed Cholesky-factored symmetric (Hermitian) positive-definite matrix.

Syntax

FORTRAN 77:

```
call spptrs( uplo, n, nrhs, ap, b, ldb, info )
call dpptrs( uplo, n, nrhs, ap, b, ldb, info )
call cpptrs( uplo, n, nrhs, ap, b, ldb, info )
call zpptrs( uplo, n, nrhs, ap, b, ldb, info )
```

Fortran 95:

```
call pptrs( a, b [,uplo] [,info] )
```

Description

This routine solves for x the system of linear equations $A * X = B$ with a packed symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A , given the Cholesky factorization of A :

$A = U^T * U$ for real data, If $uplo = 'U'$

$A = U^H * U$ for complex

data

$A = L * L^T$ for real data, If $uplo = 'L'$,

$A = L * L^H$ for complex

data

where L is a lower triangular matrix and U is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix B .

Before calling this routine, you must call [?pptrf](#) to compute the Cholesky factorization of A .

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If $uplo = 'U'$, the upper triangle of A is stored. If $uplo = 'L'$, the lower triangle of A is stored.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides ($nrhs \geq 0$).
<i>ap, b</i>	REAL for <code>spptrs</code> DOUBLE PRECISION for <code>dpptsr</code> COMPLEX for <code>cpptrs</code> DOUBLE COMPLEX for <code>zpptrs</code> . Arrays: $ap(*)$, $b(l\delta b,*)$ The dimension of ap must be at least $\max(1, n(n+1)/2)$. The array ap contains the factor U or L , as specified by $uplo$, in <i>packed storage</i> (see Matrix Storage Schemes).

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.

ldb INTEGER. The first dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

b Overwritten by the solution matrix x .

$info$ INTEGER. If $info = 0$, the execution is successful.
If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `pptrs` interface are as follows:

a Stands for argument ap in Fortran 77 interface. Holds the array A of size $(n*(n+1)/2)$.
 b Holds the matrix B of size $(n, nrhs)$.
 $uplo$ Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If $uplo = 'U'$, the computed solution for each right-hand side b is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\epsilon \|U^H\| \|U\|$$

$c(n)$ is a modest linear function of n , and ϵ is the machine precision.

A similar estimate holds for $uplo = 'L'$.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \text{cond}(A, x) \epsilon$$

where $\text{cond}(A, x) = \frac{\|A^{-1}\| \|A\| \|x\|}{\|x\|} \leq \|A^{-1}\| \|A\| = \kappa_{\infty}(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_{\infty}(A)$.

The approximate number of floating-point operations for one right-hand side vector b is $2n^2$ for real flavors and $8n^2$ for complex flavors.

To estimate the condition number $\kappa_{\infty}(A)$, call `?ppcon`.

To refine the solution and estimate the error, call `?pprfs`.

?pbtrs

Solves a system of linear equations with a Cholesky-factored symmetric (Hermitian) positive-definite band matrix.

Syntax

FORTRAN 77:

```
call spbtrs( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call dpbtrs( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call cpbtrs( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call zpbtrs( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
```

Fortran 95:

```
call pbtrs( a, b [,uplo] [,info] )
```

Description

This routine solves for real data a system of linear equations $A^*X = B$ with a symmetric positive-definite or, for complex data, Hermitian positive-definite *band* matrix A , given the Cholesky factorization of A :

$A = U^T * U$ for real data, If $uplo = 'U'$

$A = U^H * U$ for complex

data

$A = L * L^T$ for real data, If *uplo* = 'L',

$A = L * L^H$ for complex

data

where *L* is a lower triangular matrix and *U* is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix *B*.

Before calling this routine, you must call [?pbtrf](#) to compute the Cholesky factorization of *A* in the band storage form.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix <i>A</i> has been factored: If <i>uplo</i> = 'U', the upper triangular factor is stored in <i>ab</i> . If <i>uplo</i> = 'L', the lower triangular factor is stored in <i>ab</i> .
<i>n</i>	INTEGER. The order of matrix <i>A</i> ; $n \geq 0$.
<i>kd</i>	INTEGER. The number of superdiagonals or subdiagonals in the matrix <i>A</i> ; $kd \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ab, b</i>	REAL for <i>spbtrs</i> DOUBLE PRECISION for <i>dpbtrs</i> COMPLEX for <i>cpbtrs</i> DOUBLE COMPLEX for <i>zpbtrs</i> . Arrays: <i>ab</i> (<i>ldab</i> ,*), <i>b</i> (<i>ldb</i> ,*). The array <i>ab</i> contains the Cholesky factor, as returned by the factorization routine, in <i>band storage</i> form. The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations. The second dimension of <i>ab</i> must be at least $\max(1, n)$, and the second dimension of <i>b</i> at least $\max(1, nrhs)$.
<i>ldab</i>	INTEGER. The first dimension of the array <i>ab</i> ; $ldab \geq kd + 1$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; $ldb \geq \max(1, n)$.

Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>x</i> .

info INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `pbtrs` interface are as follows:

a Stands for argument *ab* in Fortran 77 interface. Holds the array *A* of size $(kd+1, n)$.
b Holds the matrix *B* of size $(n, nrhs)$.
uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(kd + 1)\varepsilon P|U^H||U| \text{ or } |E| \leq c(kd + 1)\varepsilon P|L^H||L|$$

$c(k)$ is a modest linear function of *k*, and ε is the machine precision.

If x_0 is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(kd + 1) \text{cond}(A, x)\varepsilon$$

where $\text{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \| |A^{-1}| \|_\infty \| |A| \|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector is $4n*kd$ for real flavors and $16n*kd$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call [?pbcon](#).

To refine the solution and estimate the error, call [?pbrfs](#).

?pttrs

Solves a system of linear equations with a symmetric (Hermitian) positive-definite tridiagonal matrix using the factorization computed by ?pttrf.

Syntax

FORTRAN 77:

```
call spttrs( n, nrhs, d, e, b, ldb, info )
call dpttrs( n, nrhs, d, e, b, ldb, info )
call cpttrs( uplo, n, nrhs, d, e, b, ldb, info )
call zpttrs( uplo, n, nrhs, d, e, b, ldb, info )
```

Fortran 95:

```
call pttrs( d, e, b [,info] )
call pttrs( d, e, b [,uplo] [,info] )
```

Description

This routine solves for X a system of linear equations $A*X = B$ with a symmetric (Hermitian) positive-definite tridiagonal matrix A . Before calling this routine, call [?pttrf](#) to compute the $L*D*L'$ for real data and the $L*D*L'$ or $U'*D*U$ factorization of A for complex data.

Input Parameters

<i>uplo</i>	CHARACTER*1. Used for <code>cpttrs/zpttrs</code> only. Must be 'U' or 'L'. Specifies whether the superdiagonal or the subdiagonal of the tridiagonal matrix A is stored and how A is factored: If <i>uplo</i> = 'U', the array <i>e</i> stores the superdiagonal of A , and A is factored as $U'*D*U$. If <i>uplo</i> = 'L', the array <i>e</i> stores the subdiagonal of A , and A is factored as $L*D*L'$.
<i>n</i>	INTEGER. The order of A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, that is, the number of columns of the matrix B ; $nrhs \geq 0$.
<i>d</i>	REAL for <code>spttrs, cpttrs</code>

DOUBLE PRECISION for `dpttrs`, `zpttrs`.
 Array, dimension (n). Contains the diagonal elements of the diagonal matrix D from the factorization computed by [?pttrf](#).

`e`, `b` REAL for `spttrs`
 DOUBLE PRECISION for `dpttrs`
 COMPLEX for `cpttrs`
 DOUBLE COMPLEX for `zpttrs`.
 Arrays: `e`($n-1$), `b`(`ldb`, `nrhs`).
 The array `e` contains the ($n-1$) off-diagonal elements of the unit bidiagonal factor U or L from the factorization computed by [?pttrf](#) (see `uplo`).
 The array `b` contains the matrix B whose columns are the right-hand sides for the systems of equations.

`ldb` INTEGER. The leading dimension of `b`; $ldb \geq \max(1, n)$.

Output Parameters

`b` Overwritten by the solution matrix X .

`info` INTEGER. If `info=0`, the execution is successful.
 If `info = -i`, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `pttrs` interface are as follows:

`d` Holds the vector of length (n).

`e` Holds the vector of length ($n-1$).

`b` Holds the matrix B of size (n , `nrhs`).

`uplo` Used in complex flavors only. Must be 'U' or 'L'. The default value is 'U'.

?sytrs

Solves a system of linear equations with a UDU- or LDL-factored symmetric matrix.

Syntax

FORTRAN 77:

```
call ssytrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call dsytrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call csytrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call zsytrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
```

Fortran 95:

```
call sytrs( a, b, ipiv [,uplo] [,info] )
```

Description

This routine solves for x the system of linear equations $A \cdot X = B$ with a symmetric matrix A , given the Bunch-Kaufman factorization of A :

if $uplo = 'U'$, $A = P \cdot U \cdot D \cdot U^T \cdot P^T$
 if $uplo = 'L'$, $A = P \cdot L \cdot D \cdot L^T \cdot P^T$,

where P is a permutation matrix, U and L are upper and lower triangular matrices with unit diagonal, and D is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix B . You must supply to this routine the factor U (or L) and the array $ipiv$ returned by the factorization routine [?sytrf](#).

Input Parameters

$uplo$	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If $uplo = 'U'$, the array a stores the upper triangular factor U of the factorization $A = P \cdot U \cdot D \cdot U^T \cdot P^T$. If $uplo = 'L'$, the array a stores the lower triangular factor L of the factorization $A = P \cdot L \cdot D \cdot L^T \cdot P^T$.
n	INTEGER. The order of matrix A ; $n \geq 0$.

<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?sytrf .
<i>a, b</i>	REAL for <i>ssytrs</i> DOUBLE PRECISION for <i>dsytrs</i> COMPLEX for <i>csytrs</i> DOUBLE COMPLEX for <i>zsytrs</i> . Arrays: <i>a</i> (<i>lda</i> ,*), <i>b</i> (<i>ldb</i> ,*). The array <i>a</i> contains the factor <i>U</i> or <i>L</i> (see <i>uplo</i>). The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the system of equations. The second dimension of <i>a</i> must be at least $\max(1, n)$, and the second dimension of <i>b</i> at least $\max(1, nrhs)$.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; $ldb \geq \max(1, n)$.

Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *sytrs* interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>nrhs</i>).
<i>ipiv</i>	Holds the vector of length (<i>n</i>).
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For each right-hand side b , the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon P|U||D||U^T|P^T \text{ or } |E| \leq c(n)\varepsilon P|L||D||U^T|P^T$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \text{cond}(A, x)\varepsilon$$

where $\text{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \| |A^{-1}| \|_\infty \| |A| \|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$.

The total number of floating-point operations for one right-hand side vector is approximately $2n^2$ for real flavors or $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call [?sycon](#).

To refine the solution and estimate the error, call [?sytrfs](#).

?hetrs

Solves a system of linear equations with a UDU- or LDL-factored Hermitian matrix.

Syntax

FORTRAN 77:

```
call chetrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call zhetrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
```

Fortran 95:

```
call hetrs( a, b, ipiv [, uplo] [,info] )
```

Description

This routine solves for X the system of linear equations $A * X = B$ with a Hermitian matrix A , given the Bunch-Kaufman factorization of A :

if $uplo = 'U'$ $A = P * U * D * U^H * P^T$
 if $uplo = 'L'$ $A = P * L * D * L^H * P^T,$

where P is a permutation matrix, U and L are upper and lower triangular matrices with unit diagonal, and D is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix B . You must supply to this routine the factor U (or L) and the array $ipiv$ returned by the factorization routine [?hetrf](#).

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
 Indicates how the input matrix A has been factored:
 If $uplo = 'U'$, the array a stores the upper triangular factor U of the factorization $A = P * U * D * U^H * P^T$.
 If $uplo = 'L'$, the array a stores the lower triangular factor L of the factorization $A = P * L * D * L^H * P^T$.

n INTEGER. The order of matrix A ; $n \geq 0$.

nrhs INTEGER. The number of right-hand sides; $nrhs \geq 0$.

ipiv INTEGER.
 Array, DIMENSION at least $\max(1, n)$.
 The *ipiv* array, as returned by [?hetrf](#).

a, b COMPLEX for *chetrs*
 DOUBLE COMPLEX for *zhetsr*.
 Arrays: $a(lda, *)$, $b(ldb, *)$.
 The array a contains the factor U or L (see *uplo*).
 The array b contains the matrix B whose columns are the right-hand sides for the system of equations.
 The second dimension of a must be at least $\max(1, n)$, the second dimension of b at least $\max(1, nrhs)$.

lda INTEGER. The first dimension of a ; $lda \geq \max(1, n)$.

ldb INTEGER. The first dimension of b ; $ldb \geq \max(1, nrhs)$.

Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>x</i> .
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hetrs` interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>nrhs</i>).
<i>ipiv</i>	Holds the vector of length (<i>n</i>).
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon P|U||D||U^H|P^T \text{ or } |E| \leq c(n)\varepsilon P|L||D||L^H|P^T$$

$c(n)$ is a modest linear function of *n*, and ε is the machine precision.

If x_0 is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \text{cond}(A, x)\varepsilon$$

where $\text{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \| |A^{-1}| \|_\infty \| |A| \|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$.

The total number of floating-point operations for one right-hand side vector is approximately $8n^2$.

To estimate the condition number $\kappa_{\infty}(A)$, call [?hecon](#).

To refine the solution and estimate the error, call [?herfs](#).

?sptrs

Solves a system of linear equations with a UDU- or LDL-factored symmetric matrix using packed storage.

Syntax

FORTRAN 77:

```
call ssptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call dsptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call csptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call zsptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
```

Fortran 95:

```
call sptrs( a, b, ipiv [, uplo] [,info] )
```

Description

This routine solves for x the system of linear equations $A * X = B$ with a symmetric matrix A , given the Bunch-Kaufman factorization of A :

if $uplo='U'$, $A = P U D U^T P^T$
 if $uplo='L'$, $A = P L D L^T P^T$,

where P is a permutation matrix, U and L are upper and lower *packed* triangular matrices with unit diagonal, and D is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix B . You must supply the factor U (or L) and the array $ipiv$ returned by the factorization routine [?spturf](#).

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
 Indicates how the input matrix A has been factored:

	If $uplo = 'U'$, the array ap stores the packed factor U of the factorization $A = P*U*D*U^T*P^T$. If $uplo = 'L'$, the array ap stores the packed factor L of the factorization $A = P*L*D*L^T*P^T$.
n	INTEGER. The order of matrix A ; $n \geq 0$.
$nrhs$	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
$ipiv$	INTEGER. Array, DIMENSION at least $\max(1, n)$. The $ipiv$ array, as returned by ?spturf.
ap, b	REAL for $ssptrs$ DOUBLE PRECISION for $dsptrs$ COMPLEX for $csptrs$ DOUBLE COMPLEX for $zsptrs$. Arrays: $ap(*), b(l, db, *)$. The dimension of ap must be at least $\max(1, n(n+1)/2)$. The array ap contains the factor U or L, as specified by $uplo$, in <i>packed storage</i> (see Matrix Storage Schemes). The array b contains the matrix B whose columns are the right-hand sides for the system of equations. The second dimension of b must be at least $\max(1, nrhs)$.
ldb	INTEGER. The first dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

b	Overwritten by the solution matrix X .
$info$	INTEGER. If $info=0$, the execution is successful. If $info = -i$, the i-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sptrs` interface are as follows:

a	Stands for argument ap in Fortran 77 interface. Holds the array A of size $(n*(n+1)/2)$.
b	Holds the matrix B of size $(n, nrhs)$.

ipiv Holds the vector of length (*n*).
uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon P|U||D||U^T|P^T \text{ or } |E| \leq c(n)\varepsilon P|L||D||L^T|P^T$$

$c(n)$ is a modest linear function of *n*, and ε is the machine precision.

If x_0 is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \text{cond}(A, x)\varepsilon$$

where $\text{cond}(A, x) = \|A^{-1}\|_\infty \|A\|_\infty \|x\|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$.

The total number of floating-point operations for one right-hand side vector is approximately $2n^2$ for real flavors or $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call [?spcon](#).

To refine the solution and estimate the error, call [?sprfs](#).

?hptrs

Solves a system of linear equations with a UDU- or LDL-factored Hermitian matrix using packed storage.

Syntax

FORTRAN 77:

```
call chptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call zhptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
```

Fortran 95:

```
call hptrs( a, b, ipiv [,uplo] [,info] )
```

Description

This routine solves for X the system of linear equations $A * X = B$ with a Hermitian matrix A , given the Bunch-Kaufman factorization of A :

if $uplo = 'U'$, $A = P * U * D * U^H * P^T$
 if $uplo = 'L'$, $A = P * L * D * L^H * P^T$,

where P is a permutation matrix, U and L are upper and lower *packed* triangular matrices with unit diagonal, and D is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix B .

You must supply to this routine the arrays ap (containing U or L) and $ipiv$ in the form returned by the factorization routine [?hptrf](#).

Input Parameters

$uplo$	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If $uplo = 'U'$, the array ap stores the packed factor U of the factorization $A = P * U * D * U^H * P^T$. If $uplo = 'L'$, the array ap stores the packed factor L of the factorization $A = P * L * D * L^H * P^T$.
n	INTEGER. The order of matrix A ; $n \geq 0$.
$nrhs$	INTEGER. The number of right-hand sides; $nrhs \geq 0$.

<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The <i>ipiv</i> array, as returned by <code>?hptrf</code> .
<i>ap, b</i>	COMPLEX for <code>chptrs</code> DOUBLE COMPLEX for <code>zhptrs</code> . Arrays: <i>ap</i> (*), <i>b</i> (<i>ldb</i> , *). The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$. The array <i>ap</i> contains the factor <i>U</i> or <i>L</i> , as specified by <i>uplo</i> , in <i>packed storage</i> (see Matrix Storage Schemes). The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the system of equations. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; $ldb \geq \max(1, n)$.

Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>x</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hptrs` interface are as follows:

<i>a</i>	Stands for argument <i>ap</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(n*(n+1)/2)$.
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>ipiv</i>	Holds the vector of length (n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\epsilon_P |U| |D| |U^H| P^T \text{ or } |E| \leq c(n)\epsilon_P |L| |D| |L^H| P^T$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \text{cond}(A, x) \varepsilon$$

where $\text{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \| |A^{-1}| \|_\infty \| |A| \|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$.

The total number of floating-point operations for one right-hand side vector is approximately $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call [?hpcon](#).

To refine the solution and estimate the error, call [?hprfs](#).

?trtrs

Solves a system of linear equations with a triangular matrix, with multiple right-hand sides.

Syntax

FORTRAN 77:

```
call strtrs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, info )
call dtrtrs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, info )
call ctrtrs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, info )
call ztrtrs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, info )
```

Fortran 95:

```
call trtrs( a, b [,uplo] [, trans] [,diag] [,info] )
```

Description

This routine solves for X the following systems of linear equations with a triangular matrix A , with multiple right-hand sides stored in B :

$$\begin{aligned} A * X &= B && \text{if } trans = 'N', \\ A^T * X &= B && \text{if } trans = 'T', \\ A^H * X &= B && \text{if } trans = 'C' \text{ (for complex matrices only).} \end{aligned}$$

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
Indicates whether A is upper or lower triangular:
If *uplo* = 'U', then A is upper triangular.
If *uplo* = 'L', then A is lower triangular.

trans CHARACTER*1. Must be 'N' or 'T' or 'C'.
If *trans* = 'N', then $A * X = B$ is solved for X .
If *trans* = 'T', then $A^T * X = B$ is solved for X .
If *trans* = 'C', then $A^H * X = B$ is solved for X .

diag CHARACTER*1. Must be 'N' or 'U'.
If *diag* = 'N', then A is not a unit triangular matrix.
If *diag* = 'U', then A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array a .

n INTEGER. The order of A ; the number of rows in B ; $n \geq 0$.

nrhs INTEGER. The number of right-hand sides; $nrhs \geq 0$.

a, b REAL for strtrs
DOUBLE PRECISION for dtrtrs
COMPLEX for ctrtrs
DOUBLE COMPLEX for ztrtrs.
Arrays: $a(lda,*)$, $b(ldb,*)$.
The array a contains the matrix A .
The array b contains the matrix B whose columns are the right-hand sides for the systems of equations.
The second dimension of a must be at least $\max(1, n)$, the second dimension of b at least $\max(1, nrhs)$.

lda INTEGER. The first dimension of a ; $lda \geq \max(1, n)$.

ldb INTEGER. The first dimension of *b*; $ldb \geq \max(1, n)$.

Output Parameters

b Overwritten by the solution matrix *x*.
info INTEGER. If *info*=0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `trtrs` interface are as follows:

a Stands for argument *ap* in Fortran 77 interface. Holds the matrix *A* of size $(n * (n+1) / 2)$.
b Holds the matrix *B* of size $(n, nrhs)$.
uplo Must be 'U' or 'L'. The default value is 'U'.
trans Must be 'N', 'C', or 'T'. The default value is 'N'.
diag Must be 'N' or 'U'. The default value is 'N'.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon |A|$$

$c(n)$ is a modest linear function of n , and ε is the machine precision. If x_0 is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \operatorname{cond}(A, x) \varepsilon \text{ provided } c(n) \operatorname{cond}(A, x) \varepsilon < 1$$

where $\operatorname{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_{\infty}(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_{\infty}(A)$.

The approximate number of floating-point operations for one right-hand side vector b is n^2 for real flavors and $4n^2$ for complex flavors.

To estimate the condition number $\kappa_{\infty}(A)$, call `?trcon`.

To estimate the error in the solution, call `?trrfs`.

?tptrs

Solves a system of linear equations with a packed triangular matrix, with multiple right-hand sides.

Syntax

FORTRAN 77:

```
call stptrs( uplo, trans, diag, n, nrhs, ap, b, ldb, info )
call dtptrs( uplo, trans, diag, n, nrhs, ap, b, ldb, info )
call ctptrs( uplo, trans, diag, n, nrhs, ap, b, ldb, info )
call ztptrs( uplo, trans, diag, n, nrhs, ap, b, ldb, info )
```

Fortran 95:

```
call tptrs( a, b [,uplo] [, trans] [,diag] [,info] )
```

Description

This routine solves for x the following systems of linear equations with a packed triangular matrix A , with multiple right-hand sides stored in B :

$A * X = B$	if $trans = 'N'$,
$A^T * X = B$	if $trans = 'T'$,
$A^H * X = B$	if $trans = 'C'$ (for complex matrices only).

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
Indicates whether A is upper or lower triangular:

	If <i>uplo</i> = 'U', then <i>A</i> is upper triangular. If <i>uplo</i> = 'L', then <i>A</i> is lower triangular.
<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. If <i>trans</i> = 'N', then $A * X = B$ is solved for <i>X</i> . If <i>trans</i> = 'T', then $A^T * X = B$ is solved for <i>X</i> . If <i>trans</i> = 'C', then $A^H * X = B$ is solved for <i>X</i> .
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. If <i>diag</i> = 'N', then <i>A</i> is not a unit triangular matrix. If <i>diag</i> = 'U', then <i>A</i> is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array <i>ap</i> .
<i>n</i>	INTEGER. The order of <i>A</i> ; the number of rows in <i>B</i> ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ap, b</i>	REAL for <i>stptrs</i> DOUBLE PRECISION for <i>dtptrs</i> COMPLEX for <i>ctptrs</i> DOUBLE COMPLEX for <i>ztptrs</i> . Arrays: <i>ap</i> (*), <i>b</i> (<i>ldb</i> ,*). The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$. The array <i>ap</i> contains the matrix <i>A</i> in <i>packed storage</i> (see Matrix Storage Schemes). The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the system of equations. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; $ldb \geq \max(1, n)$.

Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *tpttrs* interface are as follows:

<i>a</i>	Stands for argument <i>ap</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(n^*(n+1)/2)$.
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon |A|$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \operatorname{cond}(A, x) \varepsilon \text{ provided } c(n) \operatorname{cond}(A, x) \varepsilon < 1$$

where $\operatorname{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \| |A^{-1}| \|_\infty \| |A| \|_\infty = \kappa_\infty(A)$.

Note that $\operatorname{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector *b* is n^2 for real flavors and $4n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call [?tpcon](#).

To estimate the error in the solution, call [?tprfs](#).

?tbtrs

Solves a system of linear equations with a band triangular matrix, with multiple right-hand sides.

Syntax

FORTRAN 77:

```
call stbtrs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, info )
call dtbtrs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, info )
call ctbtrs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, info )
call ztbtrs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, info )
```

Fortran 95:

```
call tbtrs( a, b [,uplo] [, trans] [,diag] [,info] )
```

Description

This routine solves for X the following systems of linear equations with a band triangular matrix A , with multiple right-hand sides stored in B :

$A * X = B$	if $trans = 'N'$,
$A^T * X = B$	if $trans = 'T'$,
$A^H * X = B$	if $trans = 'C'$ (for complex matrices only).

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If $uplo = 'U'$, then A is upper triangular. If $uplo = 'L'$, then A is lower triangular.
<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. If $trans = 'N'$, then $A * X = B$ is solved for X . If $trans = 'T'$, then $A^T * X = B$ is solved for X . If $trans = 'C'$, then $A^H * X = B$ is solved for X .
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. If $diag = 'N'$, then A is not a unit triangular matrix.

If *diag* = 'U', then *A* is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array *ab*.

<i>n</i>	INTEGER. The order of <i>A</i> ; the number of rows in <i>B</i> ; $n \geq 0$.
<i>kd</i>	INTEGER. The number of superdiagonals or subdiagonals in the matrix <i>A</i> ; $kd \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ab, b</i>	REAL for <i>stbtrs</i> DOUBLE PRECISION for <i>dtbtrs</i> COMPLEX for <i>ctbtrs</i> DOUBLE COMPLEX for <i>ztbtrs</i> . Arrays: <i>ab</i> (<i>ldab</i> ,*), <i>b</i> (<i>ldb</i> ,*). The array <i>ab</i> contains the matrix <i>A</i> in <i>band storage</i> form. The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations. The second dimension of <i>ab</i> must be at least $\max(1, n)$, the second dimension of <i>b</i> at least $\max(1, nrhs)$.
<i>ldab</i>	INTEGER. The first dimension of <i>ab</i> ; $ldab \geq kd + 1$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; $ldb \geq \max(1, n)$.

Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>x</i> .
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *tbtrs* interface are as follows:

<i>a</i>	Stands for the argument <i>ab</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(kd+1, n)$
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

trans Must be 'N', 'C', or 'T'. The default value is 'N'.
diag Must be 'N' or 'U'. The default value is 'N'.

Application Notes

For each right-hand side b , the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon|A|$$

$c(n)$ is a modest linear function of n , and ε is the machine precision. If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \text{cond}(A, x)\varepsilon \text{ provided } c(n) \text{cond}(A, x)\varepsilon < 1$$

where $\text{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector b is $2n \times kd$ for real flavors and $8n \times kd$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call [?tbcon](#).

To estimate the error in the solution, call [?tbrfs](#).

Routines for Estimating the Condition Number

This section describes the LAPACK routines for estimating the *condition number* of a matrix. The condition number is used for analyzing the errors in the solution of a system of linear equations (see [Error Analysis](#)). Since the condition number may be arbitrarily large when the matrix is nearly singular, the routines actually compute the *reciprocal* condition number.

?gecon

Estimates the reciprocal of the condition number of a general matrix in the 1-norm or the infinity-norm.

Syntax

FORTRAN 77:

```
call sgecon( norm, n, a, lda, anorm, rcond, work, iwork, info )
call dgecon( norm, n, a, lda, anorm, rcond, work, iwork, info )
call cgecon( norm, n, a, lda, anorm, rcond, work, rwork, info )
call zgecon( norm, n, a, lda, anorm, rcond, work, rwork, info )
```

Fortran 95:

```
call gecon( a, anorm, rcond [,norm] [,info] )
```

Description

This routine estimates the reciprocal of the condition number of a general matrix A in the 1-norm or infinity-norm:

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$$

$$\kappa_\infty(A) = \|A\|_\infty \|A^{-1}\|_\infty = \kappa_1(A^T) = \kappa_1(A^H).$$

Before calling this routine:

- compute *anorm* (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)
- call [?getrf](#) to compute the *LU* factorization of A .

Input Parameters

norm CHARACTER*1. Must be '1' or 'O' or 'I'.
 If *norm* = '1' or 'O', then the routine estimates the condition number of matrix A in 1-norm.
 If *norm* = 'I', then the routine estimates the condition number of matrix A in infinity-norm.

n INTEGER. The order of the matrix A ; $n \geq 0$.

<i>a, work</i>	REAL for sgecon DOUBLE PRECISION for dgecon COMPLEX for cgecon DOUBLE COMPLEX for zgecon. Arrays: $a(lda, *)$, $work(*)$. The array <i>a</i> contains the <i>LU</i>-factored matrix <i>A</i>, as returned by ?getrf. The second dimension of <i>a</i> must be at least $\max(1, n)$. The array <i>work</i> is a workspace for the routine. The dimension of <i>work</i> must be at least $\max(1, 4*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>anorm</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. The norm of the <i>original</i> matrix <i>A</i> (see Description).
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.
<i>rwork</i>	REAL for cgecon DOUBLE PRECISION for zgecon. Workspace array, DIMENSION at least $\max(1, 2*n)$.

Output Parameters

<i>rcond</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets $rcond = 0$ if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <i>rcond</i> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
<i>info</i>	INTEGER. If $info=0$, the execution is successful. If $info = -i$, the <i>i</i>-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *gecon* interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (n, n) .

norm

Must be '1', 'O', or 'I'. The default value is '1'.

Application Notes

The computed $rcond$ is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A*x = b$ or $A^H*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2*n^2$ floating-point operations for real flavors and $8*n^2$ for complex flavors.

?gbcon

Estimates the reciprocal of the condition number of a band matrix in the 1-norm or the infinity-norm.

Syntax

FORTRAN 77:

```
call sgbcon( norm, n, kl, ku, ab, ldab, ipiv, anorm, rcond, work, iwork, info
)
call dgbcon( norm, n, kl, ku, ab, ldab, ipiv, anorm, rcond, work, iwork, info
)
call cgbcon( norm, n, kl, ku, ab, ldab, ipiv, anorm, rcond, work, rwork, info
)
call zgbcon( norm, n, kl, ku, ab, ldab, ipiv, anorm, rcond, work, rwork, info
)
```

Fortran 95:

```
call gbcon( a, ipiv, anorm, rcond [,kl] [,norm] [,info] )
```

Description

This routine estimates the reciprocal of the condition number of a general band matrix A in the 1-norm or infinity-norm:

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$$

$$\kappa_\infty(A) = \|A\|_\infty \|A^{-1}\|_\infty = \kappa_1(A^T) = \kappa_1(A^H).$$

Before calling this routine:

- compute *anorm* (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)
- call `?gbtrf` to compute the *LU* factorization of *A*.

Input Parameters

<i>norm</i>	CHARACTER*1. Must be '1' or 'O' or 'I'. If <i>norm</i> = '1' or 'O', then the routine estimates the condition number of matrix <i>A</i> in 1-norm. If <i>norm</i> = 'I', then the routine estimates the condition number of matrix <i>A</i> in infinity-norm.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ; $n \geq 0$.
<i>kl</i>	INTEGER. The number of subdiagonals within the band of <i>A</i> ; $kl \geq 0$.
<i>ku</i>	INTEGER. The number of superdiagonals within the band of <i>A</i> ; $ku \geq 0$.
<i>ldab</i>	INTEGER. The first dimension of the array <i>ab</i> . ($ldab \geq 2*kl + ku + 1$).
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The <i>ipiv</i> array, as returned by <code>?gbtrf</code> .
<i>ab, work</i>	REAL for sgbcon DOUBLE PRECISION for dgbcon COMPLEX for cgbcon DOUBLE COMPLEX for zgbcon. Arrays: <i>ab</i>(<i>ldab</i>,*), <i>work</i>(*). The array <i>ab</i> contains the factored band matrix <i>A</i>, as returned by <code>?gbtrf</code>. The second dimension of <i>ab</i> must be at least $\max(1, n)$. The array <i>work</i> is a workspace for the routine. The dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>anorm</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. The norm of the <i>original</i> matrix <i>A</i> (see Description).
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.
<i>rwork</i>	REAL for cgbcon DOUBLE PRECISION for zgbcon. Workspace array, DIMENSION at least $\max(1, 2*n)$.

Output Parameters

<i>rcond</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets <i>rcond</i> = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <i>rcond</i> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
<i>info</i>	INTEGER. If <i>info</i>=0, the execution is successful. If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *gbcon* interface are as follows:

<i>a</i>	Stands for argument <i>ab</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(2*k_l + k_u + 1, n)$.
<i>ipiv</i>	Holds the vector of length (<i>n</i>).
<i>norm</i>	Must be '1', 'O', or 'I'. The default value is '1'.
<i>kl</i>	If omitted, assumed $k_l = k_u$.
<i>ku</i>	Restored as $k_u = lda - 2*k_l - 1$.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A*x = b$ or $A^H*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n(k_u + 2k_l)$ floating-point operations for real flavors and $8n(k_u + 2k_l)$ for complex flavors.

?gtcon

Estimates the reciprocal of the condition number of a tridiagonal matrix using the factorization computed by ?gttrf.

Syntax

FORTRAN 77:

```
call sgtcon( norm, n, dl, d, du, du2, ipiv, anorm, rcond, work, iwork, info
)
call dgtcon( norm, n, dl, d, du, du2, ipiv, anorm, rcond, work, iwork, info
)
call cgtcon( norm, n, dl, d, du, du2, ipiv, anorm, rcond, work, info )
call zgtcon( norm, n, dl, d, du, du2, ipiv, anorm, rcond, work, info )
```

Fortran 95:

```
call gtcon( dl, d, du, du2, ipiv, anorm, rcond [,norm] [,info] )
```

Description

This routine estimates the reciprocal of the condition number of a real or complex tridiagonal matrix A in the 1-norm or infinity-norm:

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1$$

$$\kappa_\infty(A) = \|A\|_\infty \|A^{-1}\|_\infty$$

An estimate is obtained for $\|A^{-1}\|$, and the reciprocal of the condition number is computed as $rcond = 1 / (\|A\| \|A^{-1}\|)$.

Before calling this routine:

- compute *anorm* (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)
- call ?gttrf to compute the *LU* factorization of A .

Input Parameters

norm CHARACTER*1. Must be '1' or 'O' or 'I'.

If $norm = '1'$ or $'O'$, then the routine estimates the condition number of matrix A in 1-norm.

If $norm = 'I'$, then the routine estimates the condition number of matrix A in infinity-norm.

n	INTEGER. The order of the matrix A ; $n \geq 0$.
$dl, d, du, du2$	REAL for sgtcon DOUBLE PRECISION for dgtcon COMPLEX for cgtcon DOUBLE COMPLEX for zgtcon. Arrays: $dl(n-1)$, $d(n)$, $du(n-1)$, $du2(n-2)$. The array dl contains the $(n-1)$ multipliers that define the matrix L from the LU factorization of A as computed by ?gttrf. The array d contains the n diagonal elements of the upper triangular matrix U from the LU factorization of A. The array du contains the $(n-1)$ elements of the first superdiagonal of U. The array $du2$ contains the $(n-2)$ elements of the second superdiagonal of U.
$ipiv$	INTEGER. Array, DIMENSION (n). The array of pivot indices, as returned by ?gttrf.
$anorm$	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. The norm of the <i>original</i> matrix A (see <i>Description</i>).
$work$	REAL for sgtcon DOUBLE PRECISION for dgtcon COMPLEX for cgtcon DOUBLE COMPLEX for zgtcon. Workspace array, DIMENSION $(2*n)$.
$iwork$	INTEGER. Workspace array, DIMENSION (n) . Used for real flavors only.

Output Parameters

$rcond$	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal of the condition number. The routine sets $rcond=0$ if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime $rcond$ is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gtcon` interface are as follows:

<i>dl</i>	Holds the vector of length ($n-1$).
<i>d</i>	Holds the vector of length (n).
<i>du</i>	Holds the vector of length ($n-1$).
<i>du2</i>	Holds the vector of length ($n-2$).
<i>ipiv</i>	Holds the vector of length (n).
<i>norm</i>	Must be '1', 'O', or 'I'. The default value is '1'.

Application Notes

The computed $rcond$ is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A^*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

?pocon

Estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite matrix.

Syntax

FORTRAN 77:

```
call spocon( uplo, n, a, lda, anorm, rcond, work, iwork, info )
call dpocon( uplo, n, a, lda, anorm, rcond, work, iwork, info )
call cpocon( uplo, n, a, lda, anorm, rcond, work, rwork, info )
call zpocon( uplo, n, a, lda, anorm, rcond, work, rwork, info )
```

Fortran 95:

```
call pocon( a, anorm, rcond [,uplo] [,info] )
```

Description

This routine estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite matrix A :

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \text{ (since } A \text{ is symmetric or Hermitian, } \kappa_\infty(A) = \kappa_1(A)\text{)}.$$

Before calling this routine:

- compute *anorm* (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)
- call [?potrf](#) to compute the Cholesky factorization of A .

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>a, work</i>	REAL for spocon DOUBLE PRECISION for dpocon COMPLEX for cpocon

DOUBLE COMPLEX for `zpocon`.

Arrays: `a(lda,*)`, `work(*)`.

The array `a` contains the factored matrix `A`, as returned by `?potrf`. The second dimension of `a` must be at least $\max(1, n)$.

The array `work` is a workspace for the routine. The dimension of `work` must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

`lda`

INTEGER. The first dimension of `a`; $lda \geq \max(1, n)$.

`anorm`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

The norm of the *original* matrix `A` (see *Description*).

`iwork`

INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.

`rwork`

REAL for `cpocon`

DOUBLE PRECISION for `zpocon`.

Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

`rcond`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal of the condition number. The routine sets `rcond` = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime `rcond` is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

`info`

INTEGER. If `info` = 0, the execution is successful.

If `info` = `-i`, the `i`-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `pocon` interface are as follows:

`a`

Holds the matrix `A` of size (n, n) .

`uplo`

Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed $rcond$ is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A^*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

?ppcon

Estimates the reciprocal of the condition number of a packed symmetric (Hermitian) positive-definite matrix.

Syntax

FORTRAN 77:

```
call sppcon( uplo, n, ap, anorm, rcond, work, iwork, info )
call dppcon( uplo, n, ap, anorm, rcond, work, iwork, info )
call cppcon( uplo, n, ap, anorm, rcond, work, rwork, info )
call zppcon( uplo, n, ap, anorm, rcond, work, rwork, info )
```

Fortran 95:

```
call ppcon( a, anorm, rcond [,uplo] [,info] )
```

Description

This routine estimates the reciprocal of the condition number of a packed symmetric (Hermitian) positive-definite matrix A :

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \text{ (since } A \text{ is symmetric or Hermitian, } \kappa_\infty(A) = \kappa_1(A) \text{)}.$$

Before calling this routine:

- compute $anorm$ (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)
- call `?pptrf` to compute the Cholesky factorization of A .

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.

	Indicates how the input matrix <i>A</i> has been factored: If <i>uplo</i> = 'U', the upper triangle of <i>A</i> is stored. If <i>uplo</i> = 'L', the lower triangle of <i>A</i> is stored.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ; $n \geq 0$.
<i>ap, work</i>	REAL for sppcon DOUBLE PRECISION for dppcon COMPLEX for cppcon DOUBLE COMPLEX for zppcon. Arrays: <i>ap</i> (*), <i>work</i> (*). The array <i>ap</i> contains the packed factored matrix <i>A</i> , as returned by ?pptrf . The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$. The array <i>work</i> is a workspace for the routine. The dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>anorm</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. The norm of the <i>original</i> matrix <i>A</i> (see <i>Description</i>).
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.
<i>rwork</i>	REAL for cppcon DOUBLE PRECISION for zppcon. Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>rcond</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets <i>rcond</i> =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <i>rcond</i> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ppcon` interface are as follows:

<code>a</code>	Stands for the argument <code>ap</code> in Fortran 77 interface. Holds the array <code>A</code> of size $(n*(n+1)/2)$.
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed `rcond` is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

?pbcon

Estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite band matrix.

Syntax

FORTRAN 77:

```
call spbcon( uplo, n, kd, ab, ldab, anorm, rcond, work, iwork, info )
call dpbcon( uplo, n, kd, ab, ldab, anorm, rcond, work, iwork, info )
call cpbcon( uplo, n, kd, ab, ldab, anorm, rcond, work, rwork, info )
call zpbcon( uplo, n, kd, ab, ldab, anorm, rcond, work, rwork, info )
```

Fortran 95:

```
call pbcon( a, anorm, rcond [,uplo] [,info] )
```

Description

This routine estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite band matrix `A`:

$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1$ (since A is symmetric or Hermitian, $\kappa_\infty(A) = \kappa_1(A)$).

Before calling this routine:

- compute *anorm* (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)
- call [?pbtrf](#) to compute the Cholesky factorization of A .

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', the upper triangular factor is stored in <i>ab</i> . If <i>uplo</i> = 'L', the lower triangular factor is stored in <i>ab</i> .
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>kd</i>	INTEGER. The number of superdiagonals or subdiagonals in the matrix A ; $kd \geq 0$.
<i>ldab</i>	INTEGER. The first dimension of the array <i>ab</i> . ($ldab \geq kd + 1$).
<i>ab, work</i>	REAL for spbcon DOUBLE PRECISION for dpbcon COMPLEX for cpbcon DOUBLE COMPLEX for zpbcon. Arrays: <i>ab</i> (<i>ldab</i> ,*), <i>work</i> (*). The array <i>ab</i> contains the factored matrix A in band form, as returned by ?pbtrf . The second dimension of <i>ab</i> must be at least $\max(1, n)$. The array <i>work</i> is a workspace for the routine. The dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>anorm</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. The norm of the <i>original</i> matrix A (see <i>Description</i>).
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.
<i>rwork</i>	REAL for cpbcon DOUBLE PRECISION for zpbcon. Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>rcond</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets <i>rcond</i> = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <i>rcond</i> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `pbcon` interface are as follows:

<i>a</i>	Stands for argument <i>ab</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(kd+1, n)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $4*n(kd + 1)$ floating-point operations for real flavors and $16*n(kd + 1)$ for complex flavors.

?ptcon

Estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite tridiagonal matrix.

Syntax

FORTRAN 77:

```
call sptcon( n, d, e, anorm, rcond, work, info )
call dptcon( n, d, e, anorm, rcond, work, info )
call cptcon( n, d, e, anorm, rcond, work, info )
call zptcon( n, d, e, anorm, rcond, work, info )
```

Fortran 95:

```
call ptcon( d, e, anorm, rcond [,info] )
```

Description

This routine computes the reciprocal of the condition number (in the 1-norm) of a real symmetric or complex Hermitian positive-definite tridiagonal matrix using the factorization $A = L * D * L^T$ for real flavors and $A = L * D * L^H$ for complex flavors or $A = U^T * D * U$ for real flavors and $A = U^H * D * U$ for complex flavors computed by [?pttrf](#) :

$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1$ (since A is symmetric or Hermitian, $\kappa_\infty(A) = \kappa_1(A)$).

The norm $\|A^{-1}\|_1$ is computed by a direct method, and the reciprocal of the condition number is computed as $rcond = 1 / (\|A\|_1 \|A^{-1}\|_1)$.

Before calling this routine:

- compute *anorm* as $\|A\|_1 = \max_j \sum_i |a_{ij}|$
- call [?pttrf](#) to compute the factorization of A .

Input Parameters

<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>d, work</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

Arrays, dimension (n).
 The array d contains the n diagonal elements of the diagonal matrix D from the factorization of A , as computed by [?pttrf](#) ;
 $work$ is a workspace array.

e REAL for `sptcon`
 DOUBLE PRECISION for `dptcon`
 COMPLEX for `cptcon`
 DOUBLE COMPLEX for `zptcon`.
 Array, DIMENSION ($n - 1$).
 Contains off-diagonal elements of the unit bidiagonal factor U or L from the factorization computed by [?pttrf](#) .

anorm REAL for single precision flavors.
 DOUBLE PRECISION for double precision flavors.
 The 1- norm of the *original* matrix A (see *Description*).

Output Parameters

rcond REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 An estimate of the reciprocal of the condition number. The routine sets $rcond = 0$ if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime $rcond$ is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info INTEGER.
 If $info = 0$, the execution is successful.
 If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gtcon` interface are as follows:

d Holds the vector of length (n).
e Holds the vector of length ($n-1$).

Application Notes

The computed $rcond$ is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $4*n(kd + 1)$ floating-point operations for real flavors and $16*n(kd + 1)$ for complex flavors.

?sycon

Estimates the reciprocal of the condition number of a symmetric matrix.

Syntax

FORTRAN 77:

```
call ssycon( uplo, n, a, lda, ipiv, anorm, rcond, work, iwork, info )
call dsycon( uplo, n, a, lda, ipiv, anorm, rcond, work, iwork, info )
call csycon( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
call zsycon( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
```

Fortran 95:

```
call sycon( a, ipiv, anorm, rcond [,uplo] [,info] )
```

Description

This routine estimates the reciprocal of the condition number of a symmetric matrix A :

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \text{ (since } A \text{ is symmetric, } \kappa_\infty(A) = \kappa_1(A) \text{)}.$$

Before calling this routine:

- compute $anorm$ (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)
- call [?sytrf](#) to compute the factorization of A .

Input Parameters

$uplo$ CHARACTER*1. Must be 'U' or 'L'.

Indicates how the input matrix A has been factored:

If $uplo = 'U'$, the array a stores the upper triangular factor U of the factorization $A = P*U*D*U^T*P^T$.

If $uplo = 'L'$, the array a stores the lower triangular factor L of the factorization $A = P*L*D*L^T*P^T$.

n	INTEGER. The order of matrix A ; $n \geq 0$.
$a, work$	REAL for ssycon DOUBLE PRECISION for dsycon COMPLEX for csycon DOUBLE COMPLEX for zsycon. Arrays: $a(lda, *)$, $work(*)$. The array a contains the factored matrix A , as returned by ?sytrf . The second dimension of a must be at least $\max(1, n)$. The array $work$ is a workspace for the routine. The dimension of $work$ must be at least $\max(1, 2*n)$.
lda	INTEGER. The first dimension of a ; $lda \geq \max(1, n)$.
$ipiv$	INTEGER. Array, DIMENSION at least $\max(1, n)$. The array $ipiv$, as returned by ?sytrf .
$anorm$	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. The norm of the <i>original</i> matrix A (see <i>Description</i>).
$iwork$	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

$rcond$	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets $rcond = 0$ if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime $rcond$ is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sycon` interface are as follows:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>ipiv</code>	Holds the vector of length (n) .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed $rcond$ is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A \cdot x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

?hecon

Estimates the reciprocal of the condition number of a Hermitian matrix.

Syntax

FORTRAN 77:

```
call checon( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
call zhecon( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
```

Fortran 95:

```
call hecon( a, ipiv, anorm, rcond [,uplo] [,info] )
```

Description

This routine estimates the reciprocal of the condition number of a Hermitian matrix A :

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \text{ (since } A \text{ is Hermitian, } \kappa_\infty(A) = \kappa_1(A) \text{)}.$$

Before calling this routine:

- compute *anorm* (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)
- call `?hetrf` to compute the factorization of *A*.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix <i>A</i> has been factored: If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular factor <i>U</i> of the factorization $A = P*U*D*U^H*P^T$. If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular factor <i>L</i> of the factorization $A = P*L*D*L^H*P^T$.
<i>n</i>	INTEGER. The order of matrix <i>A</i>; $n \geq 0$.
<i>a, work</i>	COMPLEX for <code>checon</code> DOUBLE COMPLEX for <code>zhecon</code>. Arrays: <i>a</i>(<i>lda</i>,*), <i>work</i>(*). The array <i>a</i> contains the factored matrix <i>A</i>, as returned by <code>?hetrf</code>. The second dimension of <i>a</i> must be at least $\max(1, n)$. The array <i>work</i> is a workspace for the routine. The dimension of <i>work</i> must be at least $\max(1, 2*n)$.
<i>lda</i>	INTEGER. The first dimension of <i>a</i>; $lda \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The array <i>ipiv</i>, as returned by <code>?hetrf</code>.
<i>anorm</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. The norm of the <i>original</i> matrix <i>A</i> (see <i>Description</i>).

Output Parameters

<i>rcond</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets <i>rcond</i> = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <i>rcond</i> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hecon` interface are as follows:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>ipiv</code>	Holds the vector of length (n) .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed $rcond$ is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A \cdot x = b$; the number is usually 5 and never more than 11. Each solution requires approximately $8n^2$ floating-point operations.

?spcon

Estimates the reciprocal of the condition number of a packed symmetric matrix.

Syntax

FORTRAN 77:

```
call sspcon( uplo, n, ap, ipiv, anorm, rcond, work, iwork, info )
call dspcon( uplo, n, ap, ipiv, anorm, rcond, work, iwork, info )
call cspcon( uplo, n, ap, ipiv, anorm, rcond, work, info )
call zspcon( uplo, n, ap, ipiv, anorm, rcond, work, info )
```

Fortran 95:

```
call spcon( a, ipiv, anorm, rcond [,uplo] [,info] )
```

Description

This routine estimates the reciprocal of the condition number of a packed symmetric matrix A :

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \text{ (since } A \text{ is symmetric, } \kappa_\infty(A) = \kappa_1(A) \text{)}.$$

Before calling this routine:

- compute *anorm* (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)
- call `?sptfrf` to compute the factorization of *A*.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix <i>A</i> has been factored: If <i>uplo</i> = 'U', the array <i>ap</i> stores the packed upper triangular factor <i>U</i> of the factorization $A = P*U*D*U^T*P^T$. If <i>uplo</i> = 'L', the array <i>ap</i> stores the packed lower triangular factor <i>L</i> of the factorization $A = P*L*D*L^T*P^T$.
<i>n</i>	INTEGER. The order of matrix <i>A</i>; $n \geq 0$.
<i>ap, work</i>	REAL for sspcon DOUBLE PRECISION for dspcon COMPLEX for cspcon DOUBLE COMPLEX for zspcon. Arrays: <i>ap</i>(*), <i>work</i>(*). The array <i>ap</i> contains the packed factored matrix <i>A</i>, as returned by <code>?sptfrf</code>. The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$. The array <i>work</i> is a workspace for the routine. The dimension of <i>work</i> must be at least $\max(1, 2*n)$.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The array <i>ipiv</i>, as returned by <code>?sptfrf</code>.
<i>anorm</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. The norm of the <i>original</i> matrix <i>A</i> (see <i>Description</i>).
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>rcond</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal of the condition number. The routine sets $rcond = 0$ if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime $rcond$ is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info INTEGER. If $info = 0$, the execution is successful.
If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `spcon` interface are as follows:

<i>a</i>	Stands for argument <i>ap</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(n * (n + 1) / 2)$.
<i>ipiv</i>	Holds the vector of length (n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed $rcond$ is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A * x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

?hpcon

Estimates the reciprocal of the condition number of a packed Hermitian matrix.

Syntax

FORTRAN 77:

```
call chpcon( uplo, n, ap, ipiv, anorm, rcond, work, info )
call zhpcon( uplo, n, ap, ipiv, anorm, rcond, work, info )
```

Fortran 95:

```
call hpcon( a, ipiv, anorm, rcond [,uplo] [,info] )
```

Description

This routine estimates the reciprocal of the condition number of a Hermitian matrix A :

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \text{ (since } A \text{ is Hermitian, } \kappa_\infty(A) = \kappa_1(A) \text{)}.$$

Before calling this routine:

- compute *anorm* (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)
- call [?hptrf](#) to compute the factorization of A .

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', the array <i>ap</i> stores the packed upper triangular factor U of the factorization $A = P*U*D*U^T*P^T$. If <i>uplo</i> = 'L', the array <i>ap</i> stores the packed lower triangular factor L of the factorization $A = P*L*D*L^T*P^T$.
<i>n</i>	INTEGER. The order of matrix A; $n \geq 0$.
<i>ap, work</i>	COMPLEX for <i>chpcon</i> DOUBLE COMPLEX for <i>zhpcon</i>. Arrays: <i>ap</i>(*), <i>work</i>(*). The array <i>ap</i> contains the packed factored matrix A, as returned by ?hptrf. The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$. The array <i>work</i> is a workspace for the routine. The dimension of <i>work</i> must be at least $\max(1, 2*n)$.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The array <i>ipiv</i>, as returned by ?hptrf.
<i>anorm</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. The norm of the <i>original</i> matrix A (see <i>Description</i>).

Output Parameters

<i>rcond</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets <i>rcond</i> = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <i>rcond</i> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hbcon` interface are as follows:

<i>a</i>	Stands for argument <i>ap</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(n * (n + 1) / 2)$.
<i>ipiv</i>	Holds the vector of length (<i>n</i>).

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations $A * x = b$; the number is usually 5 and never more than 11. Each solution requires approximately $8n^2$ floating-point operations.

?trcon

Estimates the reciprocal of the condition number of a triangular matrix.

Syntax

FORTRAN 77:

```
call strcon( norm, uplo, diag, n, a, lda, rcond, work, iwork, info )
call dtrcon( norm, uplo, diag, n, a, lda, rcond, work, iwork, info )
call ctrcon( norm, uplo, diag, n, a, lda, rcond, work, rwork, info )
call ztrcon( norm, uplo, diag, n, a, lda, rcond, work, rwork, info )
```

Fortran 95:

```
call trcon( a, rcond [,uplo] [,diag] [,norm] [,info] )
```

Description

This routine estimates the reciprocal of the condition number of a triangular matrix A in either the 1-norm or infinity-norm:

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$$

$$\kappa_\infty(A) = \|A\|_\infty \|A^{-1}\|_\infty = \kappa_1(A^T) = \kappa_1(A^H).$$

Input Parameters

<i>norm</i>	CHARACTER*1. Must be '1' or 'O' or 'I'. If <i>norm</i> = '1' or 'O', then the routine estimates the condition number of matrix A in 1-norm. If <i>norm</i> = 'I', then the routine estimates the condition number of matrix A in infinity-norm.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangle of A , other array elements are not referenced. If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangle of A , other array elements are not referenced.
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'.

If *diag* = 'N', then *A* is not a unit triangular matrix.

If *diag* = 'U', then *A* is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array *a*.

n INTEGER. The order of the matrix *A*; $n \geq 0$.

a, work REAL for *strcon*
DOUBLE PRECISION for *dtrcon*
COMPLEX for *ctrcon*
DOUBLE COMPLEX for *ztrcon*.

Arrays: *a(lda,*)*, *work(*)*.

The array *a* contains the matrix *A*. The second dimension of *a* must be at least $\max(1, n)$.

The array *work* is a workspace for the routine. The dimension of *work* must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

lda INTEGER. The first dimension of *a*; $lda \geq \max(1, n)$.

iwork INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.

rwork REAL for *ctrcon*
DOUBLE PRECISION for *ztrcon*.
Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

rcond REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
An estimate of the reciprocal of the condition number. The routine sets *rcond*=0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `trcon` interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>norm</i>	Must be '1', 'O', or 'I'. The default value is '1'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A^*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately n^2 floating-point operations for real flavors and $4n^2$ operations for complex flavors.

?tpcon

Estimates the reciprocal of the condition number of a packed triangular matrix.

Syntax

FORTRAN 77:

```
call stpcon( norm, uplo, diag, n, ap, rcond, work, iwork, info )
call dtpcon( norm, uplo, diag, n, ap, rcond, work, iwork, info )
call ctpcon( norm, uplo, diag, n, ap, rcond, work, rwork, info )
call ztpcon( norm, uplo, diag, n, ap, rcond, work, rwork, info )
```

Fortran 95:

```
call tpcon( a, rcond [,uplo] [,diag] [,norm] [,info] )
```

Description

This routine estimates the reciprocal of the condition number of a packed triangular matrix *A* in either the 1-norm or infinity-norm:

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$$

$$\kappa_\infty(A) = \|A\|_\infty \|A^{-1}\|_\infty = \kappa_1(A^T) = \kappa_1(A^H) .$$

Input Parameters

<i>norm</i>	CHARACTER*1. Must be '1' or 'O' or 'I'. If <i>norm</i> = '1' or 'O', then the routine estimates the condition number of matrix <i>A</i> in 1-norm. If <i>norm</i> = 'I', then the routine estimates the condition number of matrix <i>A</i> in infinity-norm.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether <i>A</i> is upper or lower triangular: If <i>uplo</i> = 'U', the array <i>ap</i> stores the upper triangle of <i>A</i> in packed form. If <i>uplo</i> = 'L', the array <i>ap</i> stores the lower triangle of <i>A</i> in packed form.
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. If <i>diag</i> = 'N', then <i>A</i> is not a unit triangular matrix. If <i>diag</i> = 'U', then <i>A</i> is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array <i>ap</i>.
<i>n</i>	INTEGER. The order of the matrix <i>A</i>; $n \geq 0$.
<i>ap, work</i>	REAL for stpcon DOUBLE PRECISION for dtpcon COMPLEX for ctpcon DOUBLE COMPLEX for ztpcon. Arrays: <i>ap</i>(*), <i>work</i>(*). The array <i>ap</i> contains the packed matrix <i>A</i>. The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$. The array <i>work</i> is a workspace for the routine. The dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.
<i>rwork</i>	REAL for ctpcon DOUBLE PRECISION for ztpcon. Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>rcond</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal of the condition number. The routine sets $rcond = 0$ if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime $rcond$ is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info

INTEGER. If $info = 0$, the execution is successful.
If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tpcon` interface are as follows:

<i>a</i>	Stands for argument <i>ap</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(n * (n + 1) / 2)$.
<i>norm</i>	Must be '1', 'O', or 'I'. The default value is '1'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The computed $rcond$ is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A * x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately n^2 floating-point operations for real flavors and $4n^2$ operations for complex flavors.

?tbcon

Estimates the reciprocal of the condition number of a triangular band matrix.

Syntax

FORTRAN 77:

```
call stbcon( norm, uplo, diag, n, kd, ab, ldab, rcond, work, iwork, info )
call dtbcon( norm, uplo, diag, n, kd, ab, ldab, rcond, work, iwork, info )
call ctbcon( norm, uplo, diag, n, kd, ab, ldab, rcond, work, rwork, info )
call ztbcon( norm, uplo, diag, n, kd, ab, ldab, rcond, work, rwork, info )
```

Fortran 95:

```
call tbcon( a, rcond [,uplo] [,diag] [,norm] [,info] )
```

Description

This routine estimates the reciprocal of the condition number of a triangular band matrix A in either the 1-norm or infinity-norm:

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$$

$$\kappa_\infty(A) = \|A\|_\infty \|A^{-1}\|_\infty = \kappa_1(A^T) = \kappa_1(A^H) .$$

Input Parameters

<i>norm</i>	CHARACTER*1. Must be '1' or 'O' or 'I'. If <i>norm</i> = '1' or 'O', then the routine estimates the condition number of matrix A in 1-norm. If <i>norm</i> = 'I', then the routine estimates the condition number of matrix A in infinity-norm.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If <i>uplo</i> = 'U', the array <i>ap</i> stores the upper triangle of A in packed form. If <i>uplo</i> = 'L', the array <i>ap</i> stores the lower triangle of A in packed form.
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'.

If *diag* = 'N', then *A* is not a unit triangular matrix.

If *diag* = 'U', then *A* is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array *ab*.

n INTEGER. The order of the matrix *A*; $n \geq 0$.

kd INTEGER. The number of superdiagonals or subdiagonals in the matrix *A*; $kd \geq 0$.

ab, work REAL for stbcon
DOUBLE PRECISION for dtbcon
COMPLEX for ctbcon
DOUBLE COMPLEX for ztbcon.
Arrays: *ab*(*ldab*,*), *work*(*).
The array *ab* contains the band matrix *A*. The second dimension of *ab* must be at least $\max(1, n)$. The array *work* is a workspace for the routine.
The dimension of *work* must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

ldab INTEGER. The first dimension of the array *ab*. ($ldab \geq kd + 1$).

iwork INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.

rwork REAL for ctbcon
DOUBLE PRECISION for ztbcon.
Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

rcond REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
An estimate of the reciprocal of the condition number. The routine sets *rcond* = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tbcon` interface are as follows:

<i>a</i>	Stands for argument <i>ab</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(kd+1, n)$.
<i>norm</i>	Must be '1', 'O', or 'I'. The default value is '1'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2*n(kd + 1)$ floating-point operations for real flavors and $8*n(kd + 1)$ operations for complex flavors.

Refining the Solution and Estimating Its Error

This section describes the LAPACK routines for refining the computed solution of a system of linear equations and estimating the solution error. You can call these routines after factorizing the matrix of the system of equations and computing the solution (see [Routines for Matrix Factorization](#) and [Routines for Solving Systems of Linear Equations](#)).

?gerfs

Refines the solution of a system of linear equations with a general matrix and estimates its error.

Syntax

FORTRAN 77:

```
call sgerfs( trans, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr,
berr, work, iwork, info )

call dgerfs( trans, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr,
berr, work, iwork, info )

call cgerfs( trans, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr,
berr, work, rwork, info )

call zgerfs( trans, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr,
berr, work, rwork, info )
```

Fortran 95:

```
call gerfs( a, af, ipiv, b, x [,trans] [,ferr] [,berr] [,info] )
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations $A * X = B$ or $A^T * X = B$ or $A^H * X = B$ with a general matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}|/|a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i|/|b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?getrf](#)
- call the solver routine [?getrs](#).

Input Parameters

<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', the system has the form $A * X = B$. If <i>trans</i> = 'T', the system has the form $A^T * X = B$. If <i>trans</i> = 'C', the system has the form $A^H * X = B$.
<i>n</i>	INTEGER. The order of the matrix <i>A</i>; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>a,af,b,x,work</i>	REAL for sgerfs DOUBLE PRECISION for dgerfs COMPLEX for cgerfs DOUBLE COMPLEX for zgerfs. Arrays: <i>a</i>(<i>lda</i>,*) contains the original matrix <i>A</i>, as supplied to ?getrf. <i>af</i>(<i>ldaf</i>,*) contains the factored matrix <i>A</i>, as returned by ?getrf. <i>b</i>(<i>ldb</i>,*) contains the right-hand side matrix <i>B</i>. <i>x</i>(<i>ldx</i>,*) contains the solution matrix <i>X</i>. <i>work</i>(*) is a workspace array. The second dimension of <i>a</i> and <i>af</i> must be at least $\max(1, n)$; the second dimension of <i>b</i> and <i>x</i> must be at least $\max(1, nrhs)$; the dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>lda</i>	INTEGER. The first dimension of <i>a</i>; $lda \geq \max(1, n)$.
<i>ldaf</i>	INTEGER. The first dimension of <i>af</i>; $ldaf \geq \max(1, n)$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i>; $ldb \geq \max(1, n)$.
<i>ldx</i>	INTEGER. The first dimension of <i>x</i>; $ldx \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?getrf.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.
<i>rwork</i>	REAL for cgerfs DOUBLE PRECISION for zgerfs.

Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>x</i>	The refined solution matrix <i>X</i> .
<i>ferr</i> , <i>berr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *gerfs* interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>af</i>	Holds the matrix <i>AF</i> of size (<i>n</i> , <i>n</i>).
<i>ipiv</i>	Holds the vector of length (<i>n</i>).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>nrhs</i>).
<i>x</i>	Holds the matrix <i>X</i> of size (<i>n</i> , <i>nrhs</i>).
<i>ferr</i>	Holds the vector of length (<i>nrhs</i>).
<i>berr</i>	Holds the vector of length (<i>nrhs</i>).
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward

error involves solving a number of systems of linear equations $A^*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

?gbrfs

Refines the solution of a system of linear equations with a general band matrix and estimates its error.

Syntax

FORTRAN 77:

```
call sgbrfs( trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, b, ldb, x,
            ldx, ferr, berr, work, iwork, info )

call dgbrfs( trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, b, ldb, x,
            ldx, ferr, berr, work, iwork, info )

call cgbrfs( trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, b, ldb, x,
            ldx, ferr, berr, work, rwork, info )

call zgbrfs( trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, b, ldb, x,
            ldx, ferr, berr, work, rwork, info )
```

Fortran 95:

```
call gbrfs( a, af, ipiv, b, x [,kl] [,trans] [,ferr] [,berr] [,info] )
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations $A^*X = B$ or $A^T * X = B$ or $A^H * X = B$ with a band matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}|/|a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i|/|b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?gbtrf](#)

- call the solver routine [?gbtrs](#).

Input Parameters

<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', the system has the form $A^*X = B$. If <i>trans</i> = 'T', the system has the form $A^T * X = B$. If <i>trans</i> = 'C', the system has the form $A^H * X = B$.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ; $n \geq 0$.
<i>kl</i>	INTEGER. The number of sub-diagonals within the band of <i>A</i> ; $kl \geq 0$.
<i>ku</i>	INTEGER. The number of super-diagonals within the band of <i>A</i> ; $ku \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ab,afb,b,x,work</i>	REAL for <i>sgbrfs</i> DOUBLE PRECISION for <i>dgbrfs</i> COMPLEX for <i>cgbrfs</i> DOUBLE COMPLEX for <i>zgbrfs</i>. Arrays: <i>ab</i>(<i>ldab</i>,*) contains the original band matrix <i>A</i>, as supplied to ?gbtrf, but stored in rows from 1 to $kl + ku + 1$. <i>afb</i>(<i>ldaafb</i>,*) contains the factored band matrix <i>A</i>, as returned by ?gbtrf. <i>b</i>(<i>ldb</i>,*) contains the right-hand side matrix <i>B</i>. <i>x</i>(<i>ldx</i>,*) contains the solution matrix <i>X</i>. <i>work</i>(*) is a workspace array. The second dimension of <i>ab</i> and <i>afb</i> must be at least $\max(1, n)$; the second dimension of <i>b</i> and <i>x</i> must be at least $\max(1, nrhs)$; the dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>ldab</i>	INTEGER. The first dimension of <i>ab</i> .
<i>ldaafb</i>	INTEGER. The first dimension of <i>afb</i> .
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; $ldb \geq \max(1, n)$.
<i>ldx</i>	INTEGER. The first dimension of <i>x</i> ; $ldx \geq \max(1, n)$.
<i>ipiv</i>	INTEGER.

	Array, DIMENSION at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?gbtrf .
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.
<i>rwork</i>	REAL for <i>cgbrfs</i> DOUBLE PRECISION for <i>zgbrfs</i> . Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>x</i>	The refined solution matrix <i>X</i> .
<i>ferr, berr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *gbrfs* interface are as follows:

<i>a</i>	Stands for argument <i>ab</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(kl+ku+1, n)$.
<i>af</i>	Stands for argument <i>afb</i> in Fortran 77 interface. Holds the array <i>AF</i> of size $(2*kl*ku+1, n)$.
<i>ipiv</i>	Holds the vector of length (n) .
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>kl</i>	If omitted, assumed $kl = ku$.
<i>ku</i>	Restored as $ku = lda-kl-1$.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n(k_l + k_u)$ floating-point operations (for real flavors) or $16n(k_l + k_u)$ operations (for complex flavors). In addition, each step of iterative refinement involves $2n(4k_l + 3k_u)$ operations (for real flavors) or $8n(4k_l + 3k_u)$ operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations $A^*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

?gtrfs

Refines the solution of a system of linear equations with a tridiagonal matrix and estimates its error.

Syntax

FORTRAN 77:

```
call sgtrfs( trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x,
            ldx, ferr, berr, work, iwork, info )

call dgtrfs( trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x,
            ldx, ferr, berr, work, iwork, info )

call cgtrfs( trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x,
            ldx, ferr, berr, work, rwork, info )

call zgtrfs( trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x,
            ldx, ferr, berr, work, rwork, info )
```

Fortran 95:

```
call gtrfs( dl, d, du, dlf, df, duf, du2, ipiv, b, x [,trans] [,ferr] [,berr]
            [,info] )
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations $A * X = B$ or $A^T * X = B$ or $A^H * X = B$ with a tridiagonal matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}|/|a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i|/|b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?gttrf](#)
- call the solver routine [?gttrs](#).

Input Parameters

<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', the system has the form $A * X = B$. If <i>trans</i> = 'T', the system has the form $A^T * X = B$. If <i>trans</i> = 'C', the system has the form $A^H * X = B$.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, that is, the number of columns of the matrix B ; $nrhs \geq 0$.
<i>dl,d,du,dlf,</i>	REAL for sgtrfs DOUBLE PRECISION for dgtrfs
<i>df,duf,du2,</i>	COMPLEX for cgtrfs DOUBLE COMPLEX for zgtrfs.
<i>b,x,work</i>	Arrays: <i>dl</i> , dimension $(n - 1)$, contains the subdiagonal elements of A . <i>d</i> , dimension (n) , contains the diagonal elements of A . <i>du</i> , dimension $(n - 1)$, contains the superdiagonal elements of A . <i>dlf</i> , dimension $(n - 1)$, contains the $(n - 1)$ multipliers that define the matrix L from the LU factorization of A as computed by ?gttrf .

	<i>df</i> , dimension (<i>n</i>), contains the <i>n</i> diagonal elements of the upper triangular matrix <i>U</i> from the <i>LU</i> factorization of <i>A</i> .
	<i>duf</i> , dimension (<i>n</i> - 1), contains the (<i>n</i> - 1) elements of the first superdiagonal of <i>U</i> .
	<i>du2</i> , dimension (<i>n</i> - 2), contains the (<i>n</i> - 2) elements of the second superdiagonal of <i>U</i> .
	<i>b(ldb, nrhs)</i> contains the right-hand side matrix <i>B</i> .
	<i>x(ldx, nrhs)</i> contains the solution matrix <i>X</i> , as computed by ?gttrs .
	<i>work</i> (*) is a workspace array; the dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; $ldb \geq \max(1, n)$.
<i>ldx</i>	INTEGER. The first dimension of <i>x</i> ; $ldx \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?gttrf .
<i>iwork</i>	INTEGER. Workspace array, DIMENSION (<i>n</i>). Used for real flavors only.
<i>rwork</i>	REAL for <i>cgtrfs</i> DOUBLE PRECISION for <i>zgtrfs</i> . Workspace array, DIMENSION (<i>n</i>). Used for complex flavors only.

Output Parameters

<i>x</i>	The refined solution matrix <i>X</i> .
<i>ferr, berr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *gtrfs* interface are as follows:

<i>dl</i>	Holds the vector of length (<i>n</i> -1).
<i>d</i>	Holds the vector of length (<i>n</i>).
<i>du</i>	Holds the vector of length (<i>n</i> -1).
<i>dlf</i>	Holds the vector of length (<i>n</i> -1).
<i>df</i>	Holds the vector of length (<i>n</i>).
<i>duf</i>	Holds the vector of length (<i>n</i> -1).
<i>du2</i>	Holds the vector of length (<i>n</i> -2).
<i>ipiv</i>	Holds the vector of length (<i>n</i>).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>nrhs</i>).
<i>x</i>	Holds the matrix <i>X</i> of size (<i>n</i> , <i>nrhs</i>).
<i>ferr</i>	Holds the vector of length (<i>nrhs</i>).
<i>berr</i>	Holds the vector of length (<i>nrhs</i>).
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.

?porfs

Refines the solution of a system of linear equations with a symmetric (Hermitian) positive-definite matrix and estimates its error.

Syntax

FORTRAN 77:

```
call sporfs( uplo, n, nrhs, a, lda, af, ldaf, b, ldb, x, ldx, ferr, berr,
work, iwork, info )

call dporfs( uplo, n, nrhs, a, lda, af, ldaf, b, ldb, x, ldx, ferr, berr,
work, iwork, info )

call cporfs( uplo, n, nrhs, a, lda, af, ldaf, b, ldb, x, ldx, ferr, berr,
work, rwork, info )

call zporfs( uplo, n, nrhs, a, lda, af, ldaf, b, ldb, x, ldx, ferr, berr,
work, rwork, info )
```

Fortran 95:

```
call porfs( a, af, b, x [,uplo] [,ferr] [,berr] [,info] )
```


Description

This routine performs an iterative refinement of the solution to a system of linear equations $A^*X = B$ with a symmetric (Hermitian) positive definite matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}|/|a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i|/|b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?potrf](#)
- call the solver routine [?potrs](#).

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
If *uplo* = 'U', the upper triangle of A is stored.
If *uplo* = 'L', the lower triangle of A is stored.

n INTEGER. The order of the matrix A ; $n \geq 0$.

nrhs INTEGER. The number of right-hand sides; $nrhs \geq 0$.

a,af,b,x,work REAL for *sporfs*
DOUBLE PRECISION for *dporfs*
COMPLEX for *cporfs*
DOUBLE COMPLEX for *zporfs*.

Arrays:
a(lda,)* contains the original matrix A , as supplied to [?potrf](#).
af(ldaf,)* contains the factored matrix A , as returned by [?potrf](#).
b(ldb,)* contains the right-hand side matrix B .
x(ldx,)* contains the solution matrix X .
work()* is a workspace array.

The second dimension of *a* and *af* must be at least $\max(1, n)$; the second dimension of *b* and *x* must be at least $\max(1, nrhs)$; the dimension of *work* must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldaf</i>	INTEGER. The first dimension of <i>af</i> ; $ldaf \geq \max(1, n)$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; $ldb \geq \max(1, n)$.
<i>ldx</i>	INTEGER. The first dimension of <i>x</i> ; $ldx \geq \max(1, n)$.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.
<i>rwork</i>	REAL for <i>cporfs</i> DOUBLE PRECISION for <i>zporfs</i> . Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>x</i>	The refined solution matrix <i>X</i> .
<i>ferr, berr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *porfs* interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (n, n) .
<i>af</i>	Holds the matrix <i>AF</i> of size (n, n) .
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

?pprfs

Refines the solution of a system of linear equations with a packed symmetric (Hermitian) positive-definite matrix and estimates its error.

Syntax

FORTRAN 77:

```
call spprfs( uplo, n, nrhs, ap, afp, b, ldb, x, ldx, ferr, berr, work, iwork,
info )

call dpprfs( uplo, n, nrhs, ap, afp, b, ldb, x, ldx, ferr, berr, work, iwork,
info )

call cpprfs( uplo, n, nrhs, ap, afp, b, ldb, x, ldx, ferr, berr, work, rwork,
info )

call zpprfs( uplo, n, nrhs, ap, afp, b, ldb, x, ldx, ferr, berr, work, rwork,
info )
```

Fortran 95:

```
call pprfs( a, af, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations $A * X = B$ with a packed symmetric (Hermitian) positive definite matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| / |a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| / |b_i| \leq \beta |b_i| \text{ such that } (A + \delta A) x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?pptrf](#)
- call the solver routine [?pptrs](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ap, afp, b, x, work</i>	REAL for <i>spprfs</i> DOUBLE PRECISION for <i>dpprfs</i> COMPLEX for <i>cpprfs</i> DOUBLE COMPLEX for <i>zpprfs</i> . Arrays: <i>ap</i> (*) contains the original packed matrix A , as supplied to ?pptrf . <i>afp</i> (*) contains the factored packed matrix A , as returned by ?pptrf . <i>b</i> (ldb, *) contains the right-hand side matrix B . <i>x</i> (ldx, *) contains the solution matrix X . <i>work</i> (*) is a workspace array.

The dimension of arrays *ap* and *afp* must be at least $\max(1, n(n+1)/2)$; the second dimension of *b* and *x* must be at least $\max(1, nrhs)$; the dimension of *work* must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

ldb INTEGER. The first dimension of *b*; $ldb \geq \max(1, n)$.

ldx INTEGER. The first dimension of *x*; $ldx \geq \max(1, n)$.

iwork INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.

rwork REAL for *cpprfs*
DOUBLE PRECISION for *zpprfs*.
Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

x The refined solution matrix *X*.

ferr, berr REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
Arrays, DIMENSION at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

info INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *pprfs* interface are as follows:

a Stands for argument *ap* in Fortran 77 interface. Holds the array *A* of size $(n*(n+1)/2)$.

af Stands for argument *afp* in Fortran 77 interface. Holds the array *AF* of size $(n*(n+1)/2)$.

b Holds the matrix *B* of size $(n, nrhs)$.

x Holds the matrix *X* of size $(n, nrhs)$.

ferr Holds the vector of length $(nrhs)$.

berr Holds the vector of length $(nrhs)$.

`uplo`

Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The bounds returned in `ferr` are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations $A^*x = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

?pbrfs

Refines the solution of a system of linear equations with a band symmetric (Hermitian) positive-definite matrix and estimates its error.

Syntax

Fortran 77:

```
call spbrfs( uplo, n, kd, nrhs, ab, ldab, afb, ldafb, b, ldb, x, ldx, ferr,
berr, work, iwork, info )

call dpbrfs( uplo, n, kd, nrhs, ab, ldab, afb, ldafb, b, ldb, x, ldx, ferr,
berr, work, iwork, info )

call cpbrfs( uplo, n, kd, nrhs, ab, ldab, afb, ldafb, b, ldb, x, ldx, ferr,
berr, work, rwork, info )

call zpbrfs( uplo, n, kd, nrhs, ab, ldab, afb, ldafb, b, ldb, x, ldx, ferr,
berr, work, rwork, info )
```

Fortran 95:

```
call pbrfs( a, af, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations $A^*X = B$ with a symmetric (Hermitian) positive definite band matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}|/|a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i|/|b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?pbtrf](#)
- call the solver routine [?pbtrs](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>kd</i>	INTEGER. The number of superdiagonals or subdiagonals in the matrix A ; $kd \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ab,afb,b,x,work</i>	REAL for spbrfs DOUBLE PRECISION for dpbrfs COMPLEX for cpbrfs DOUBLE COMPLEX for zpbrfs. Arrays: <i>ab(ldab,*)</i> contains the original band matrix A , as supplied to ?pbtrf . <i>afb(ldafb,*)</i> contains the factored band matrix A , as returned by ?pbtrf . <i>b(l db,*)</i> contains the right-hand side matrix B . <i>x(ldx,*)</i> contains the solution matrix X . <i>work(*)</i> is a workspace array.

The second dimension of *ab* and *afb* must be at least $\max(1, n)$; the second dimension of *b* and *x* must be at least $\max(1, nrhs)$; the dimension of *work* must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

<i>ldab</i>	INTEGER. The first dimension of <i>ab</i> ; $ldab \geq kd + 1$.
<i>ldafb</i>	INTEGER. The first dimension of <i>afb</i> ; $ldafb \geq kd + 1$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; $ldb \geq \max(1, n)$.
<i>ldx</i>	INTEGER. The first dimension of <i>x</i> ; $ldx \geq \max(1, n)$.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.
<i>rwork</i>	REAL for <i>cpbrfs</i> DOUBLE PRECISION for <i>zpbrfs</i> . Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>x</i>	The refined solution matrix <i>X</i> .
<i>ferr, berr</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *pbrfs* interface are as follows:

<i>a</i>	Stands for argument <i>ab</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(kd+1, n)$.
<i>af</i>	Stands for argument <i>afb</i> in Fortran 77 interface. Holds the array <i>AF</i> of size $(kd+1, n)$.
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.

<i>ferr</i>	Holds the vector of length (<i>nrhs</i>).
<i>berr</i>	Holds the vector of length (<i>nrhs</i>).
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $8n*kd$ floating-point operations (for real flavors) or $32n*kd$ operations (for complex flavors). In addition, each step of iterative refinement involves $12n*kd$ operations (for real flavors) or $48n*kd$ operations (for complex flavors); the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $4n*kd$ floating-point operations for real flavors or $16n*kd$ for complex flavors.

?ptrfs

Refines the solution of a system of linear equations with a symmetric (Hermitian) positive-definite tridiagonal matrix and estimates its error.

Syntax

FORTRAN 77:

```
call sptrfs( n, nrhs, d, e, df, ef, b, ldb, x, ldx, ferr, berr, work, info )
call dptrfs( n, nrhs, d, e, df, ef, b, ldb, x, ldx, ferr, berr, work, info )
call cptrfs( uplo, n, nrhs, d, e, df, ef, b, ldb, x, ldx, ferr, berr, work,
            rwork, info )
call zptrfs( uplo, n, nrhs, d, e, df, ef, b, ldb, x, ldx, ferr, berr, work,
            rwork, info )
```

Fortran 95:

```
call ptrfs( d, df, e, ef, b, x [,ferr] [,berr] [,info] )
call ptrfs( d, df, e, ef, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations $A * X = B$ with a symmetric (Hermitian) positive definite tridiagonal matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}|/|a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i|/|b_i| \leq \beta |b_i| \text{ such that } (A + \delta A) x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?pttrf](#)
- call the solver routine [?pttrs](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Used for complex flavors only. Must be 'U' or 'L'. Specifies whether the superdiagonal or the subdiagonal of the tridiagonal matrix A is stored and how A is factored: If <i>uplo</i> = 'U', the array <i>e</i> stores the superdiagonal of A , and A is factored as $U^H * D * U$. If <i>uplo</i> = 'L', the array <i>e</i> stores the subdiagonal of A , and A is factored as $L * D * L^H$.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>d, df, rwork</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors Arrays: <i>d</i> (<i>n</i>), <i>df</i> (<i>n</i>), <i>rwork</i> (<i>n</i>). The array <i>d</i> contains the <i>n</i> diagonal elements of the tridiagonal matrix A . The array <i>df</i> contains the <i>n</i> diagonal elements of the diagonal matrix D from the factorization of A as computed by ?pttrf . The array <i>rwork</i> is a workspace array used for complex flavors only.
<i>e, ef, b, x, work</i>	REAL for <i>spttrs</i> DOUBLE PRECISION for <i>dpttrs</i>

COMPLEX for `cptrfs`

DOUBLE COMPLEX for `zptrfs`.

Arrays: `e(n-1)`, `ef(n-1)`, `b(ldb,nrhs)`, `x(ldx,nrhs)`, `work(*)`.

The array `e` contains the $(n-1)$ off-diagonal elements of the tridiagonal matrix A (see `uplo`).

The array `ef` contains the $(n-1)$ off-diagonal elements of the unit bidiagonal factor U or L from the factorization computed by `?pttrf` (see `uplo`).

The array `b` contains the matrix B whose columns are the right-hand sides for the systems of equations.

The array `x` contains the solution matrix X as computed by `?pttrs`.

The array `work` is a workspace array. The dimension of `work` must be at least $2*n$ for real flavors, and at least n for complex flavors.

`ldb` INTEGER. The leading dimension of `b`; $ldb \geq \max(1, n)$.

`ldx` INTEGER. The leading dimension of `x`; $ldx \geq \max(1, n)$.

Output Parameters

`x` The refined solution matrix X .

`ferr, berr` REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

Arrays, DIMENSION at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

`info` INTEGER.

If `info` = 0, the execution is successful.

If `info` = $-i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ptrfs` interface are as follows:

`d` Holds the vector of length (n) .

`df` Holds the vector of length (n) .

`e` Holds the vector of length $(n-1)$.

<i>ef</i>	Holds the vector of length (<i>n</i> -1).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>nrhs</i>).
<i>x</i>	Holds the matrix <i>X</i> of size (<i>n</i> , <i>nrhs</i>).
<i>ferr</i>	Holds the vector of length (<i>nrhs</i>).
<i>berr</i>	Holds the vector of length (<i>nrhs</i>).
<i>uplo</i>	Used in complex flavors only. Must be 'U' or 'L'. The default value is 'U'.

?syrrfs

Refines the solution of a system of linear equations with a symmetric matrix and estimates its error.

Syntax

Fortran 77:

```
call ssyrrfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr,
berr, work, iwork, info )

call dsyrrfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr,
berr, work, iwork, info )

call csyrrfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr,
berr, work, rwork, info )

call zsyrrfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr,
berr, work, rwork, info )
```

Fortran 95:

```
call syrrfs( a, af, ipiv, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations $A \cdot X = B$ with a symmetric full-storage matrix *A*, with multiple right-hand sides. For each computed solution vector *x*, the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of *A* and *b* such that *x* is the exact solution of the perturbed system:

$$|\delta a_{ij}|/|a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i|/|b_i| \leq \beta |b_i| \text{ such that } (A + \delta A) x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine `?sytrf`
- call the solver routine `?sytrs`.

Input Parameters

<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'. If <code>uplo = 'U'</code> , the upper triangle of A is stored. If <code>uplo = 'L'</code> , the lower triangle of A is stored.
<code>n</code>	INTEGER. The order of the matrix A ; $n \geq 0$.
<code>nrhs</code>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<code>a, af, b, x, work</code>	REAL for <code>ssyrfs</code> DOUBLE PRECISION for <code>dsyrfs</code> COMPLEX for <code>csyrfs</code> DOUBLE COMPLEX for <code>zsyrfs</code> . Arrays: <code>a(lda, *)</code> contains the original matrix A , as supplied to <code>?sytrf</code> . <code>af(ldaf, *)</code> contains the factored matrix A , as returned by <code>?sytrf</code> . <code>b(ldb, *)</code> contains the right-hand side matrix B . <code>x(ldx, *)</code> contains the solution matrix X . <code>work(*)</code> is a workspace array. The second dimension of <code>a</code> and <code>af</code> must be at least $\max(1, n)$; the second dimension of <code>b</code> and <code>x</code> must be at least $\max(1, nrhs)$; the dimension of <code>work</code> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<code>lda</code>	INTEGER. The first dimension of <code>a</code> ; $lda \geq \max(1, n)$.
<code>ldaf</code>	INTEGER. The first dimension of <code>af</code> ; $ldaf \geq \max(1, n)$.
<code>ldb</code>	INTEGER. The first dimension of <code>b</code> ; $ldb \geq \max(1, n)$.
<code>ldx</code>	INTEGER. The first dimension of <code>x</code> ; $ldx \geq \max(1, n)$.
<code>ipiv</code>	INTEGER.

	Array, DIMENSION at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?sytrf .
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.
<i>rwork</i>	REAL for <i>csyrfs</i> DOUBLE PRECISION for <i>zsyrfs</i> . Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>x</i>	The refined solution matrix <i>X</i> .
<i>ferr, berr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *syrfs* interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (n, n) .
<i>af</i>	Holds the matrix <i>AF</i> of size (n, n) .
<i>ipiv</i>	Holds the vector of length (n) .
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations $A^*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

?herfs

Refines the solution of a system of linear equations with a complex Hermitian matrix and estimates its error.

Syntax

FORTRAN 77:

```
call cherfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr,
berr, work, rwork, info )

call zherfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr,
berr, work, rwork, info )
```

Fortran 95:

```
call herfs( a, af, ipiv, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations $A^*X = B$ with a complex Hermitian full-storage matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}|/|a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i|/|b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?hetrf](#)

- call the solver routine [?hetrs](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', the upper triangle of <i>A</i> is stored. If <i>uplo</i> = 'L', the lower triangle of <i>A</i> is stored.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>a,af,b,x,work</i>	COMPLEX for cherfs DOUBLE COMPLEX for zherfs. Arrays: <i>a(lda,*)</i> contains the original matrix <i>A</i> , as supplied to ?hetrf . <i>af(ldaf,*)</i> contains the factored matrix <i>A</i> , as returned by ?hetrf . <i>b(ldb,*)</i> contains the right-hand side matrix <i>B</i> . <i>x(ldx,*)</i> contains the solution matrix <i>X</i> . <i>work(*)</i> is a workspace array. The second dimension of <i>a</i> and <i>af</i> must be at least $\max(1, n)$; the second dimension of <i>b</i> and <i>x</i> must be at least $\max(1, nrhs)$; the dimension of <i>work</i> must be at least $\max(1, 2*n)$.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldaf</i>	INTEGER. The first dimension of <i>af</i> ; $ldaf \geq \max(1, n)$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; $ldb \geq \max(1, n)$.
<i>ldx</i>	INTEGER. The first dimension of <i>x</i> ; $ldx \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?hetrf .
<i>rwork</i>	REAL for cherfs DOUBLE PRECISION for zherfs. Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>x</i>	The refined solution matrix <i>X</i> .
<i>ferr, berr</i>	REAL for cherfs

DOUBLE PRECISION for `zherfs`.
 Arrays, DIMENSION at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.
info INTEGER. If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `herfs` interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (n, n) .
<i>af</i>	Holds the matrix <i>AF</i> of size (n, n) .
<i>ipiv</i>	Holds the vector of length (n) .
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $16n^2$ operations. In addition, each step of iterative refinement involves $24n^2$ operations; the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations $A * x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $8n^2$ floating-point operations.

The real counterpart of this routine is [?ssyrfs/?dsyrfs](#)

?sprfs

Refines the solution of a system of linear equations with a packed symmetric matrix and estimates the solution error.

Syntax

FORTRAN 77:

```
call ssprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work,
            iwork, info )

call dsprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work,
            iwork, info )

call csprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work,
            rwork, info )

call zsprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work,
            rwork, info )
```

Fortran 95:

```
call sprfs( a, af, ipiv, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations $A * X = B$ with a packed symmetric matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| / |a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| / |b_i| \leq \beta |b_i| \text{ such that } (A + \delta A) x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?sptf](#)
- call the solver routine [?sptrs](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', the upper triangle of <i>A</i> is stored. If <i>uplo</i> = 'L', the lower triangle of <i>A</i> is stored.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ap,afp,b,x,work</i>	REAL for <i>ssprfs</i> DOUBLE PRECISION for <i>dsprfs</i> COMPLEX for <i>csprfs</i> DOUBLE COMPLEX for <i>zsprfs</i> . Arrays: <i>ap</i> (*) contains the original packed matrix <i>A</i> , as supplied to ?sprtf . <i>afp</i> (*) contains the factored packed matrix <i>A</i> , as returned by ?sprtf . <i>b</i> (<i>ldb</i> ,*) contains the right-hand side matrix <i>B</i> . <i>x</i> (<i>ldx</i> ,*) contains the solution matrix <i>X</i> . <i>work</i> (*) is a workspace array. The dimension of arrays <i>ap</i> and <i>afp</i> must be at least $\max(1, n(n+1)/2)$; the second dimension of <i>b</i> and <i>x</i> must be at least $\max(1, nrhs)$; the dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; $ldb \geq \max(1, n)$.
<i>ldx</i>	INTEGER. The first dimension of <i>x</i> ; $ldx \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?sprtf .
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.
<i>rwork</i>	REAL for <i>csprfs</i> DOUBLE PRECISION for <i>zsprfs</i> . Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>x</i>	The refined solution matrix <i>X</i> .
<i>ferr, berr</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors.

Arrays, DIMENSION at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

info INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sprfs` interface are as follows:

<i>a</i>	Stands for argument <i>ap</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(n * (n+1) / 2)$.
<i>af</i>	Stands for argument <i>afp</i> in Fortran 77 interface. Holds the array <i>AF</i> of size $(n * (n+1) / 2)$.
<i>ipiv</i>	Holds the vector of length (n) .
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations $A * X = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

?hprfs

Refines the solution of a system of linear equations with a packed complex Hermitian matrix and estimates the solution error.

Syntax

FORTRAN 77:

```
call chprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work,
            rwork, info )

call zhprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work,
            rwork, info )
```

Fortran 95:

```
call hprfs( a, af, ipiv, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations $A \cdot X = B$ with a packed complex Hermitian matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}|/|a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i|/|b_i| \leq \beta |b_i| \text{ such that } (A + \delta A) x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?hptrf](#)
- call the solver routine [?hptrs](#).

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
 If *uplo* = 'U', the upper triangle of A is stored.
 If *uplo* = 'L', the lower triangle of A is stored.

<i>n</i>	INTEGER. The order of the matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ap,afp,b,x,work</i>	COMPLEX for <code>chprfs</code> DOUBLE COMPLEX for <code>zhprfs</code> . Arrays: <i>ap</i> (*) contains the original packed matrix <i>A</i> , as supplied to ?hptrf . <i>afp</i> (*) contains the factored packed matrix <i>A</i> , as returned by ?hptrf . <i>b</i> (<i>ldb</i> ,*) contains the right-hand side matrix <i>B</i> . <i>x</i> (<i>ldx</i> ,*) contains the solution matrix <i>X</i> . <i>work</i> (*) is a workspace array. The dimension of arrays <i>ap</i> and <i>afp</i> must be at least $\max(1, n(n+1)/2)$; the second dimension of <i>b</i> and <i>x</i> must be at least $\max(1, nrhs)$; the dimension of <i>work</i> must be at least $\max(1, 2*n)$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; $ldb \geq \max(1, n)$.
<i>ldx</i>	INTEGER. The first dimension of <i>x</i> ; $ldx \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?hptrf .
<i>rwork</i>	REAL for <code>chprfs</code> DOUBLE PRECISION for <code>zhprfs</code> . Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>x</i>	The refined solution matrix <i>X</i> .
<i>ferr,berr</i>	REAL for <code>chprfs</code> . DOUBLE PRECISION for <code>zhprfs</code> . Arrays, DIMENSION at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hprfs` interface are as follows:

<i>a</i>	Stands for argument <i>ap</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(n^* (n+1) / 2)$.
<i>af</i>	Stands for argument <i>afp</i> in Fortran 77 interface. Holds the array <i>AF</i> of size $(n^* (n+1) / 2)$.
<i>ipiv</i>	Holds the vector of length (n) .
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $16n^2$ operations. In addition, each step of iterative refinement involves $24n^2$ operations; the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations $A^*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $8n^2$ floating-point operations.

The real counterpart of this routine is [?ssprfs/?dsprfs](#).

?trrfs

Estimates the error in the solution of a system of linear equations with a triangular matrix.

Syntax

FORTRAN 77:

```
call strrfs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, x, ldx, ferr, berr,
work, iwork, info )

call dtrrfs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, x, ldx, ferr, berr,
work, iwork, info )

call ctrrfs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, x, ldx, ferr, berr,
work, rwork, info )

call ztrrfs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, x, ldx, ferr, berr,
work, rwork, info )
```

Fortran 95:

```
call trrfs( a, b, x [,uplo] [,trans] [,diag] [,ferr] [,berr] [,info] )
```

Description

This routine estimates the errors in the solution to a system of linear equations $A^*X = B$ or $A^T * X = B$ or $A^H * X = B$ with a triangular matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}|/|a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i|/|b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

The routine also estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine, call the solver routine [?trtrs](#).

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
Indicates whether A is upper or lower triangular:
If *uplo* = 'U', then A is upper triangular.

If *uplo* = 'L', then *A* is lower triangular.

trans CHARACTER*1. Must be 'N' or 'T' or 'C'.
Indicates the form of the equations:
If *trans* = 'N', the system has the form $A * X = B$.
If *trans* = 'T', the system has the form $A^T * X = B$.
If *trans* = 'C', the system has the form $A^H * X = B$.

diag CHARACTER*1. Must be 'N' or 'U'.
If *diag* = 'N', then *A* is not a unit triangular matrix.
If *diag* = 'U', then *A* is unit triangular: diagonal elements of *A* are assumed to be 1 and not referenced in the array *a*.

n INTEGER. The order of the matrix *A*; $n \geq 0$.

nrhs INTEGER. The number of right-hand sides; $nrhs \geq 0$.

a, *b*, *x*, *work* REAL for *strrfs*
DOUBLE PRECISION for *dtrrfs*
COMPLEX for *ctrfrs*
DOUBLE COMPLEX for *ztrrfs*.
Arrays:
a(*lda*,*) contains the upper or lower triangular matrix *A*, as specified by *uplo*.
b(*ldb*,*) contains the right-hand side matrix *B*.
x(*ldx*,*) contains the solution matrix *X*.
work(*) is a workspace array.
The second dimension of *a* must be at least $\max(1, n)$; the second dimension of *b* and *x* must be at least $\max(1, nrhs)$; the dimension of *work* must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

lda INTEGER. The first dimension of *a*; $lda \geq \max(1, n)$.

ldb INTEGER. The first dimension of *b*; $ldb \geq \max(1, n)$.

ldx INTEGER. The first dimension of *x*; $ldx \geq \max(1, n)$.

iwork INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.

rwork REAL for *ctrfrs*
DOUBLE PRECISION for *ztrrfs*.
Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>ferr, berr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `trrfs` interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (n, n) .
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

A call to this routine involves, for each right-hand side, solving a number of systems of linear equations $A^*x = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately n^2 floating-point operations for real flavors or $4n^2$ for complex flavors.

?tprfs

Estimates the error in the solution of a system of linear equations with a packed triangular matrix.

Syntax

FORTRAN 77:

```
call stprfs( uplo, trans, diag, n, nrhs, ap, b, ldb, x, ldx, ferr, berr, work,
            iwork, info )
call dtprfs( uplo, trans, diag, n, nrhs, ap, b, ldb, x, ldx, ferr, berr, work,
            iwork, info )
call ctprfs( uplo, trans, diag, n, nrhs, ap, b, ldb, x, ldx, ferr, berr, work,
            rwork, info )
call ztprfs( uplo, trans, diag, n, nrhs, ap, b, ldb, x, ldx, ferr, berr, work,
            rwork, info )
```

Fortran 95:

```
call tprfs( a, b, x [,uplo] [,trans] [,diag] [,ferr] [,berr] [,info] )
```

Description

This routine estimates the errors in the solution to a system of linear equations $A^*X = B$ or $A^T * X = B$ or $A^H * X = B$ with a packed triangular matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}|/|a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i|/|b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

The routine also estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine, call the solver routine [?tpttrs](#).

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
 Indicates whether A is upper or lower triangular:
 If *uplo* = 'U', then A is upper triangular.

	If <i>uplo</i> = 'L', then <i>A</i> is lower triangular.
<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', the system has the form $A^*X = B$. If <i>trans</i> = 'T', the system has the form $A^T X = B$. If <i>trans</i> = 'C', the system has the form $A^H X = B$.
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. If <i>diag</i> = 'N', <i>A</i> is not a unit triangular matrix. If <i>diag</i> = 'U', <i>A</i> is unit triangular: diagonal elements of <i>A</i> are assumed to be 1 and not referenced in the array <i>ap</i>.
<i>n</i>	INTEGER. The order of the matrix <i>A</i>; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ap, b, x, work</i>	REAL for <i>stprfs</i> DOUBLE PRECISION for <i>dtprfs</i> COMPLEX for <i>ctprfs</i> DOUBLE COMPLEX for <i>ztpfrfs</i>. Arrays: <i>ap</i>(*) contains the upper or lower triangular matrix <i>A</i>, as specified by <i>uplo</i>. <i>b</i>(<i>ldb</i>,*) contains the right-hand side matrix <i>B</i>. <i>x</i>(<i>ldx</i>,*) contains the solution matrix <i>X</i>. <i>work</i>(*) is a workspace array. The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$; the second dimension of <i>b</i> and <i>x</i> must be at least $\max(1, nrhs)$; the dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i>; $ldb \geq \max(1, n)$.
<i>ldx</i>	INTEGER. The first dimension of <i>x</i>; $ldx \geq \max(1, n)$.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.
<i>rwork</i>	REAL for <i>ctprfs</i> DOUBLE PRECISION for <i>ztpfrfs</i>. Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>ferr, berr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tprfs` interface are as follows:

<i>a</i>	Stands for argument <i>ap</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(n * (n+1) / 2)$.
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

A call to this routine involves, for each right-hand side, solving a number of systems of linear equations $A * x = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately n^2 floating-point operations for real flavors or $4n^2$ for complex flavors.

?tbrfs

Estimates the error in the solution of a system of linear equations with a triangular band matrix.

Syntax

FORTRAN 77:

```
call stbrfs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, x, ldx, ferr,
berr, work, iwork, info )

call dtbrfs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, x, ldx, ferr,
berr, work, iwork, info )

call ctbrfs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, x, ldx, ferr,
berr, work, rwork, info )

call ztbrfs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, x, ldx, ferr,
berr, work, rwork, info )
```

Fortran 95:

```
call tbrfs( a, b, x [,uplo] [,trans] [,diag] [,ferr] [,berr] [,info] )
```

Description

This routine estimates the errors in the solution to a system of linear equations $A * X = B$ or $A^T * X = B$ or $A^H * X = B$ with a triangular band matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| / |a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| / |b_i| \leq \beta |b_i| \text{ such that } (A + \delta A) x = (b + \delta b).$$

The routine also estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine, call the solver routine [?tbtrs](#).

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
Indicates whether A is upper or lower triangular:
If *uplo* = 'U', then A is upper triangular.

	If <i>uplo</i> = 'L', then <i>A</i> is lower triangular.
<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', the system has the form $A^*X = B$. If <i>trans</i> = 'T', the system has the form $A^T * X = B$. If <i>trans</i> = 'C', the system has the form $A^H * X = B$.
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. If <i>diag</i> = 'N', <i>A</i> is not a unit triangular matrix. If <i>diag</i> = 'U', <i>A</i> is unit triangular: diagonal elements of <i>A</i> are assumed to be 1 and not referenced in the array <i>ab</i> .
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ; $n \geq 0$.
<i>kd</i>	INTEGER. The number of super-diagonals or sub-diagonals in the matrix <i>A</i> ; $kd \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ab, b, x, work</i>	REAL for stbrfs DOUBLE PRECISION for dtbrfs COMPLEX for ctbrfs DOUBLE COMPLEX for ztbrfs. Arrays: <i>ab(ldab,*)</i> contains the upper or lower triangular matrix <i>A</i> , as specified by <i>uplo</i> , in band storage format. <i>b(l db,*)</i> contains the right-hand side matrix <i>B</i> . <i>x(ldx,*)</i> contains the solution matrix <i>X</i> . <i>work(*)</i> is a workspace array. The second dimension of <i>a</i> must be at least $\max(1, n)$; the second dimension of <i>b</i> and <i>x</i> must be at least $\max(1, nrhs)$. The dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>ldab</i>	INTEGER. The first dimension of the array <i>ab</i> ; $ldab \geq kd + 1$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; $ldb \geq \max(1, n)$.
<i>ldx</i>	INTEGER. The first dimension of <i>x</i> ; $ldx \geq \max(1, n)$.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.
<i>rwork</i>	REAL for ctbrfs DOUBLE PRECISION for ztbrfs.

Workspace array, `DIMENSION` at least $\max(1, n)$.

Output Parameters

<i>ferr, berr</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. Arrays, <code>DIMENSION</code> at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tbrfs` interface are as follows:

<i>a</i>	Stands for argument <i>ab</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(kd+1, n)$.
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

A call to this routine involves, for each right-hand side, solving a number of systems of linear equations $A*x = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately $2n*kd$ floating-point operations for real flavors or $8n*kd$ operations for complex flavors.

Routines for Matrix Inversion

It is seldom necessary to compute an explicit inverse of a matrix. In particular, do not attempt to solve a system of equations $Ax = b$ by first computing A^{-1} and then forming the matrix-vector product $x = A^{-1}b$. Call a solver routine instead (see [Routines for Solving Systems of Linear Equations](#)); this is more efficient and more accurate.

However, matrix inversion routines are provided for the rare occasions when an explicit inverse matrix is needed.

?getri

Computes the inverse of an LU-factored general matrix.

Syntax

FORTRAN 77:

```
call sgetri( n, a, lda, ipiv, work, lwork, info )
call dgetri( n, a, lda, ipiv, work, lwork, info )
call cgetri( n, a, lda, ipiv, work, lwork, info )
call zgetri( n, a, lda, ipiv, work, lwork, info )
```

Fortran 95:

```
call getri( a, ipiv [,info] )
```

Description

This routine computes the inverse $\text{inv}(A)$ of a general matrix A . Before calling this routine, call [?getrf](#) to factorize A .

Input Parameters

n	INTEGER. The order of the matrix A ; $n \geq 0$.
$a, work$	REAL for sgetri DOUBLE PRECISION for dgetri COMPLEX for cgetri DOUBLE COMPLEX for zgetri.

Arrays: $a(lda, *)$, $work(*)$.
 $a(lda, *)$ contains the factorization of the matrix A , as returned by [?getrf](#): $A = P * L * U$.
The second dimension of a must be at least $\max(1, n)$.
 $work(*)$ is a workspace array of dimension at least $\max(1, lwork)$.

lda INTEGER. The first dimension of a ; $lda \geq \max(1, n)$.

ipiv INTEGER.
Array, DIMENSION at least $\max(1, n)$.
The *ipiv* array, as returned by [?getrf](#).

lwork INTEGER. The size of the *work* array; $lwork \geq n$.
If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by `xerbla`.

See *Application Notes* below for the suggested value of *lwork*.

Output Parameters

a Overwritten by the n -by- n matrix $\text{inv}(A)$.

work(1) If $info = 0$, on exit *work(1)* contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER. If $info = 0$, the execution is successful.
If $info = -i$, the i -th parameter had an illegal value.
If $info = i$, the i -th diagonal element of the factor U is zero, U is singular, and the inversion could not be completed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `getri` interface are as follows:

a Holds the matrix A of size (n, n) .
ipiv Holds the vector of length (n) .

Application Notes

For better performance, try using $lwork = n * blocksize$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed inverse X satisfies the following error bound:

$$\|XA - I\| \leq c(n)\epsilon\|X\|P\|L\|\|U\|,$$

where $c(n)$ is a modest linear function of n ; ϵ is the machine precision; I denotes the identity matrix; P , L , and U are the factors of the matrix factorization $A = P * L * U$.

The total number of floating-point operations is approximately $(4/3)n^3$ for real flavors and $(16/3)n^3$ for complex flavors.

?potri

Computes the inverse of a symmetric (Hermitian) positive-definite matrix.

Syntax

FORTRAN 77:

```
call spotri( uplo, n, a, lda, info )
call dpotri( uplo, n, a, lda, info )
call cpotri( uplo, n, a, lda, info )
call zpotri( uplo, n, a, lda, info )
```

Fortran 95:

```
call potri( a [,uplo] [,info] )
```

Description

This routine computes the inverse $\text{inv}(A)$ of a symmetric positive definite or, for complex flavors, Hermitian positive-definite matrix A . Before calling this routine, call [?potrf](#) to factorize A .

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If <i>uplo</i> = 'U', then A is upper triangular. If <i>uplo</i> = 'L', then A is lower triangular.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>a</i>	REAL for spotri DOUBLE PRECISION for dpotri COMPLEX for cpotri DOUBLE COMPLEX for zpotri. Array $a(lda, *)$. Contains the factorization of the matrix A , as returned by ?potrf . The second dimension of a must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The first dimension of a ; $lda \geq \max(1, n)$.

Output Parameters

a Overwritten by the n -by- n matrix `inv(A)`.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.
 If *info* = *i*, the *i*-th diagonal element of the Cholesky factor (and therefore the factor itself) is zero, and the inversion could not be completed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `potri` interface are as follows:

a Holds the matrix *A* of size (n, n) .

uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed inverse X satisfies the following error bounds:

$$\|XA - I\|_2 \leq c(n)\varepsilon\kappa_2(A), \quad \|AX - I\|_2 \leq c(n)\varepsilon\kappa_2(A),$$

where $c(n)$ is a modest linear function of n , and ε is the machine precision; I denotes the identity matrix.

The 2-norm $\|A\|_2$ of a matrix *A* is defined by $\|A\|_2 = \max_{x \cdot x=1} (Ax \cdot Ax)^{1/2}$, and the condition number $\kappa_2(A)$ is defined by $\kappa_2(A) = \|A\|_2 \|A^{-1}\|_2$.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

?pptri

Computes the inverse of a packed symmetric (Hermitian) positive-definite matrix

Syntax

FORTRAN 77:

```
call spptri( uplo, n, ap, info )
call dpptri( uplo, n, ap, info )
call cpptri( uplo, n, ap, info )
call zpptri( uplo, n, ap, info )
```

Fortran 95:

```
call pptri( a [,uplo] [,info] )
```

Description

This routine computes the inverse $\text{inv}(A)$ of a symmetric positive definite or, for complex flavors, Hermitian positive-definite matrix A in *packed* form. Before calling this routine, call [?pptrf](#) to factorize A .

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular factor is stored in <i>ap</i> : If <i>uplo</i> = 'U', then the upper triangular factor is stored. If <i>uplo</i> = 'L', then the lower triangular factor is stored.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>ap</i>	REAL for spptri DOUBLE PRECISION for dpptri COMPLEX for cpptri DOUBLE COMPLEX for zpptri. Array, DIMENSION at least $\max(1, n(n+1)/2)$. Contains the factorization of the packed matrix A , as returned by ?pptrf . The dimension <i>ap</i> must be at least $\max(1, n(n+1)/2)$.

Output Parameters

ap Overwritten by the packed n -by- n matrix $\text{inv}(A)$.

info INTEGER.
 If $\text{info} = 0$, the execution is successful.
 If $\text{info} = -i$, the i -th parameter had an illegal value.
 If $\text{info} = i$, the i -th diagonal element of the Cholesky factor (and therefore the factor itself) is zero, and the inversion could not be completed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `pptri` interface are as follows:

a Stands for argument *ap* in Fortran 77 interface. Holds the array *A* of size $(n*(n+1)/2)$.

uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed inverse X satisfies the following error bounds:

$$\|XA - I\|_2 \leq c(n)\varepsilon\kappa_2(A), \quad \|AX - I\|_2 \leq c(n)\varepsilon\kappa_2(A),$$

where $c(n)$ is a modest linear function of n , and ε is the machine precision; I denotes the identity matrix.

The 2-norm $\|A\|_2$ of a matrix A is defined by $\|A\|_2 = \max_{x \cdot x=1} (Ax \cdot Ax)^{1/2}$, and the condition number $\kappa_2(A)$ is defined by $\kappa_2(A) = \|A\|_2 \|A^{-1}\|_2$.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

?sytri

Computes the inverse of a symmetric matrix.

Syntax

FORTRAN 77:

```
call ssytri( uplo, n, a, lda, ipiv, work, info )
call dsytri( uplo, n, a, lda, ipiv, work, info )
call csytri( uplo, n, a, lda, ipiv, work, info )
call zsytri( uplo, n, a, lda, ipiv, work, info )
```

Fortran 95:

```
call sytri( a, ipiv [,uplo] [,info] )
```

Description

This routine computes the inverse $\text{inv}(A)$ of a symmetric matrix A . Before calling this routine, call [?sytrf](#) to factorize A .

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', the array <i>a</i> stores the Bunch-Kaufman factorization $A = P*U*D*U^T*P^T$. If <i>uplo</i> = 'L', the array <i>a</i> stores the Bunch-Kaufman factorization $A = P*L*D*L^T*P^T$.
<i>n</i>	INTEGER. The order of the matrix A; $n \geq 0$.
<i>a, work</i>	REAL for ssytri DOUBLE PRECISION for dsytri COMPLEX for csytri DOUBLE COMPLEX for zsytri. Arrays: <i>a</i>(<i>lda</i>,*) contains the factorization of the matrix A, as returned by ?sytrf. The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i>(*) is a workspace array.

The dimension of *work* must be at least $\max(1, 2*n)$.

lda INTEGER. The first dimension of *a*; $lda \geq \max(1, n)$.

ipiv INTEGER.
Array, DIMENSION at least $\max(1, n)$.
The *ipiv* array, as returned by [?sytrf](#).

Output Parameters

a Overwritten by the n -by- n matrix $\text{inv}(A)$.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.
If *info* = *i*, the *i*-th diagonal element of *D* is zero, *D* is singular, and the inversion could not be completed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sytri` interface are as follows:

a Holds the matrix *A* of size (n, n) .

ipiv Holds the vector of length (n) .

uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed inverse *x* satisfies the following error bounds:

$$|D*U^T*P^T*X*P*U - I| \leq c(n)\epsilon(|D||U^T|P^T|X|P|U| + |D||D^{-1}|)$$

for *uplo* = 'U', and

$$|D*L^T*P^T*X*P*L - I| \leq c(n)\epsilon(|D||L^T|P^T|X|P|L| + |D||D^{-1}|)$$

for *uplo* = 'L'. Here $c(n)$ is a modest linear function of n , and ϵ is the machine precision; *I* denotes the identity matrix.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

?hetri

Computes the inverse of a complex Hermitian matrix.

Syntax

FORTRAN 77:

```
call chetri( uplo, n, a, lda, ipiv, work, info )
call zhetri( uplo, n, a, lda, ipiv, work, info )
```

Fortran 95:

```
call hetri( a, ipiv [,uplo] [,info] )
```

Description

This routine computes the inverse $\text{inv}(A)$ of a complex Hermitian matrix A . Before calling this routine, call [?hetrf](#) to factorize A .

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', the array <i>a</i> stores the Bunch-Kaufman factorization $A = P*U*D*U^H*P^T$. If <i>uplo</i> = 'L', the array <i>a</i> stores the Bunch-Kaufman factorization $A = P*L*D*L^H*P^T$.
<i>n</i>	INTEGER. The order of the matrix A; $n \geq 0$.
<i>a, work</i>	COMPLEX for <code>chetri</code> DOUBLE COMPLEX for <code>zhetri</code>. Arrays: <i>a</i>(<i>lda</i>,*) contains the factorization of the matrix A, as returned by ?hetrf. The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i>(*) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The first dimension of <i>a</i>; $lda \geq \max(1, n)$.
<i>ipiv</i>	INTEGER.

Array, DIMENSION at least $\max(1, n)$. The *ipiv* array, as returned by [?hetrf](#).

Output Parameters

a Overwritten by the n -by- n matrix $\text{inv}(A)$.
info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.
 If *info* = *i*, the *i*-th diagonal element of *D* is zero, *D* is singular, and the inversion could not be completed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hetri` interface are as follows:

a Holds the matrix *A* of size (n, n) .
ipiv Holds the vector of length (n) .
uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed inverse X satisfies the following error bounds:

$$|D^*U^H P^T X P U - I| \leq c(n)\epsilon(|D||U^H|P^T|X|P|U| + |D||D^{-1}|)$$

for *uplo* = 'U', and

$$|D^*L^H P^T X P L - I| \leq c(n)\epsilon(|D||L^H|P^T|X|P|L| + |D||D^{-1}|)$$

for *uplo* = 'L'. Here $c(n)$ is a modest linear function of n , and ϵ is the machine precision; I denotes the identity matrix.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

The real counterpart of this routine is [?sytri](#).

?sptri

Computes the inverse of a symmetric matrix using packed storage.

Syntax

FORTRAN 77:

```
call ssptri( uplo, n, ap, ipiv, work, info )
call dsptri( uplo, n, ap, ipiv, work, info )
call csptri( uplo, n, ap, ipiv, work, info )
call zsptri( uplo, n, ap, ipiv, work, info )
```

Fortran 95:

```
call sptri( a, ipiv [,uplo] [,info] )
```

Description

This routine computes the inverse $\text{inv}(A)$ of a packed symmetric matrix A . Before calling this routine, call [?sptrf](#) to factorize A .

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', the array <i>ap</i> stores the Bunch-Kaufman factorization $A = P*U*D*U^T*P^T$. If <i>uplo</i> = 'L', the array <i>ap</i> stores the Bunch-Kaufman factorization $A = P*L*D*L^T*P^T$.
<i>n</i>	INTEGER. The order of the matrix A; $n \geq 0$.
<i>ap, work</i>	REAL for ssptri DOUBLE PRECISION for dsptri COMPLEX for csptri DOUBLE COMPLEX for zsptri. Arrays: <i>ap</i>(*) contains the factorization of the matrix A, as returned by ?sptrf. The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$. <i>work</i>(*) is a workspace array.

The dimension of *work* must be at least $\max(1, n)$.

ipiv INTEGER.
Array, DIMENSION at least $\max(1, n)$. The *ipiv* array, as returned by [?spturf](#).

Output Parameters

ap Overwritten by the n -by- n matrix $\text{inv}(A)$ in packed form.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.
If *info* = *i*, the *i*-th diagonal element of *D* is zero, *D* is singular, and the inversion could not be completed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *spturi* interface are as follows:

a Stands for argument *ap* in Fortran 77 interface. Holds the array *A* of size $(n * (n+1) / 2)$.

ipiv Holds the vector of length (*n*).

uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed inverse *x* satisfies the following error bounds:

$$|D * U^T * P^T * X * P * U - I| \leq c(n) \epsilon (|D| |U^T| P^T |X| P |U| + |D| |D^{-1}|)$$

for *uplo* = 'U', and

$$|D * L^T * P^T * X * P * L - I| \leq c(n) \epsilon (|D| |L^T| P^T |X| P |L| + |D| |D^{-1}|)$$

for *uplo* = 'L'. Here $c(n)$ is a modest linear function of n , and ϵ is the machine precision; *I* denotes the identity matrix.

The total number of floating-point operations is approximately $(2/3) n^3$ for real flavors and $(8/3) n^3$ for complex flavors.

?hptri

Computes the inverse of a complex Hermitian matrix using packed storage.

Syntax

FORTRAN 77:

```
call chptri( uplo, n, ap, ipiv, work, info )
call zhptri( uplo, n, ap, ipiv, work, info )
```

Fortran 95:

```
call hptri( a, ipiv [,uplo] [,info] )
```

Description

This routine computes the inverse $\text{inv}(A)$ of a complex Hermitian matrix A using packed storage. Before calling this routine, call [?hptrf](#) to factorize A .

Input Parameters

<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <code>uplo</code> = 'U', the array <code>ap</code> stores the packed Bunch-Kaufman factorization $A = P*U*D*U^H*P^T$. If <code>uplo</code> = 'L', the array <code>ap</code> stores the packed Bunch-Kaufman factorization $A = P*L*D*L^H*P^T$.
<code>n</code>	INTEGER. The order of the matrix A; $n \geq 0$.
<code>ap, work</code>	COMPLEX for <code>chptri</code> DOUBLE COMPLEX for <code>zhptri</code>. Arrays: <code>ap(*)</code> contains the factorization of the matrix A, as returned by ?hptrf. The dimension of <code>ap</code> must be at least $\max(1, n(n+1)/2)$. <code>work(*)</code> is a workspace array. The dimension of <code>work</code> must be at least $\max(1, n)$.
<code>ipiv</code>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The <code>ipiv</code> array, as returned by ?hptrf.

Output Parameters

<i>ap</i>	Overwritten by the n -by- n matrix <code>inv(A)</code> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , the <i>i</i> -th diagonal element of <i>D</i> is zero, <i>D</i> is singular, and the inversion could not be completed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hptri` interface are as follows:

<i>a</i>	Stands for argument <i>ap</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(n * (n+1) / 2)$.
<i>ipiv</i>	Holds the vector of length (<i>n</i>).
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed inverse *X* satisfies the following error bounds:

$$|D^*U^H P^T X^* P^* U - I| \leq c(n) \varepsilon (|D| |U^H| P^T |X| P |U| + |D| |D^{-1}|)$$

for *uplo* = 'U', and

$$|D^* L^H P^T X^* P L - I| \leq c(n) \varepsilon (|D| |L^H| P^T |X| P |L| + |D| |D^{-1}|)$$

for *uplo* = 'L'. Here $c(n)$ is a modest linear function of n , and ε is the machine precision; *I* denotes the identity matrix.

The total number of floating-point operations is approximately $(2/3) n^3$ for real flavors and $(8/3) n^3$ for complex flavors.

The real counterpart of this routine is [?sptri](#).

ztrtri

Computes the inverse of a triangular matrix.

Syntax

FORTRAN 77:

```
call strtri( uplo, diag, n, a, lda, info )
call dtrtri( uplo, diag, n, a, lda, info )
call ctrtri( uplo, diag, n, a, lda, info )
call ztrtri( uplo, diag, n, a, lda, info )
```

Fortran 95:

```
call trtri( a [,uplo] [,diag] [,info] )
```

Description

This routine computes the inverse $\text{inv}(A)$ of a triangular matrix A .

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If <i>uplo</i> = 'U', then A is upper triangular. If <i>uplo</i> = 'L', then A is lower triangular.
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. If <i>diag</i> = 'N', then A is not a unit triangular matrix. If <i>diag</i> = 'U', A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array a.
<i>n</i>	INTEGER. The order of the matrix A; $n \geq 0$.
<i>a</i>	REAL for strtri DOUBLE PRECISION for dtrtri COMPLEX for ctrtri DOUBLE COMPLEX for ztrtri. Array: DIMENSION (, *). Contains the matrix A. The second dimension of a must be at least $\max(1, n)$.

lda INTEGER. The first dimension of *a*; $lda \geq \max(1, n)$.

Output Parameters

a Overwritten by the n -by- n matrix $\text{inv}(A)$.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.
 If *info* = *i*, the *i*-th diagonal element of *A* is zero, *A* is singular, and the inversion could not be completed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `trtri` interface are as follows:

a Holds the matrix *A* of size (n, n) .

uplo Must be 'U' or 'L'. The default value is 'U'.

diag Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The computed inverse X satisfies the following error bounds:

$$\|XA - I\| \leq c(n)\varepsilon \|X\| \|A\|$$

$$\|XA - I\| \leq c(n)\varepsilon \|A^{-1}\| \|A\| \|X\|,$$

where $c(n)$ is a modest linear function of n ; ε is the machine precision; I denotes the identity matrix.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors and $(4/3)n^3$ for complex flavors.

?tptri

Computes the inverse of a triangular matrix using packed storage.

Syntax

FORTRAN 77:

```
call stptri( uplo, diag, n, ap, info )
call dtptri( uplo, diag, n, ap, info )
call ctptri( uplo, diag, n, ap, info )
call ztptri( uplo, diag, n, ap, info )
```

Fortran 95:

```
call tptri( a [,uplo] [,diag] [,info] )
```

Description

This routine computes the inverse $\text{inv}(A)$ of a packed triangular matrix A .

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If <i>uplo</i> = 'U', then A is upper triangular. If <i>uplo</i> = 'L', then A is lower triangular.
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. If <i>diag</i> = 'N', then A is not a unit triangular matrix. If <i>diag</i> = 'U', A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array <i>ap</i>.
<i>n</i>	INTEGER. The order of the matrix A; $n \geq 0$.
<i>ap</i>	REAL for stptri DOUBLE PRECISION for dtptri COMPLEX for ctptri DOUBLE COMPLEX for ztptri. Array, DIMENSION at least $\max(1, n(n+1)/2)$. Contains the packed triangular matrix A.

Output Parameters

ap Overwritten by the packed n -by- n matrix $\text{inv}(A)$.

info INTEGER.
 If $\text{info} = 0$, the execution is successful.
 If $\text{info} = -i$, the i -th parameter had an illegal value.
 If $\text{info} = i$, the i -th diagonal element of A is zero, A is singular, and the inversion could not be completed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tptri` interface are as follows:

a Stands for argument *ap* in Fortran 77 interface. Holds the array A of size $(n*(n+1)/2)$.

uplo Must be 'U' or 'L'. The default value is 'U'.

diag Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The computed inverse X satisfies the following error bounds:

$$|XA - I| \leq c(n)\epsilon |X| |A|$$

$$|X - A^{-1}| \leq c(n)\epsilon |A^{-1}| |A| |X|,$$

where $c(n)$ is a modest linear function of n ; ϵ is the machine precision; I denotes the identity matrix.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors and $(4/3)n^3$ for complex flavors.

Routines for Matrix Equilibration

Routines described in this section are used to compute scaling factors needed to equilibrate a matrix. Note that these routines do not actually scale the matrices.

?geequ

Computes row and column scaling factors intended to equilibrate a matrix and reduce its condition number.

Syntax

FORTRAN 77:

```
call sgeequ( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
call dgeequ( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
call cgeequ( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
call zgeequ( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
```

Fortran 95:

```
call geequ( a, r, c [,rowcnd] [,colcnd] [,amax] [,info] )
```

Description

This routine computes row and column scalings intended to equilibrate an m -by- n matrix A and reduce its condition number. The output array r returns the row scale factors and the array c the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix B with elements $b_{ij}=r(i)*a_{ij}*c(j)$ have absolute value 1.

See [?laqge](#) auxiliary function that uses scaling factors computed by [?geequ](#).

Input Parameters

m	INTEGER. The number of rows of the matrix A ; $m \geq 0$.
n	INTEGER. The number of columns of the matrix A ; $n \geq 0$.
a	REAL for sgeequ DOUBLE PRECISION for dgeequ COMPLEX for cgeequ DOUBLE COMPLEX for zgeequ . Array: DIMENSION ($lda, *$). Contains the m -by- n matrix A whose equilibration factors are to be computed. The second dimension of a must be at least $\max(1, n)$.

lda INTEGER. The leading dimension of *a*; $lda \geq \max(1, m)$.

Output Parameters

r, c REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Arrays: $r(m)$, $c(n)$.
If $info = 0$, or $info > m$, the array *r* contains the row scale factors of the matrix *A*.
If $info = 0$, the array *c* contains the column scale factors of the matrix *A*.

rowcnd REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
If $info = 0$ or $info > m$, *rowcnd* contains the ratio of the smallest $r(i)$ to the largest $r(i)$.

colcnd REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
If $info = 0$, *colcnd* contains the ratio of the smallest $c(i)$ to the largest $c(i)$.

amax REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Absolute value of the largest element of the matrix *A*.

info INTEGER.
If $info = 0$, the execution is successful.
If $info = -i$, the *i*-th parameter had an illegal value.
If $info = i$ and
 $i \leq m$, the *i*-th row of *A* is exactly zero;
 $i > m$, the $(i-m)$ th column of *A* is exactly zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `geequ` interface are as follows:

a Holds the matrix *A* of size (m, n) .
r Holds the vector of length (m) .

c Holds the vector of length (*n*).

Application Notes

All the components of *r* and *c* are restricted to be between SMLNUM = smallest safe number and BIGNUM= largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of *A* but works well in practice.

If *rowcnd* ≥ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *r*.

If *colcnd* ≥ 0.1 , it is not worth scaling by *c*.

If *amax* is very close to overflow or very close to underflow, the matrix *A* should be scaled.

?gbequ

Computes row and column scaling factors intended to equilibrate a band matrix and reduce its condition number.

Syntax

FORTRAN 77:

```
call sgbequ( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
call dgbequ( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
call cgbequ( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
call zgbequ( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
```

Fortran 95:

```
call gbequ( a, r, c [,kl] [,rowcnd] [,colcnd] [,amax] [,info] )
```

Description

This routine computes row and column scalings intended to equilibrate an *m*-by-*n* band matrix *A* and reduce its condition number. The output array *r* returns the row scale factors and the array *c* the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix *B* with elements $b_{ij}=r(i)*a_{ij}*c(j)$ have absolute value 1.

See ?laqgb auxiliary function that uses scaling factors computed by ?gbequ.

Input Parameters

<i>m</i>	INTEGER. The number of rows of the matrix <i>A</i> ; $m \geq 0$.
<i>n</i>	INTEGER. The number of columns of the matrix <i>A</i> ; $n \geq 0$.
<i>kl</i>	INTEGER. The number of subdiagonals within the band of <i>A</i> ; $kl \geq 0$.
<i>ku</i>	INTEGER. The number of superdiagonals within the band of <i>A</i> ; $ku \geq 0$.
<i>ab</i>	REAL for sgbequ DOUBLE PRECISION for dgbequ COMPLEX for cgbequ DOUBLE COMPLEX for zgbequ. Array, DIMENSION (<i>ldab</i> , *). Contains the original band matrix <i>A</i> stored in rows from 1 to $kl + ku + 1$. The second dimension of <i>ab</i> must be at least $\max(1, n)$;
<i>ldab</i>	INTEGER. The leading dimension of <i>ab</i> ; $ldab \geq kl + ku + 1$.

Output Parameters

<i>r</i> , <i>c</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays: <i>r</i> (<i>m</i>), <i>c</i> (<i>n</i>). If <i>info</i> = 0, or <i>info</i> > <i>m</i> , the array <i>r</i> contains the row scale factors of the matrix <i>A</i> . If <i>info</i> = 0, the array <i>c</i> contains the column scale factors of the matrix <i>A</i> .
<i>rowcnd</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. If <i>info</i> = 0 or <i>info</i> > <i>m</i> , <i>rowcnd</i> contains the ratio of the smallest <i>r</i> (<i>i</i>) to the largest <i>r</i> (<i>i</i>).
<i>colcnd</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. If <i>info</i> = 0, <i>colcnd</i> contains the ratio of the smallest <i>c</i> (<i>i</i>) to the largest <i>c</i> (<i>i</i>).
<i>amax</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

info Absolute value of the largest element of the matrix *A*.
 INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.
 If *info* = *i* and
 i ≤ *m*, the *i*-th row of *A* is exactly zero;
 i > *m*, the (*i*-*m*)th column of *A* is exactly zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gbequ` interface are as follows:

<i>a</i>	Stands for argument <i>ab</i> in Fortran 77 interface. Holds the array <i>A</i> of size (<i>kl</i> + <i>ku</i> +1, <i>n</i>).
<i>r</i>	Holds the vector of length (<i>m</i>).
<i>c</i>	Holds the vector of length (<i>n</i>).
<i>kl</i>	If omitted, assumed <i>kl</i> = <i>ku</i> .
<i>ku</i>	Restored as <i>ku</i> = <i>lda</i> - <i>kl</i> -1.

Application Notes

All the components of *r* and *c* are restricted to be between SMLNUM = smallest safe number and BIGNUM= largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of *A* but works well in practice.

If *rowcnd* ≥ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *r*.

If *colcnd* ≥ 0.1, it is not worth scaling by *c*.

If *amax* is very close to overflow or very close to underflow, the matrix *A* should be scaled.

?poequ

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite matrix and reduce its condition number.

Syntax

FORTRAN 77:

```
call spoequ( n, a, lda, s, scond, amax, info )
call dpoequ( n, a, lda, s, scond, amax, info )
call cpoequ( n, a, lda, s, scond, amax, info )
call zpoequ( n, a, lda, s, scond, amax, info )
```

Fortran 95:

```
call poequ( a, s [,scond] [,amax] [,info] )
```

Description

This routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive-definite matrix A and reduce its condition number (with respect to the two-norm). The output array s returns scale factors computed as

$$s(i) = \frac{1}{\sqrt{a_{i,i}}}$$

These factors are chosen so that the scaled matrix B with elements $b_{ij}=s(i)*a_{ij}*s(j)$ has diagonal elements equal to 1.

This choice of s puts the condition number of B within a factor n of the smallest possible condition number over all possible diagonal scalings.

See [?laqsy](#) auxiliary function that uses scaling factors computed by [?poequ](#).

Input Parameters

<i>n</i>	INTEGER. The order of the matrix <i>A</i> ; $n \geq 0$.
<i>a</i>	REAL for spoequ DOUBLE PRECISION for dpoequ COMPLEX for cpoequ DOUBLE COMPLEX for zpoequ. Array: DIMENSION (<i>lda</i> , *). Contains the <i>n</i> -by- <i>n</i> symmetric or Hermitian positive definite matrix <i>A</i> whose scaling factors are to be computed. Only diagonal elements of <i>A</i> are referenced. The second dimension of <i>a</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $lda \geq \max(1, m)$.

Output Parameters

<i>s</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION (<i>n</i>). If <i>info</i> = 0, the array <i>s</i> contains the scale factors for <i>A</i> .
<i>scond</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. If <i>info</i> = 0, <i>scond</i> contains the ratio of the smallest <i>s</i> (<i>i</i>) to the largest <i>s</i> (<i>i</i>).
<i>amax</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Absolute value of the largest element of the matrix <i>A</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , the <i>i</i> -th diagonal element of <i>A</i> is nonpositive.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `poequ` interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (n, n) .
<i>s</i>	Holds the vector of length (n) .

Application Notes

If $scond \geq 0.1$ and $amax$ is neither too large nor too small, it is not worth scaling by s .

If $amax$ is very close to overflow or very close to underflow, the matrix *A* should be scaled.

?ppequ

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite matrix in packed storage and reduce its condition number.

Syntax

FORTRAN 77:

```
call sppequ( uplo, n, ap, s, scond, amax, info )
call dppequ( uplo, n, ap, s, scond, amax, info )
call cppequ( uplo, n, ap, s, scond, amax, info )
call zppequ( uplo, n, ap, s, scond, amax, info )
```

Fortran 95:

```
call ppequ( a, s [,scond] [,amax] [,uplo] [,info] )
```

Description

This routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive definite matrix *A* in packed storage and reduce its condition number (with respect to the two-norm). The output array *s* returns scale factors computed as

$$s(i) = \frac{1}{\sqrt{a_{i,i}}}$$

These factors are chosen so that the scaled matrix B with elements $b_{ij}=s(i)*a_{ij}*s(j)$ has diagonal elements equal to 1.

This choice of s puts the condition number of B within a factor n of the smallest possible condition number over all possible diagonal scalings.

See [?laqsp](#) auxiliary function that uses scaling factors computed by [?ppequ](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is packed in the array <i>ap</i>: If <i>uplo</i> = 'U', the array <i>ap</i> stores the upper triangular part of the matrix A. If <i>uplo</i> = 'L', the array <i>ap</i> stores the lower triangular part of the matrix A.
<i>n</i>	INTEGER. The order of matrix A; $n \geq 0$.
<i>ap</i>	REAL for sppequ DOUBLE PRECISION for dppequ COMPLEX for cppequ DOUBLE COMPLEX for zppequ. Array, DIMENSION at least $\max(1, n(n+1)/2)$. The array <i>ap</i> contains the upper or the lower triangular part of the matrix A (as specified by <i>uplo</i>) in <i>packed storage</i> (see Matrix Storage Schemes).

Output Parameters

<i>s</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION (n). If <i>info</i> = 0, the array <i>s</i> contains the scale factors for A.
<i>scond</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. If <i>info</i> = 0, <i>scond</i> contains the ratio of the smallest $s(i)$ to the largest $s(i)$.
<i>amax</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Absolute value of the largest element of the matrix A.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.
 If *info* = *i*, the *i*-th diagonal element of *A* is nonpositive.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ppequ` interface are as follows:

a Stands for argument *ap* in Fortran 77 interface. Holds the array *A* of size $(n * (n+1) / 2)$.
s Holds the vector of length (*n*).
uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If *scond* ≥ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *s*.

If *amax* is very close to overflow or very close to underflow, the matrix *A* should be scaled.

?pbequ

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive-definite band matrix and reduce its condition number.

Syntax

FORTRAN 77:

```
call spbequ( uplo, n, kd, ab, ldab, s, acond, amax, info )
call dpbequ( uplo, n, kd, ab, ldab, s, acond, amax, info )
call cpbequ( uplo, n, kd, ab, ldab, s, acond, amax, info )
call zpbequ( uplo, n, kd, ab, ldab, s, acond, amax, info )
```

Fortran 95:

```
call pbequ( a, s [,scond] [,amax] [,uplo] [,info] )
```

Description

This routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive definite matrix *A* in packed storage and reduce its condition number (with respect to the two-norm). The output array *s* returns scale factors computed as

$$s(i) = \frac{1}{\sqrt{a_{i,i}}}$$

These factors are chosen so that the scaled matrix *B* with elements $b_{ij}=s(i)*a_{ij}*s(j)$ has diagonal elements equal to 1. This choice of *s* puts the condition number of *B* within a factor *n* of the smallest possible condition number over all possible diagonal scalings.

See [?laqsb](#) auxiliary function that uses scaling factors computed by [?pbequ](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of <i>A</i> is packed in the array <i>ab</i> : If <i>uplo</i> = 'U', the array <i>ab</i> stores the upper triangular part of the matrix <i>A</i> . If <i>uplo</i> = 'L', the array <i>ab</i> stores the lower triangular part of the matrix <i>A</i> .
<i>n</i>	INTEGER. The order of matrix <i>A</i> ; $n \geq 0$.
<i>kd</i>	INTEGER. The number of superdiagonals or subdiagonals in the matrix <i>A</i> ; $kd \geq 0$.
<i>ab</i>	REAL for spbequ DOUBLE PRECISION for dpbequ COMPLEX for cpbequ DOUBLE COMPLEX for zpbequ .

Array, DIMENSION (*ldab*, *).

The array *ap* contains either the upper or the lower triangular part of the matrix *A* (as specified by *uplo*) in *band storage* (see [Matrix Storage Schemes](#)).

The second dimension of *ab* must be at least $\max(1, n)$.

ldab INTEGER. The leading dimension of the array *ab*; $ldab \geq kd + 1$.

Output Parameters

s REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Array, DIMENSION (*n*).
If *info* = 0, the array *s* contains the scale factors for *A*.

scond REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
If *info* = 0, *scond* contains the ratio of the smallest *s*(*i*) to the largest *s*(*i*).

amax REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Absolute value of the largest element of the matrix *A*.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.
If *info* = *i*, the *i*-th diagonal element of *A* is nonpositive.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *pbequ* interface are as follows:

a Stands for argument *ab* in Fortran 77 interface. Holds the array *A* of size ($kd+1, n$).

s Holds the vector of length (*n*).

uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If $scond \geq 0.1$ and $amax$ is neither too large nor too small, it is not worth scaling by s .

If $amax$ is very close to overflow or very close to underflow, the matrix A should be scaled.

Driver Routines

Table 3-3 lists the LAPACK driver routines for solving systems of linear equations with real or complex matrices.

Table 3-3 Driver Routines for Solving Systems of Linear Equations

Matrix type, storage scheme	Simple Driver	Expert Driver
general	?gesv	?gesvx
general band	?gbsv	?gbsvx
general tridiagonal	?gtsv	?gtsvx
symmetric/Hermitian positive-definite	?posv	?posvx
symmetric/Hermitian positive-definite, storage	?ppsv	?ppsvx
symmetric/Hermitian positive-definite, band	?pbsv	?pbsvx
symmetric/Hermitian positive-definite, tridiagonal	?ptsv	?ptsvx
symmetric/Hermitian indefinite	?sysv/?hesv	?sysvx/?hesvx
symmetric/Hermitian indefinite, packed storage	?spsv/?hpsv	?spsvx/?hpsvx
complex symmetric	?sysv	?sysvx
complex symmetric, packed storage	?spsv	?spsvx

In this table α stands for s (single precision real), d (double precision real), c (single precision complex), or z (double precision complex). In the description of `?gesv` routine α stands for combined character codes ds and zc for the mixed precision subroutines.

?gesv

Computes the solution to the system of linear equations with a square matrix A and multiple right-hand sides.

Syntax

FORTRAN 77:

```
call sgesv( n, nrhs, a, lda, ipiv, b, ldb, info )
call dgesv( n, nrhs, a, lda, ipiv, b, ldb, info )
call cgesv( n, nrhs, a, lda, ipiv, b, ldb, info )
call zgesv( n, nrhs, a, lda, ipiv, b, ldb, info )
call dsgesv( n, nrhs, a, lda, ipiv, b, ldb, x, ldx, work, swork, iter, info )
call zcgesv( n, nrhs, a, lda, ipiv, b, ldb, x, ldx, work, swork, iter, info )
```

Fortran 95:

```
call gesv( a, b [,ipiv] [,info] )
```

Description

This routine solves for X the system of linear equations $A \cdot X = B$, where A is an n -by- n matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The LU decomposition with partial pivoting and row interchanges is used to factor A as $A = P \cdot L \cdot U$, where P is a permutation matrix, L is unit lower triangular, and U is upper triangular. The factored form of A is then used to solve the system of equations $A \cdot X = B$.

The `dsgesv` and `zcgesv` are mixed precision iterative refinement subroutines for exploiting fast single precision hardware. They first attempt to factorize the matrix in single precision (`dsgesv`) or single complex precision (`zcgesv`) and use this factorization within an iterative refinement procedure to produce a solution with double precision (`dsgesv`) / double complex

precision (`zcgesv`) normwise backward error quality (see below). If the approach fails, the method switches to a double precision or double complex precision factorization respectively and computes the solution.

The iterative refinement is not going to be a winning strategy if the ratio single precision performance over double precision performance is too small. A reasonable strategy should take the number of right-hand sides and the size of the matrix into account. This might be done with a call to `ilaenv` in the future. At present, iterative refinement is implemented.

The iterative refinement process is stopped if

```
iter > itermax
```

or for all the right-hand sides:

```
rnrm < sqrt(n)*xnrm*anrm*eps*bwdmax,
```

where

- `iter` is the number of the current iteration in the iterative refinement process
- `rnrm` is the infinity-norm of the residual
- `xnrm` is the infinity-norm of the solution
- `anrm` is the infinity-operator-norm of the matrix A
- `eps` is the machine epsilon returned by `dlamch` ('Epsilon').

The values `itermax` and `bwdmax` are fixed to 30 and 1.0d+00 respectively.

Input Parameters

<code>n</code>	INTEGER. The number of linear equations, that is, the order of the matrix A ; $n \geq 0$.
<code>nrhs</code>	INTEGER. The number of right-hand sides, that is, the number of columns of the matrix B ; $nrhs \geq 0$.
<code>a, b</code>	REAL for <code>sgesv</code> DOUBLE PRECISION for <code>dgesv</code> and <code>dsgesv</code> COMPLEX for <code>cgesv</code> DOUBLE COMPLEX for <code>zgesv</code> and <code>zcgesv</code> . Arrays: <code>a(lda,*)</code> , <code>b(ldb,*)</code> . The array <code>a</code> contains the n -by- n coefficient matrix A .

The array b contains the n -by- $nrhs$ matrix of right hand side matrix B . The second dimension of a must be at least $\max(1, n)$, the second dimension of b at least $\max(1, nrhs)$.

lda INTEGER. The leading dimension of the array a ; $lda \geq \max(1, n)$.

ldb INTEGER. The leading dimension of the array b ; $ldb \geq \max(1, nrhs)$.

ldx INTEGER. The leading dimension of the array x ; $ldx \geq \max(1, n)$.

$work$ DOUBLE PRECISION for `dsgesv`
DOUBLE COMPLEX for `zcgesv`.
Workspace array, DIMENSION at least $\max(1, n*nrhs)$. This array is used to hold the residual vectors.

$swork$ REAL for `dsgesv`
COMPLEX for `zcgesv`.
Workspace array, DIMENSION at least $\max(1, n*nrhs)$. This array is used to use the single precision matrix and the right-hand sides or solutions in single precision.

Output Parameters

a Overwritten by the factors L and U from the factorization of $A = P*L*U$; the unit diagonal elements of L are not stored.
If iterative refinement has been successfully used ($info=0$ and $iter \geq 0$, see description below), then A is unchanged.
If double precision factorization has been used ($info=0$ and $iter < 0$, see description below), then the array A contains the factors L and U from the factorization $A = P*L*U$; the unit diagonal elements of L are not stored.

b Overwritten by the solution matrix X for `dgesv`, `sgesv`, `zgesv`, `zgesv`.
Unchanged for `dsgesv` and `zcgesv`.

$ipiv$ INTEGER.
Array, DIMENSION at least $\max(1, n)$. The pivot indices that define the permutation matrix P ; row i of the matrix was interchanged with row $ipiv(i)$. Corresponds to the single precision factorization (if $info=0$ and $iter \geq 0$) or the double precision factorization (if $info=0$ and $iter < 0$).

x DOUBLE PRECISION for `dsgesv`

iter DOUBLE COMPLEX for *zcgsv*.
 Array, DIMENSION (*ldx*, *nrhs*). If *info* = 0, contains the *n*-by-*nrhs* solution matrix *X*.

iter INTEGER.
 If *iter* < 0: iterative refinement has failed, double precision factorization has been performed

- If *iter* = -1: taking into account machine parameters, *n*, *nrhs*, it is a priori not worth working in single precision
- If *iter* = -2: overflow of an entry when moving from double to single precision
- If *iter* = -3: failure of *sgetrf*
- If *iter* = -31: stop the iterative refinement after the 30th iteration.

If *iter* > 0: iterative refinement has been successfully used. Returns the number of iterations.

info INTEGER. If *info*=0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.
 If *info* = *i*, *U*(*i*, *i*) (computed in double precision for mixed precision subroutines) is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *gesv* interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>nrhs</i>).
<i>ipiv</i>	Holds the vector of length (<i>n</i>).



NOTE. Fortran 95 Interface is so far not available for the mixed precision subroutines *dsgsv*/*zcgsv*.

?gesvx

Computes the solution to the system of linear equations with a square matrix A and multiple right-hand sides, and provides error bounds on the solution.

Syntax

FORTRAN 77:

```
call sgesvx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b,
ldb, x, ldx, rcond, ferr, berr, work, iwork, info )

call dgesvx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b,
ldb, x, ldx, rcond, ferr, berr, work, iwork, info )

call cgesvx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b,
ldb, x, ldx, rcond, ferr, berr, work, rwork, info )

call zgesvx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b,
ldb, x, ldx, rcond, ferr, berr, work, rwork, info )
```

Fortran 95:

```
call gesvx( a, b, x [,af] [,ipiv] [,fact] [,trans] [,equed] [,r] [,c] [,ferr]
[,berr] [,rcond] [,rpvgrw] [,info] )
```

Description

This routine uses the LU factorization to compute the solution to a real or complex system of linear equations $A \cdot X = B$, where A is an n -by- n matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?gesvx performs the following steps:

1. If $fact = 'E'$, real scaling factors r and c are computed to equilibrate the system:

$$trans = 'N': diag(r) \cdot A \cdot diag(c) \cdot inv(diag(c)) \cdot X = diag(r) \cdot B$$

$$trans = 'T': (diag(r) \cdot A \cdot diag(c))^T \cdot inv(diag(r)) \cdot X = diag(c) \cdot B$$

$$trans = 'C': (diag(r) \cdot A \cdot diag(c))^H \cdot inv(diag(r)) \cdot X = diag(c) \cdot B$$

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(r) * A * \text{diag}(c)$ and B by $\text{diag}(r) * B$ (if $\text{trans} = 'N'$) or $\text{diag}(c) * B$ (if $\text{trans} = 'T'$ or $'C'$).

2. If $\text{fact} = 'N'$ or $'E'$, the LU decomposition is used to factor the matrix A (after equilibration if $\text{fact} = 'E'$) as $A = P * L * U$, where P is a permutation matrix, L is a unit lower triangular matrix, and U is upper triangular.
3. If some $U_{i,i} = 0$, so that U is exactly singular, then the routine returns with $\text{info} = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $\text{info} = n + 1$ is returned as a warning, but the routine still goes on to solve for x and compute error bounds as described below.
4. The system of equations is solved for x using the factored form of A .
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix x is premultiplied by $\text{diag}(c)$ (if $\text{trans} = 'N'$) or $\text{diag}(r)$ (if $\text{trans} = 'T'$ or $'C'$) so that it solves the original system before equilibration.

Input Parameters

fact

CHARACTER*1. Must be 'F', 'N', or 'E'.

Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.

If $\text{fact} = 'F'$: on entry, af and $ipiv$ contain the factored form of A .

If equed is not 'N', the matrix A has been equilibrated with scaling factors given by r and c .

a , af , and $ipiv$ are not modified.

If $\text{fact} = 'N'$, the matrix A will be copied to af and factored.

If $\text{fact} = 'E'$, the matrix A will be equilibrated if necessary, then copied to af and factored.

trans

CHARACTER*1. Must be 'N', 'T', or 'C'.

Specifies the form of the system of equations:

If $\text{trans} = 'N'$, the system has the form $A * X = B$ (No transpose).

If $\text{trans} = 'T'$, the system has the form $A^T * X = B$ (Transpose).

If $\text{trans} = 'C'$, the system has the form $A^H * X = B$ (Transpose for real flavors, conjugate transpose for complex flavors).

<i>n</i>	INTEGER. The number of linear equations; the order of the matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right hand sides; the number of columns of the matrices <i>B</i> and <i>X</i> ; $nrhs \geq 0$.
<i>a,af,b,work</i>	REAL for sgesvx DOUBLE PRECISION for dgesvx COMPLEX for cgesvx DOUBLE COMPLEX for zgesvx. Arrays: <i>a</i>(<i>lda</i>,*), <i>af</i>(<i>ldaf</i>,*), <i>b</i>(<i>ldb</i>,*), <i>work</i>(*). The array <i>a</i> contains the matrix <i>A</i>. If <i>fact</i> = 'F' and <i>equed</i> is not 'N', then <i>A</i> must have been equilibrated by the scaling factors in <i>r</i> and/or <i>c</i>. The second dimension of <i>a</i> must be at least $\max(1, n)$. The array <i>af</i> is an input argument if <i>fact</i> = 'F'. It contains the factored form of the matrix <i>A</i>, that is, the factors <i>L</i> and <i>U</i> from the factorization $A = P * L * U$ as computed by ?getrf. If <i>equed</i> is not 'N', then <i>af</i> is the factored form of the equilibrated matrix <i>A</i>. The second dimension of <i>af</i> must be at least $\max(1, n)$. The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$. <i>work</i>(*) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, 4 * n)$ for real flavors, and at least $\max(1, 2 * n)$ for complex flavors.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldaf</i>	INTEGER. The first dimension of <i>af</i> ; $ldaf \geq \max(1, n)$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; $ldb \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The array <i>ipiv</i> is an input argument if <i>fact</i> = 'F'. It contains the pivot indices from the factorization $A = P * L * U$ as computed by ?getrf; row <i>i</i> of the matrix was interchanged with row <i>ipiv</i>(<i>i</i>).
<i>equed</i>	CHARACTER*1. Must be 'N', 'R', 'C', or 'B'. <i>equed</i> is an input argument if <i>fact</i> = 'F'. It specifies the form of equilibration that was done:

If `equed = 'N'`, no equilibration was done (always true if `fact = 'N'`).

If `equed = 'R'`, row equilibration was done, that is, *A* has been premultiplied by `diag(r)`.

If `equed = 'C'`, column equilibration was done, that is, *A* has been postmultiplied by `diag(c)`.

If `equed = 'B'`, both row and column equilibration was done, that is, *A* has been replaced by `diag(r)*A*diag(c)`.

r, c
REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Arrays: `r(n)`, `c(n)`. The array *r* contains the row scale factors for *A*, and the array *c* contains the column scale factors for *A*. These arrays are input arguments if `fact = 'F'` only; otherwise they are output arguments.

If `equed = 'R'` or `'B'`, *A* is multiplied on the left by `diag(r)`; if `equed = 'N'` or `'C'`, *r* is not accessed.

If `fact = 'F'` and `equed = 'R'` or `'B'`, each element of *r* must be positive.

If `equed = 'C'` or `'B'`, *A* is multiplied on the right by `diag(c)`; if `equed = 'N'` or `'R'`, *c* is not accessed.

If `fact = 'F'` and `equed = 'C'` or `'B'`, each element of *c* must be positive.

ldx
INTEGER. The first dimension of the output array *x*; `ldx ≥ max(1, n)`.

iwork
INTEGER. Workspace array, **DIMENSION** at least `max(1, n)`; used in real flavors only.

rwork
REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Workspace array, **DIMENSION** at least `max(1, 2*n)`; used in complex flavors only.

Output Parameters

x
REAL for sgesvx
DOUBLE PRECISION for dgesvx
COMPLEX for cgesvx
DOUBLE COMPLEX for zgesvx.

	Array, DIMENSION ($ldx, *$). If $info = 0$ or $info = n+1$, the array x contains the solution matrix x to the <i>original</i> system of equations. Note that A and B are modified on exit if $equed \neq 'N'$, and the solution to the <i>equilibrated</i> system is: $diag(C)^{-1} * X$, if $trans = 'N'$ and $equed = 'C'$ or $'B'$; $diag(R)^{-1} * X$, if $trans = 'T'$ or $'C'$ and $equed = 'R'$ or $'B'$. The second dimension of x must be at least $\max(1, nrhs)$.
<i>a</i>	Array a is not modified on exit if $fact = 'F'$ or $'N'$, or if $fact = 'E'$ and $equed = 'N'$. If $equed \neq 'N'$, A is scaled on exit as follows: $equed = 'R'$: $A = diag(R) * A$ $equed = 'C'$: $A = A * diag(c)$ $equed = 'B'$: $A = diag(R) * A * diag(c)$.
<i>af</i>	If $fact = 'N'$ or $'E'$, then af is an output argument and on exit returns the factors L and U from the factorization $A = PLU$ of the original matrix A (if $fact = 'N'$) or of the equilibrated matrix A (if $fact = 'E'$). See the description of a for the form of the equilibrated matrix.
<i>b</i>	Overwritten by $diag(r) * B$ if $trans = 'N'$ and $equed = 'R'$ or $'B'$; overwritten by $diag(c) * B$ if $trans = 'T'$ or $'C'$ and $equed = 'C'$ or $'B'$; not changed if $equed = 'N'$.
<i>r, c</i>	These arrays are output arguments if $fact \neq 'F'$. See the description of r, c in <i>Input Arguments</i> section.
<i>rcond</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal condition number of the matrix A after equilibration (if done). If $rcond$ is less than the machine precision, in particular, if $rcond = 0$, the matrix is singular to working precision. This condition is indicated by a return code of $info > 0$.
<i>ferr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector $x(j)$ (the j-th column of the solution matrix X). If $xtrue$ is the true solution corresponding to $x(j)$, $ferr(j)$ is an estimated upper bound for the magnitude of the

	largest element in $(x(j) - x_{true})$ divided by the magnitude of the largest element in $x(j)$. The estimate is as reliable as the estimate for $rcond$, and is almost always a slight overestimate of the true error.
<i>berr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of A or B that makes $x(j)$ an exact solution.
<i>ipiv</i>	If <i>fact</i> = 'N' or 'E', then <i>ipiv</i> is an output argument and on exit contains the pivot indices from the factorization $A = P*L*U$ of the original matrix A (if <i>fact</i> = 'N') or of the equilibrated matrix A (if <i>fact</i> = 'E').
<i>equed</i>	If <i>fact</i> ≠ 'F', then <i>equed</i> is an output argument. It specifies the form of equilibration that was done (see the description of <i>equed</i> in <i>Input Arguments</i> section).
<i>work, rwork</i>	On exit, <i>work</i> (1) for real flavors, or <i>rwork</i> (1) for complex flavors, contains the reciprocal pivot growth factor $\text{norm}(A)/\text{norm}(U)$. The "max absolute element" norm is used. If <i>work</i> (1) for real flavors, or <i>rwork</i> (1) for complex flavors is much less than 1, then the stability of the LU factorization of the (equilibrated) matrix A could be poor. This also means that the solution x , condition estimator <i>rcond</i> , and forward error bound <i>ferr</i> could be unreliable. If factorization fails with $0 < info \leq n$, then <i>work</i> (1) for real flavors, or <i>rwork</i> (1) for complex flavors contains the reciprocal pivot growth factor for the leading <i>info</i> columns of A .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , and $i \leq n$, then $U(i, i)$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; <i>rcond</i> = 0 is returned. If <i>info</i> = <i>i</i> , and $i = n+1$, then U is nonsingular, but <i>rcond</i> is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of <i>rcond</i> would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gesvx` interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (n, n) .
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.
<i>af</i>	Holds the matrix <i>AF</i> of size (n, n) .
<i>ipiv</i>	Holds the vector of length (n) .
<i>r</i>	Holds the vector of length (n) . Default value for each element is $r(i) = 1.0\_WP$.
<i>c</i>	Holds the vector of length (n) . Default value for each element is $c(i) = 1.0\_WP$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>fact</i>	Must be 'N', 'E', or 'F'. The default value is 'N'. If <i>fact</i> = 'F', then both arguments <i>af</i> and <i>ipiv</i> must be present; otherwise, an error is returned.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>equed</i>	Must be 'N', 'B', 'C', or 'R'. The default value is 'N'.
<i>rpvgrw</i>	Real value that contains the reciprocal pivot growth factor $\text{norm}(A)/\text{norm}(U)$.

?gbsv

Computes the solution to the system of linear equations with a band matrix A and multiple right-hand sides.

Syntax

FORTRAN 77:

```
call sgbsv( n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call dgbsv( n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call cgbsv( n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call zgbsv( n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
```

Fortran 95:

```
call gbsv( a, b [,kl] [,ipiv] [,info] )
```

Description

This routine solves for X the real or complex system of linear equations $A^*X = B$, where A is an n -by- n band matrix with kl subdiagonals and ku superdiagonals, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The LU decomposition with partial pivoting and row interchanges is used to factor A as $A = L^*U$, where L is a product of permutation and unit lower triangular matrices with kl subdiagonals, and U is upper triangular with $kl+ku$ superdiagonals. The factored form of A is then used to solve the system of equations $A^*X = B$.

Input Parameters

n	INTEGER. The order of A . The number of rows in B ; $n \geq 0$.
kl	INTEGER. The number of subdiagonals within the band of A ; $kl \geq 0$.
ku	INTEGER. The number of superdiagonals within the band of A ; $ku \geq 0$.
$nrhs$	INTEGER. The number of right-hand sides. The number of columns in B ; $nrhs \geq 0$.
ab, b	REAL for sgbsv DOUBLE PRECISION for dgbsv COMPLEX for cgbsv

DOUBLE COMPLEX for zgbsv.

Arrays: $ab(ldab, *)$, $b ldb, *)$.

The array ab contains the matrix A in band storage (see [Matrix Storage Schemes](#)). The second dimension of ab must be at least $\max(1, n)$. The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.

$ldab$ INTEGER. The first dimension of the array ab . ($ldab \geq 2kl + ku + 1$)

ldb INTEGER. The first dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

ab Overwritten by L and U . The diagonal and $kl + ku$ superdiagonals of U are stored in the first $1 + kl + ku$ rows of ab . The multipliers used to form L are stored in the next kl rows.

b Overwritten by the solution matrix X .

$ipiv$ INTEGER.
Array, DIMENSION at least $\max(1, n)$. The pivot indices: row i was interchanged with row $ipiv(i)$.

$info$ INTEGER. If $info = 0$, the execution is successful.
If $info = -i$, the i -th parameter had an illegal value.
If $info = i$, $U(i, i)$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gbsv` interface are as follows:

a Stands for argument ab in Fortran 77 interface. Holds the array A of size $(2*kl+ku+1, n)$.

b Holds the matrix B of size $(n, nrhs)$.

$ipiv$ Holds the vector of length (n) .

kl If omitted, assumed $kl = ku$.

ku

Restored as $ku = lda - 2 * kl - 1$.

?gbsvx

Computes the solution to the real or complex system of linear equations with a band matrix A and multiple right-hand sides, and provides error bounds on the solution.

Syntax

FORTRAN 77:

```
call sgbsvx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed,
r, c, b, ldb, x, ldx, rcond, ferr, berr, work, iwork, info )

call dgbsvx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed,
r, c, b, ldb, x, ldx, rcond, ferr, berr, work, iwork, info )

call cgbsvx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed,
r, c, b, ldb, x, ldx, rcond, ferr, berr, work, rwork, info )

call zgbsvx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed,
r, c, b, ldb, x, ldx, rcond, ferr, berr, work, rwork, info )
```

Fortran 95:

```
call gbsvx( a, b, x [,kl] [,af] [,ipiv] [,fact] [,trans] [,equed] [,r] [,c]
[,ferr] [,berr] [,rcond] [,rpvgrw] [,info] )
```

Description

This routine uses the LU factorization to compute the solution to a real or complex system of linear equations $A * X = B$, $A^T * X = B$, or $A^H * X = B$, where A is a band matrix of order n with kl subdiagonals and ku superdiagonals, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?gbsvx performs the following steps:

1. If $fact = 'E'$, real scaling factors r and c are computed to equilibrate the system:

$$trans = 'N': diag(r) * A * diag(c) * inv(diag(c)) * X = diag(r) * B$$

$$trans = 'T': (diag(r) * A * diag(c))^T * inv(diag(r)) * X = diag(c) * B$$

$trans = 'C': (diag(r)*A*diag(c))^H * inv(diag(r)) * X = diag(c) * B$

Whether the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $diag(r)*A*diag(c)$ and B by $diag(r)*B$ (if $trans='N'$) or $diag(c)*B$ (if $trans = 'T'$ or $'C'$).

2. If $fact = 'N'$ or $'E'$, the LU decomposition is used to factor the matrix A (after equilibration if $fact = 'E'$) as $A = L*U$, where L is a product of permutation and unit lower triangular matrices with kl subdiagonals, and U is upper triangular with $kl+ku$ superdiagonals.
3. If some $U_{i,i} = 0$, so that U is exactly singular, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $info = n + 1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
4. The system of equations is solved for X using the factored form of A .
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix X is premultiplied by $diag(c)$ (if $trans = 'N'$) or $diag(r)$ (if $trans = 'T'$ or $'C'$) so that it solves the original system before equilibration.

Input Parameters

fact

CHARACTER*1. Must be 'F', 'N', or 'E'.

Specifies whether the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.

If $fact = 'F'$: on entry, *afb* and *ipiv* contain the factored form of A . If *equed* is not 'N', the matrix A is equilibrated with scaling factors given by *r* and *c*.

ab, *afb*, and *ipiv* are not modified.

If $fact = 'N'$, the matrix A will be copied to *afb* and factored.

If $fact = 'E'$, the matrix A will be equilibrated if necessary, then copied to *afb* and factored.

trans

CHARACTER*1. Must be 'N', 'T', or 'C'.

Specifies the form of the system of equations:

If $trans = 'N'$, the system has the form $A*X = B$ (No transpose).

If $trans = 'T'$, the system has the form $A^T*X = B$ (Transpose).

	If <i>trans</i> = 'C', the system has the form $A^H * X = B$ (Transpose for real flavors, conjugate transpose for complex flavors).
<i>n</i>	INTEGER. The number of linear equations, the order of the matrix <i>A</i> ; $n \geq 0$.
<i>kl</i>	INTEGER. The number of subdiagonals within the band of <i>A</i> ; $kl \geq 0$.
<i>ku</i>	INTEGER. The number of superdiagonals within the band of <i>A</i> ; $ku \geq 0$.
<i>nrhs</i>	INTEGER. The number of right hand sides, the number of columns of the matrices <i>B</i> and <i>X</i> ; $nrhs \geq 0$.
<i>ab,afb,b,work</i>	REAL for sgesvx DOUBLE PRECISION for dgesvx COMPLEX for cgesvx DOUBLE COMPLEX for zgesvx. Arrays: <i>a(lda,*)</i> , <i>af(ldaf,*)</i> , <i>b(ldb,*)</i> , <i>work(*)</i> . The array <i>ab</i> contains the matrix <i>A</i> in band storage (see Matrix Storage Schemes). The second dimension of <i>ab</i> must be at least $\max(1, n)$. If <i>fact</i> = 'F' and <i>equed</i> is not 'N', then <i>A</i> must have been equilibrated by the scaling factors in <i>r</i> and/or <i>c</i> . The array <i>afb</i> is an input argument if <i>fact</i> = 'F'. The second dimension of <i>afb</i> must be at least $\max(1, n)$. It contains the factored form of the matrix <i>A</i> , that is, the factors <i>L</i> and <i>U</i> from the factorization $A = L * U$ as computed by ?gbtrf. <i>U</i> is stored as an upper triangular band matrix with $kl + ku$ superdiagonals in the first $1 + kl + ku$ rows of <i>afb</i> . The multipliers used during the factorization are stored in the next <i>kl</i> rows. If <i>equed</i> is not 'N', then <i>afb</i> is the factored form of the equilibrated matrix <i>A</i> . The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$. <i>work(*)</i> is a workspace array. The dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors, and at least $\max(1, 2*n)$ for complex flavors.
<i>ldab</i>	INTEGER. The first dimension of <i>ab</i> ; $ldab \geq kl + ku + 1$.
<i>ldaafb</i>	INTEGER. The first dimension of <i>afb</i> ; $ldaafb \geq 2*kl + ku + 1$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; $ldb \geq \max(1, nrhs)$.

<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The array <i>ipiv</i> is an input argument if <i>fact</i> = 'F'. It contains the pivot indices from the factorization $A = L*U$ as computed by ?gbtrf; row <i>i</i> of the matrix was interchanged with row <i>ipiv</i>(<i>i</i>).
<i>equed</i>	CHARACTER*1. Must be 'N', 'R', 'C', or 'B'. <i>equed</i> is an input argument if <i>fact</i> = 'F'. It specifies the form of equilibration that was done: If <i>equed</i> = 'N', no equilibration was done (always true if <i>fact</i> = 'N'). If <i>equed</i> = 'R', row equilibration was done, that is, <i>A</i> has been premultiplied by <i>diag</i>(<i>r</i>). If <i>equed</i> = 'C', column equilibration was done, that is, <i>A</i> has been postmultiplied by <i>diag</i>(<i>c</i>). if <i>equed</i> = 'B', both row and column equilibration was done, that is, <i>A</i> has been replaced by <i>diag</i>(<i>r</i>)*<i>A</i>*<i>diag</i>(<i>c</i>).
<i>r, c</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays: <i>r</i>(<i>n</i>), <i>c</i>(<i>n</i>). The array <i>r</i> contains the row scale factors for <i>A</i>, and the array <i>c</i> contains the column scale factors for <i>A</i>. These arrays are input arguments if <i>fact</i> = 'F' only; otherwise they are output arguments. If <i>equed</i> = 'R' or 'B', <i>A</i> is multiplied on the left by <i>diag</i>(<i>r</i>); if <i>equed</i> = 'N' or 'C', <i>r</i> is not accessed. If <i>fact</i> = 'F' and <i>equed</i> = 'R' or 'B', each element of <i>r</i> must be positive. If <i>equed</i> = 'C' or 'B', <i>A</i> is multiplied on the right by <i>diag</i>(<i>c</i>); if <i>equed</i> = 'N' or 'R', <i>c</i> is not accessed. If <i>fact</i> = 'F' and <i>equed</i> = 'C' or 'B', each element of <i>c</i> must be positive.
<i>ldx</i>	INTEGER. The first dimension of the output array <i>x</i>; $ldx \geq \max(1, n)$.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$; used in real flavors only.
<i>rwork</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

Workspace array, `DIMENSION` at least `max(1, n)`; used in complex flavors only.

Output Parameters

<code>x</code>	REAL for <code>sghbsvx</code> DOUBLE PRECISION for <code>dghbsvx</code> COMPLEX for <code>cghbsvx</code> DOUBLE COMPLEX for <code>zghbsvx</code>. Array, <code>DIMENSION (ldx,*)</code>. If <code>info = 0</code> or <code>info = n+1</code>, the array <code>x</code> contains the solution matrix <code>x</code> to the <i>original</i> system of equations. Note that <code>A</code> and <code>B</code> are modified on exit if <code>equed</code> $\neq$ 'N', and the solution to the <i>equilibrated</i> system is: <code>inv(diag(c))*X</code>, if <code>trans = 'N'</code> and <code>equed = 'C'</code> or 'B'; <code>inv(diag(r))*X</code>, if <code>trans = 'T'</code> or 'C' and <code>equed = 'R'</code> or 'B'. The second dimension of <code>x</code> must be at least <code>max(1, nrhs)</code>.
<code>ab</code>	Array <code>ab</code> is not modified on exit if <code>fact = 'F'</code> or 'N', or if <code>fact = 'E'</code> and <code>equed = 'N'</code>. If <code>equed</code> $\neq$ 'N', <code>A</code> is scaled on exit as follows: <code>equed = 'R': A = diag(r)*A</code> <code>equed = 'C': A = A*diag(c)</code> <code>equed = 'B': A = diag(r)*A*diag(c)</code>.
<code>afb</code>	If <code>fact = 'N'</code> or 'E', then <code>afb</code> is an output argument and on exit returns details of the <i>LU</i> factorization of the original matrix <code>A</code> (if <code>fact = 'N'</code>) or of the equilibrated matrix <code>A</code> (if <code>fact = 'E'</code>). See the description of <code>ab</code> for the form of the equilibrated matrix.
<code>b</code>	Overwritten by <code>diag(r)*b</code> if <code>trans = 'N'</code> and <code>equed = 'R'</code> or 'B'; overwritten by <code>diag(c)*b</code> if <code>trans = 'T'</code> or 'C' and <code>equed = 'C'</code> or 'B'; not changed if <code>equed = 'N'</code>.
<code>r, c</code>	These arrays are output arguments if <code>fact</code> $\neq$ 'F'. See the description of <code>r, c</code> in <i>Input Arguments</i> section.
<code>rcond</code>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal condition number of the matrix <code>A</code> after equilibration (if done).

	If <i>rcond</i> is less than the machine precision (in particular, if <i>rcond</i> = 0), the matrix is singular to working precision. This condition is indicated by a return code of <i>info</i> > 0.
<i>ferr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector $x(j)$ (the j-th column of the solution matrix X). If <i>xtrue</i> is the true solution corresponding to $x(j)$, <i>ferr</i>(j) is an estimated upper bound for the magnitude of the largest element in $(x(j) - xtrue)$ divided by the magnitude of the largest element in $x(j)$. The estimate is as reliable as the estimate for <i>rcond</i>, and is almost always a slight overestimate of the true error.
<i>berr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of <i>A</i> or <i>B</i> that makes $x(j)$ an exact solution.
<i>ipiv</i>	If <i>fact</i> = 'N' or 'E', then <i>ipiv</i> is an output argument and on exit contains the pivot indices from the factorization $A = L*U$ of the original matrix <i>A</i> (if <i>fact</i> = 'N') or of the equilibrated matrix <i>A</i> (if <i>fact</i> = 'E').
<i>equed</i>	If <i>fact</i> ≠ 'F', then <i>equed</i> is an output argument. It specifies the form of equilibration that was done (see the description of <i>equed</i> in <i>Input Arguments</i> section).
<i>work, rwork</i>	On exit, <i>work</i>(1) for real flavors, or <i>rwork</i>(1) for complex flavors, contains the reciprocal pivot growth factor $\text{norm}(A)/\text{norm}(U)$. The "max absolute element" norm is used. If <i>work</i>(1) for real flavors, or <i>rwork</i>(1) for complex flavors is much less than 1, then the stability of the <i>LU</i> factorization of the (equilibrated) matrix <i>A</i> could be poor. This also means that the solution <i>x</i>, condition estimator <i>rcond</i>, and forward error bound <i>ferr</i> could be unreliable. If factorization fails with $0 < info \leq n$, then <i>work</i>(1) for real flavors, or <i>rwork</i>(1) for complex flavors contains the reciprocal pivot growth factor for the leading <i>info</i> columns of <i>A</i>.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, and $i \leq n$, then $U(i, i)$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; $rcond = 0$ is returned. If $info = i$, and $i = n+1$, then U is nonsingular, but $rcond$ is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of $rcond$ would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gbsvx` interface are as follows:

<i>a</i>	Stands for argument <i>ab</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(kl+ku+1, n)$.
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.
<i>af</i>	Stands for argument <i>ab</i> in Fortran 77 interface. Holds the array <i>AF</i> of size $(2*kl+ku+1, n)$.
<i>ipiv</i>	Holds the vector of length (n) .
<i>r</i>	Holds the vector of length (n) . Default value for each element is $r(i) = 1.0\_WP$.
<i>c</i>	Holds the vector of length (n) . Default value for each element is $c(i) = 1.0\_WP$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>equed</i>	Must be 'N', 'B', 'C', or 'R'. The default value is 'N'.
<i>fact</i>	Must be 'N', 'E', or 'F'. The default value is 'N'. If <i>fact</i> = 'F', then both arguments <i>af</i> and <i>ipiv</i> must be present; otherwise, an error is returned.

<i>rpvgrw</i>	Real value that contains the reciprocal pivot growth factor $\text{norm}(A)/\text{norm}(U)$.
<i>kl</i>	If omitted, assumed $kl = ku$.
<i>ku</i>	Restored as $ku = lda - kl - 1$.

?gtsv

Computes the solution to the system of linear equations with a tridiagonal matrix A and multiple right-hand sides.

Syntax

FORTRAN 77:

```
call sgtsv( n, nrhs, dl, d, du, b, ldb, info )
call dgtsv( n, nrhs, dl, d, du, b, ldb, info )
call cgtsv( n, nrhs, dl, d, du, b, ldb, info )
call zgtsv( n, nrhs, dl, d, du, b, ldb, info )
```

Fortran 95:

```
call gtsv( dl, d, du, b [,info] )
```

Description

This routine solves for X the system of linear equations $A * X = B$, where A is an n -by- n tridiagonal matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions. The routine uses Gaussian elimination with partial pivoting.

Note that the equation $A^T * X = B$ may be solved by interchanging the order of the arguments du and dl .

Input Parameters

<i>n</i>	INTEGER. The order of A , the number of rows in B ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.
<i>dl, d, du, b</i>	REAL for sgtsv DOUBLE PRECISION for dgtsv

COMPLEX for `cgtsv`

DOUBLE COMPLEX for `zgtsv`.

Arrays: `dl(n - 1)`, `d(n)`, `du(n - 1)`, `b(ldb,*)`.

The array `dl` contains the $(n - 1)$ subdiagonal elements of A .

The array `d` contains the diagonal elements of A .

The array `du` contains the $(n - 1)$ superdiagonal elements of A .

The array `b` contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of `b` must be at least $\max(1, nrhs)$.

`ldb`

INTEGER. The first dimension of `b`; $ldb \geq \max(1, n)$.

Output Parameters

`dl`

Overwritten by the $(n-2)$ elements of the second superdiagonal of the upper triangular matrix U from the LU factorization of A . These elements are stored in `dl(1)`, $\dots$, `dl(n-2)`.

`d`

Overwritten by the n diagonal elements of U .

`du`

Overwritten by the $(n-1)$ elements of the first superdiagonal of U .

`b`

Overwritten by the solution matrix X .

`info`

INTEGER. If `info` = 0, the execution is successful.

If `info` = $-i$, the i -th parameter had an illegal value.

If `info` = i , $U(i, i)$ is exactly zero, and the solution has not been computed. The factorization has not been completed unless $i = n$.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gtsv` interface are as follows:

`dl`

Holds the vector of length $(n-1)$.

`d`

Holds the vector of length (n) .

`du`

Holds the vector of length $(n-1)$.

`b`

Holds the matrix B of size $(n, nrhs)$.

?gtsvx

Computes the solution to the real or complex system of linear equations with a tridiagonal matrix A and multiple right-hand sides, and provides error bounds on the solution.

Syntax

FORTRAN 77:

```
call sgtsvx( fact, trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb,
x, ldx, rcond, ferr, berr, work, iwork, info )

call dgtsvx( fact, trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb,
x, ldx, rcond, ferr, berr, work, iwork, info )

call cgtsvx( fact, trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb,
x, ldx, rcond, ferr, berr, work, rwork, info )

call zgtsvx( fact, trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb,
x, ldx, rcond, ferr, berr, work, rwork, info )
```

Fortran 95:

```
call gtsvx( dl, d, du, b, x [,dlf] [,df] [,duf] [,du2] [,ipiv] [,fact] [,trans]
[,ferr] [,berr] [,rcond] [,info] )
```

Description

This routine uses the LU factorization to compute the solution to a real or complex system of linear equations $A^*X = B$, $A^T X = B$, or $A^H X = B$, where A is a tridiagonal matrix of order n , the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?gtsvx performs the following steps:

1. If $fact = 'N'$, the LU decomposition is used to factor the matrix A as $A = L^*U$, where L is a product of permutation and unit lower bidiagonal matrices and U is an upper triangular matrix with nonzeros in only the main diagonal and first two superdiagonals.

2. If some $U_{i,i} = 0$, so that U is exactly singular, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $info = n + 1$ is returned as a warning, but the routine still goes on to solve for x and compute error bounds as described below.
3. The system of equations is solved for x using the factored form of A .
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

<i>fact</i>	CHARACTER*1. Must be 'F' or 'N'. Specifies whether or not the factored form of the matrix A has been supplied on entry. If <i>fact</i> = 'F': on entry, <i>dlf</i>, <i>df</i>, <i>duf</i>, <i>du2</i>, and <i>ipiv</i> contain the factored form of A; arrays <i>dl</i>, <i>d</i>, <i>du</i>, <i>dlf</i>, <i>df</i>, <i>duf</i>, <i>du2</i>, and <i>ipiv</i> will not be modified. If <i>fact</i> = 'N', the matrix A will be copied to <i>dlf</i>, <i>df</i>, and <i>duf</i> and factored.
<i>trans</i>	CHARACTER*1. Must be 'N', 'T', or 'C'. Specifies the form of the system of equations: If <i>trans</i> = 'N', the system has the form $A^*X = B$ (No transpose). If <i>trans</i> = 'T', the system has the form $A^T * X = B$ (Transpose). If <i>trans</i> = 'C', the system has the form $A^H * X = B$ (Conjugate transpose).
<i>n</i>	INTEGER. The number of linear equations, the order of the matrix A; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right hand sides, the number of columns of the matrices B and X; $nrhs \geq 0$.
<i>dl,d,du,dlf,df,duf,du2,b,</i>	REAL for sgtsvx DOUBLE PRECISION for dgtsvx COMPLEX for cgtsvx
<i>x,work</i>	DOUBLE COMPLEX for zgtsvx. Arrays: <i>dl</i>, dimension $(n - 1)$, contains the subdiagonal elements of A. <i>d</i>, dimension (n), contains the diagonal elements of A. <i>du</i>, dimension $(n - 1)$, contains the superdiagonal elements of A.

d1f, dimension $(n-1)$. If *fact* = 'F' , then *d1f* is an input argument and on entry contains the $(n-1)$ multipliers that define the matrix *L* from the *LU* factorization of *A* as computed by [?gttrf](#).

df, dimension (n) . If *fact* = 'F' , then *df* is an input argument and on entry contains the n diagonal elements of the upper triangular matrix *U* from the *LU* factorization of *A*.

duf, dimension $(n-1)$. If *fact* = 'F' , then *duf* is an input argument and on entry contains the $(n-1)$ elements of the first superdiagonal of *U*.

du2, dimension $(n-2)$. If *fact* = 'F' , then *du2* is an input argument and on entry contains the $(n-2)$ elements of the second superdiagonal of *U*.

b(*ldb**) contains the right-hand side matrix *B*. The second dimension of *b* must be at least $\max(1, nrhs)$.

x(*ldx**) contains the solution matrix *X*. The second dimension of *x* must be at least $\max(1, nrhs)$.

work(*) is a workspace array;

the dimension of *work* must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

ldb INTEGER. The first dimension of *b*; $ldb \geq \max(1, n)$.

ldx INTEGER. The first dimension of *x*; $ldx \geq \max(1, n)$.

ipiv INTEGER.

Array, DIMENSION at least $\max(1, n)$. If *fact* = 'F' , then *ipiv* is an input argument and on entry contains the pivot indices, as returned by [?gttrf](#).

iwork INTEGER. Workspace array, DIMENSION (n) . Used for real flavors only.

rwork REAL for *cgtsvx*

DOUBLE PRECISION for *zgtsvx*.

Workspace array, DIMENSION (n) . Used for complex flavors only.

Output Parameters

x REAL for *sgtsvx*
 DOUBLE PRECISION for *dgtsvx*
 COMPLEX for *cgtsvx*
 DOUBLE COMPLEX for *zgtsvx*.
 Array, DIMENSION $(ldx, *)$.

	If $info = 0$ or $info = n+1$, the array x contains the solution matrix X . The second dimension of x must be at least $\max(1, nrhs)$.
<i>d1f</i>	If $fact = 'N'$, then <i>d1f</i> is an output argument and on exit contains the $(n-1)$ multipliers that define the matrix L from the LU factorization of A .
<i>df</i>	If $fact = 'N'$, then <i>df</i> is an output argument and on exit contains the n diagonal elements of the upper triangular matrix U from the LU factorization of A .
<i>d1f</i>	If $fact = 'N'$, then <i>d1f</i> is an output argument and on exit contains the $(n-1)$ elements of the first superdiagonal of U .
<i>du2</i>	If $fact = 'N'$, then <i>du2</i> is an output argument and on exit contains the $(n-2)$ elements of the second superdiagonal of U .
<i>ipiv</i>	The array <i>ipiv</i> is an output argument if $fact = 'N'$ and, on exit, contains the pivot indices from the factorization $A = L*U$; row i of the matrix was interchanged with row <i>ipiv</i> (i). The value of <i>ipiv</i> (i) will always be i or $i+1$; <i>ipiv</i> (i)= i indicates a row interchange was not required.
<i>rcond</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal condition number of the matrix A . If <i>rcond</i> is less than the machine precision (in particular, if <i>rcond</i> =0), the matrix is singular to working precision. This condition is indicated by a return code of $info>0$.
<i>ferr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector $x(j)$ (the j -th column of the solution matrix X). If x_{true} is the true solution corresponding to $x(j)$, <i>ferr</i> (j) is an estimated upper bound for the magnitude of the largest element in $(x(j) - x_{true})$ divided by the magnitude of the largest element in $x(j)$. The estimate is as reliable as the estimate for <i>rcond</i> , and is almost always a slight overestimate of the true error.
<i>berr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

Array, DIMENSION at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of A or B that makes $x(j)$ an exact solution.

info

INTEGER. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, and $i \leq n$, then $U(i, i)$ is exactly zero. The factorization has not been completed unless $i = n$, but the factor U is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned. If *info* = *i*, and $i = n + 1$, then U is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gtsvx` interface are as follows:

<i>dl</i>	Holds the vector of length $(n-1)$.
<i>d</i>	Holds the vector of length (n) .
<i>du</i>	Holds the vector of length $(n-1)$.
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>x</i>	Holds the matrix X of size $(n, nrhs)$.
<i>dlf</i>	Holds the vector of length $(n-1)$.
<i>df</i>	Holds the vector of length (n) .
<i>duf</i>	Holds the vector of length $(n-1)$.
<i>du2</i>	Holds the vector of length $(n-2)$.
<i>ipiv</i>	Holds the vector of length (n) .
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.

<i>fact</i>	Must be 'N' or 'F'. The default value is 'N'. If <i>fact</i> = 'F', then the arguments <i>dlf</i> , <i>df</i> , <i>duf</i> , <i>du2</i> , and <i>ipiv</i> must be present; otherwise, an error is returned.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.

?posv

Computes the solution to the system of linear equations with a symmetric or Hermitian positive-definite matrix A and multiple right-hand sides.

Syntax

FORTRAN 77:

```
call sposv( uplo, n, nrhs, a, lda, b, ldb, info )
call dposv( uplo, n, nrhs, a, lda, b, ldb, info )
call cposv( uplo, n, nrhs, a, lda, b, ldb, info )
call zposv( uplo, n, nrhs, a, lda, b, ldb, info )
```

Fortran 95:

```
call posv( a, b [,uplo] [,info] )
```

Description

This routine solves for X the real or complex system of linear equations $A * X = B$, where A is an n -by- n symmetric/Hermitian positive-definite matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The Cholesky decomposition is used to factor A as

$A = U^T * U$ (real flavors) and $A = U^H * U$ (complex flavors), if *uplo* = 'U'

or $A = L * L^T$ (real flavors) and $A = L * L^H$ (complex flavors), if *uplo* = 'L',

where U is an upper triangular matrix and L is a lower triangular matrix. The factored form of A is then used to solve the system of equations $A * X = B$.

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.

	Indicates whether the upper or lower triangular part of A is stored: If $uplo = 'U'$, the upper triangle of A is stored. If $uplo = 'L'$, the lower triangle of A is stored.
n	INTEGER. The order of matrix A ; $n \geq 0$.
$nrhs$	INTEGER. The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.
a, b	REAL for <code>sposv</code> DOUBLE PRECISION for <code>dposv</code> COMPLEX for <code>cposv</code> DOUBLE COMPLEX for <code>zposv</code> . Arrays: $a(lda, *)$, $b(l db, *)$. The array a contains the upper or the lower triangular part of the matrix A (see $uplo$). The second dimension of a must be at least $\max(1, n)$. The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.
lda	INTEGER. The first dimension of a ; $lda \geq \max(1, n)$.
ldb	INTEGER. The first dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

a	If $info = 0$, the upper or lower triangular part of a is overwritten by the Cholesky factor U or L , as specified by $uplo$.
b	Overwritten by the solution matrix X .
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value. If $info = i$, the leading minor of order i (and therefore the matrix A itself) is not positive definite, so the factorization could not be completed, and the solution has not been computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `posv` interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (n, n) .
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

?posvx

*Uses the Cholesky factorization to compute the solution to the system of linear equations with a symmetric or Hermitian positive definite matrix *A*, and provides error bounds on the solution.*

Syntax

FORTRAN 77:

```
call sposvx( fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x, ldx,
rcond, ferr, berr, work, iwork, info )

call dposvx( fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x, ldx,
rcond, ferr, berr, work, iwork, info )

call cposvx( fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x, ldx,
rcond, ferr, berr, work, rwork, info )

call zposvx( fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x, ldx,
rcond, ferr, berr, work, rwork, info )
```

Fortran 95:

```
call posvx( a, b, x [,uplo] [,af] [,fact] [,equed] [,s] [,ferr] [,berr] [,rcond]
[,info] )
```

Description

This routine uses the Cholesky factorization $A=U^T*U$ (real flavors) / $A=U^H*U$ (complex flavors) or $A=L*L^T$ (real flavors) / $A=L*L^H$ (complex flavors) to compute the solution to a real or complex system of linear equations $A*X = B$, where *A* is a *n*-by-*n* real symmetric/Hermitian positive definite matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *x* are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?posvx performs the following steps:

1. If $fact = 'E'$, real scaling factors s are computed to equilibrate the system:

$$diag(s) * A * diag(s) * inv(diag(s)) * X = diag(s) * B.$$

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $diag(s) * A * diag(s)$ and B by $diag(s) * B$.

2. If $fact = 'N'$ or $'E'$, the Cholesky decomposition is used to factor the matrix A (after equilibration if $fact = 'E'$) as

$$A = U^T * U \text{ (real)}, A = U^H * U \text{ (complex)}, \text{ if } uplo = 'U',$$

$$\text{or } A = L * L^T \text{ (real)}, A = L * L^H \text{ (complex)}, \text{ if } uplo = 'L',$$

where U is an upper triangular matrix and L is a lower triangular matrix.

3. If the leading i -by- i principal minor is not positive-definite, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $info = n + 1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
4. The system of equations is solved for X using the factored form of A .
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix X is premultiplied by $diag(s)$ so that it solves the original system before equilibration.

Input Parameters

$fact$

CHARACTER*1. Must be 'F', 'N', or 'E'.

Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.

If $fact = 'F'$: on entry, af contains the factored form of A . If $equed = 'Y'$, the matrix A has been equilibrated with scaling factors given by s .

a and af will not be modified.

If $fact = 'N'$, the matrix A will be copied to af and factored.

If $fact = 'E'$, the matrix A will be equilibrated if necessary, then copied to af and factored.

$uplo$

CHARACTER*1. Must be 'U' or 'L'.

	Indicates whether the upper or lower triangular part of A is stored: If $uplo = 'U'$, the upper triangle of A is stored. If $uplo = 'L'$, the lower triangle of A is stored.
n	INTEGER. The order of matrix A ; $n \geq 0$.
$nrhs$	INTEGER. The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.
$a, af, b, work$	REAL for <code>sposvx</code> DOUBLE PRECISION for <code>dposvx</code> COMPLEX for <code>cposvx</code> DOUBLE COMPLEX for <code>zposvx</code> . Arrays: $a(lda, *)$, $af(ldaf, *)$, $b(ldb, *)$, $work(*)$. The array a contains the matrix A as specified by $uplo$. If $fact = 'F'$ and $equed = 'Y'$, then A must have been equilibrated by the scaling factors in s , and a must contain the equilibrated matrix $diag(s) * A * diag(s)$. The second dimension of a must be at least $\max(1, n)$. The array af is an input argument if $fact = 'F'$. It contains the triangular factor U or L from the Cholesky factorization of A in the same storage format as A . If $equed$ is not $'N'$, then af is the factored form of the equilibrated matrix $diag(s) * A * diag(s)$. The second dimension of af must be at least $\max(1, n)$. The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$. $work(*)$ is a workspace array. The dimension of $work$ must be at least $\max(1, 3*n)$ for real flavors, and at least $\max(1, 2*n)$ for complex flavors.
lda	INTEGER. The first dimension of a ; $lda \geq \max(1, n)$.
$ldaf$	INTEGER. The first dimension of af ; $ldaf \geq \max(1, n)$.
ldb	INTEGER. The first dimension of b ; $ldb \geq \max(1, n)$.
$equed$	CHARACTER*1. Must be $'N'$ or $'Y'$. $equed$ is an input argument if $fact = 'F'$. It specifies the form of equilibration that was done: if $equed = 'N'$, no equilibration was done (always true if $fact = 'N'$);

	if <i>equed</i> = 'Y', equilibration was done, that is, <i>A</i> has been replaced by $\text{diag}(s) * A * \text{diag}(s)$.
<i>s</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION (<i>n</i>). The array <i>s</i> contains the scale factors for <i>A</i> . This array is an input argument if <i>fact</i> = 'F' only; otherwise it is an output argument. If <i>equed</i> = 'N', <i>s</i> is not accessed. If <i>fact</i> = 'F' and <i>equed</i> = 'Y', each element of <i>s</i> must be positive.
<i>ldx</i>	INTEGER. The first dimension of the output array <i>x</i> ; $\text{ldx} \geq \max(1, n)$.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$; used in real flavors only.
<i>rwork</i>	REAL for cposvx DOUBLE PRECISION for zposvx. Workspace array, DIMENSION at least $\max(1, n)$; used in complex flavors only.

Output Parameters

<i>x</i>	REAL for sposvx DOUBLE PRECISION for dposvx COMPLEX for cposvx DOUBLE COMPLEX for zposvx. Array, DIMENSION (<i>ldx</i> , *). If <i>info</i> = 0 or <i>info</i> = <i>n</i> +1, the array <i>x</i> contains the solution matrix <i>X</i> to the <i>original</i> system of equations. Note that if <i>equed</i> = 'Y', <i>A</i> and <i>B</i> are modified on exit, and the solution to the equilibrated system is $\text{inv}(\text{diag}(s)) * X$. The second dimension of <i>x</i> must be at least $\max(1, \text{nrhs})$.
<i>a</i>	Array <i>a</i> is not modified on exit if <i>fact</i> = 'F' or 'N', or if <i>fact</i> = 'E' and <i>equed</i> = 'N'. If <i>fact</i> = 'E' and <i>equed</i> = 'Y', <i>A</i> is overwritten by $\text{diag}(s) * A * \text{diag}(s)$.
<i>af</i>	If <i>fact</i> = 'N' or 'E', then <i>af</i> is an output argument and on exit returns the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $A = U^T * U$ or $A = L * L^T$ (real routines), $A = U^H * U$ or $A = L * L^H$ (complex routines)

	of the original matrix A (if $fact = 'N'$), or of the equilibrated matrix A (if $fact = 'E'$). See the description of a for the form of the equilibrated matrix.
b	Overwritten by $diag(s) * B$, if $equed = 'Y'$; not changed if $equed = 'N'$.
s	This array is an output argument if $fact \neq 'F'$. See the description of s in <i>Input Arguments</i> section.
$rcond$	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal condition number of the matrix A after equilibration (if done). If $rcond$ is less than the machine precision (in particular, if $rcond = 0$), the matrix is singular to working precision. This condition is indicated by a return code of $info > 0$.
$ferr$	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector $x(j)$ (the j -th column of the solution matrix X). If $xtrue$ is the true solution corresponding to $x(j)$, $ferr(j)$ is an estimated upper bound for the magnitude of the largest element in $(x(j) - xtrue)$ divided by the magnitude of the largest element in $x(j)$. The estimate is as reliable as the estimate for $rcond$, and is almost always a slight overestimate of the true error.
$berr$	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of A or B that makes $x(j)$ an exact solution.
$equed$	If $fact \neq 'F'$, then $equed$ is an output argument. It specifies the form of equilibration that was done (see the description of $equed$ in <i>Input Arguments</i> section).
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, and $i \leq n$, the leading minor of order i (and therefore the matrix A itself) is not positive-definite, so the factorization could not be completed, and the solution and error bounds could not be computed; $rcond = 0$ is returned.

If $info = i$, and $i = n + 1$, then U is nonsingular, but $rcond$ is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of $rcond$ would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `posvx` interface are as follows:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>x</i>	Holds the matrix X of size $(n, nrhs)$.
<i>af</i>	Holds the matrix AF of size (n, n) .
<i>s</i>	Holds the vector of length (n) . Default value for each element is $s(i) = 1.0\_WP$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>fact</i>	Must be 'N', 'E', or 'F'. The default value is 'N'. If <i>fact</i> = 'F', then <i>af</i> must be present; otherwise, an error is returned.
<i>equed</i>	Must be 'N' or 'Y'. The default value is 'N'.

?ppsv

Computes the solution to the system of linear equations with a symmetric (Hermitian) positive definite packed matrix A and multiple right-hand sides.

Syntax

FORTRAN 77:

```
call sppsv( uplo, n, nrhs, ap, b, ldb, info )
call dppsv( uplo, n, nrhs, ap, b, ldb, info )
call cppsv( uplo, n, nrhs, ap, b, ldb, info )
call zppsv( uplo, n, nrhs, ap, b, ldb, info )
```

Fortran 95:

```
call ppsv( a, b [,uplo] [,info] )
```

Description

This routine solves for X the real or complex system of linear equations $A^*X = B$, where A is an n -by- n real symmetric/Hermitian positive-definite matrix stored in packed format, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The Cholesky decomposition is used to factor A as

$A = U^T * U$ (real flavors) and $A = U^H * U$ (complex flavors), if $uplo = 'U'$

or $A = L * L^T$ (real flavors) and $A = L * L^H$ (complex flavors), if $uplo = 'L'$,

where U is an upper triangular matrix and L is a lower triangular matrix. The factored form of A is then used to solve the system of equations $A^*X = B$.

Input Parameters

<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If $uplo = 'U'$, the upper triangle of A is stored. If $uplo = 'L'$, the lower triangle of A is stored.
<code>n</code>	INTEGER. The order of matrix A ; $n \geq 0$.

<i>nrhs</i>	INTEGER. The number of right-hand sides, the number of columns in <i>B</i> ; $nrhs \geq 0$.
<i>ap</i> , <i>b</i>	REAL for <i>sppsv</i> DOUBLE PRECISION for <i>dppsv</i> COMPLEX for <i>cppsv</i> DOUBLE COMPLEX for <i>zppsv</i> . Arrays: <i>ap</i> (*), <i>b</i> (<i>ldb</i> ,*). The array <i>ap</i> contains the upper or the lower triangular part of the matrix <i>A</i> (as specified by <i>uplo</i>) in <i>packed storage</i> (see Matrix Storage Schemes). The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$. The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; $ldb \geq \max(1, n)$.

Output Parameters

<i>ap</i>	If <i>info</i> = 0, the upper or lower triangular part of <i>A</i> in packed storage is overwritten by the Cholesky factor <i>U</i> or <i>L</i> , as specified by <i>uplo</i> .
<i>b</i>	Overwritten by the solution matrix <i>X</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , the leading minor of order <i>i</i> (and therefore the matrix <i>A</i> itself) is not positive-definite, so the factorization could not be completed, and the solution has not been computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *ppsv* interface are as follows:

<i>a</i>	Stands for argument <i>ap</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(n*(n+1)/2)$.
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

?ppsvx

Uses the Cholesky factorization to compute the solution to the system of linear equations with a symmetric (Hermitian) positive definite packed matrix A , and provides error bounds on the solution.

Syntax

FORTRAN 77:

```
call sppsvx( fact, uplo, n, nrhs, ap, afp, equed, s, b, ldb, x, ldx, rcond,
ferr, berr, work, iwork, info )

call dppsvx( fact, uplo, n, nrhs, ap, afp, equed, s, b, ldb, x, ldx, rcond,
ferr, berr, work, iwork, info )

call cppsvx( fact, uplo, n, nrhs, ap, afp, equed, s, b, ldb, x, ldx, rcond,
ferr, berr, work, rwork, info )

call zppsvx( fact, uplo, n, nrhs, ap, afp, equed, s, b, ldb, x, ldx, rcond,
ferr, berr, work, rwork, info )
```

Fortran 95:

```
call ppsvx( a, b, x [,uplo] [,af] [,fact] [,equed] [,s] [,ferr] [,berr] [,rcond]
[,info] )
```

Description

This routine uses the Cholesky factorization $A=U^T*U$ (real flavors) / $A=U^H*U$ (complex flavors) or $A=L*L^T$ (real flavors) / $A=L*L^H$ (complex flavors) to compute the solution to a real or complex system of linear equations $A*X = B$, where A is a n -by- n symmetric or Hermitian positive-definite matrix stored in packed format, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?ppsvx performs the following steps:

1. If $fact = 'E'$, real scaling factors s are computed to equilibrate the system:

$$diag(s)*A*diag(s)*inv(diag(s))*X = diag(s)*B.$$

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(s) * A * \text{diag}(s)$ and B by $\text{diag}(s) * B$.

- 2.** If $fact = 'N'$ or $'E'$, the Cholesky decomposition is used to factor the matrix A (after equilibration if $fact = 'E'$) as

$A = U^T * U$ (real), $A = U^H * U$ (complex), if $uplo = 'U'$,

or $A = L * L^T$ (real), $A = L * L^H$ (complex), if $uplo = 'L'$,

where U is an upper triangular matrix and L is a lower triangular matrix.

- 3.** If the leading i -by- i principal minor is not positive-definite, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $info = n+1$ is returned as a warning, but the routine still goes on to solve for x and compute error bounds as described below.
- 4.** The system of equations is solved for x using the factored form of A .
- 5.** Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
- 6.** If equilibration was used, the matrix x is premultiplied by $\text{diag}(s)$ so that it solves the original system before equilibration.

Input Parameters

fact

CHARACTER*1. Must be 'F', 'N', or 'E'.

Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.

If $fact = 'F'$: on entry, *afp* contains the factored form of A . If *eques* = 'Y', the matrix A has been equilibrated with scaling factors given by s .

ap and *afp* will not be modified.

If $fact = 'N'$, the matrix A will be copied to *afp* and factored.

If $fact = 'E'$, the matrix A will be equilibrated if necessary, then copied to *afp* and factored.

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of A is stored:

If $uplo = 'U'$, the upper triangle of A is stored.

	If <i>uplo</i> = 'L', the lower triangle of <i>A</i> is stored.
<i>n</i>	INTEGER. The order of matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns in <i>B</i> ; $nrhs \geq 0$.
<i>ap,afp,b,work</i>	REAL for <i>sppsvx</i> DOUBLE PRECISION for <i>dppsvx</i> COMPLEX for <i>cppsvx</i> DOUBLE COMPLEX for <i>zppsvx</i>. Arrays: <i>ap</i>(*), <i>afp</i>(*), <i>b</i>(<i>ldb</i>,*), <i>work</i>(*). The array <i>ap</i> contains the upper or lower triangle of the original symmetric/Hermitian matrix <i>A</i> in <i>packed storage</i> (see Matrix Storage Schemes). In case when <i>fact</i> = 'F' and <i>equed</i> = 'Y', <i>ap</i> must contain the equilibrated matrix $diag(s) * A * diag(s)$. The array <i>afp</i> is an input argument if <i>fact</i> = 'F' and contains the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization of <i>A</i> in the same storage format as <i>A</i>. If <i>equed</i> is not 'N', then <i>afp</i> is the factored form of the equilibrated matrix <i>A</i>. The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations. <i>work</i>(*) is a workspace array. The dimension of arrays <i>ap</i> and <i>afp</i> must be at least $\max(1, n(n+1)/2)$; the second dimension of <i>b</i> must be at least $\max(1, nrhs)$; the dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; $ldb \geq \max(1, n)$.
<i>equed</i>	CHARACTER*1. Must be 'N' or 'Y'. <i>equed</i> is an input argument if <i>fact</i> = 'F'. It specifies the form of equilibration that was done: if <i>equed</i> = 'N', no equilibration was done (always true if <i>fact</i> = 'N'); if <i>equed</i> = 'Y', equilibration was done, that is, <i>A</i> has been replaced by $diag(s) * A * diag(s)$.
<i>s</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

Array, `DIMENSION (n)`. The array s contains the scale factors for A . This array is an input argument if $fact = 'F'$ only; otherwise it is an output argument.

If $equed = 'N'$, s is not accessed.

If $fact = 'F'$ and $equed = 'Y'$, each element of s must be positive.

ldx INTEGER. The first dimension of the output array x ; $ldx \geq \max(1, n)$.

$iwork$ INTEGER. Workspace array, `DIMENSION` at least $\max(1, n)$; used in real flavors only.

$rwork$ REAL for `cppsvx`;
DOUBLE PRECISION for `zppsvx`.
Workspace array, `DIMENSION` at least $\max(1, n)$; used in complex flavors only.

Output Parameters

x REAL for `sppsvx`
DOUBLE PRECISION for `dppsvx`
COMPLEX for `cppsvx`
DOUBLE COMPLEX for `zppsvx`.
Array, `DIMENSION (ldx, *)`.
If $info = 0$ or $info = n+1$, the array x contains the solution matrix X to the *original* system of equations. Note that if $equed = 'Y'$, A and B are modified on exit, and the solution to the equilibrated system is $\text{inv}(\text{diag}(s)) * X$. The second dimension of x must be at least $\max(1, nrhs)$.

ap Array ap is not modified on exit if $fact = 'F'$ or $'N'$, or if $fact = 'E'$ and $equed = 'N'$.
If $fact = 'E'$ and $equed = 'Y'$, A is overwritten by $\text{diag}(s) * A * \text{diag}(s)$.

afp If $fact = 'N'$ or $'E'$, then afp is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization $A = U^T * U$ or $A = L * L^T$ (real routines), $A = U^H * U$ or $A = L * L^H$ (complex routines) of the original matrix A (if $fact = 'N'$), or of the equilibrated matrix A (if $fact = 'E'$). See the description of ap for the form of the equilibrated matrix.

<i>b</i>	Overwritten by $diag(s) * B$, if <i>equed</i> = 'Y'; not changed if <i>equed</i> = 'N'.
<i>s</i>	This array is an output argument if <i>fact</i> ≠ 'F'. See the description of <i>s</i> in <i>Input Arguments</i> section.
<i>rcond</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal condition number of the matrix <i>A</i> after equilibration (if done). If <i>rcond</i> is less than the machine precision (in particular, if <i>rcond</i> = 0), the matrix is singular to working precision. This condition is indicated by a return code of <i>info</i> > 0.
<i>ferr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector $x(j)$ (the <i>j</i> -th column of the solution matrix <i>X</i>). If <i>xtrue</i> is the true solution corresponding to $x(j)$, <i>ferr</i> (<i>j</i>) is an estimated upper bound for the magnitude of the largest element in $(x(j) - xtrue)$ divided by the magnitude of the largest element in $x(j)$. The estimate is as reliable as the estimate for <i>rcond</i> , and is almost always a slight overestimate of the true error.
<i>berr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of <i>A</i> or <i>B</i> that makes $x(j)$ an exact solution.
<i>equed</i>	If <i>fact</i> ≠ 'F', then <i>equed</i> is an output argument. It specifies the form of equilibration that was done (see the description of <i>equed</i> in <i>Input Arguments</i> section).
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , and $i \leq n$, the leading minor of order <i>i</i> (and therefore the matrix <i>A</i> itself) is not positive-definite, so the factorization could not be completed, and the solution and error bounds could not be computed; <i>rcond</i> = 0 is returned.

If $info = i$, and $i = n + 1$, then U is nonsingular, but $rcond$ is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of $rcond$ would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ppsvx` interface are as follows:

<i>a</i>	Stands for argument <i>ap</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(n * (n+1) / 2)$.
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.
<i>af</i>	Stands for argument <i>afp</i> in Fortran 77 interface. Holds the matrix <i>AF</i> of size $(n * (n+1) / 2)$.
<i>s</i>	Holds the vector of length (n) . Default value for each element is $s(i) = 1.0\_WP$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>fact</i>	Must be 'N', 'E', or 'F'. The default value is 'N'. If <i>fact</i> = 'F', then <i>af</i> must be present; otherwise, an error is returned.
<i>equed</i>	Must be 'N' or 'Y'. The default value is 'N'.

?pbsv

Computes the solution to the system of linear equations with a symmetric or Hermitian positive-definite band matrix A and multiple right-hand sides.

Syntax

FORTRAN 77:

```
call spbsv( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call dpbsv( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call cpbsv( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call zpbsv( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
```

Fortran 95:

```
call pbsv( a, b [,uplo] [,info] )
```

Description

This routine solves for X the real or complex system of linear equations $A^*X = B$, where A is an n -by- n symmetric/Hermitian positive definite band matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The Cholesky decomposition is used to factor A as

$A = U^T * U$ (real flavors) and $A = U^H * U$ (complex flavors), if $uplo = 'U'$

or $A = L * L^T$ (real flavors) and $A = L * L^H$ (complex flavors), if $uplo = 'L'$,

where U is an upper triangular band matrix and L is a lower triangular band matrix, with the same number of superdiagonals or subdiagonals as A . The factored form of A is then used to solve the system of equations $A^*X = B$.

Input Parameters

<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If $uplo = 'U'$, the upper triangle of A is stored. If $uplo = 'L'$, the lower triangle of A is stored.
<code>n</code>	INTEGER. The order of matrix A ; $n \geq 0$.

<i>kd</i>	INTEGER. The number of superdiagonals of the matrix <i>A</i> if <i>uplo</i> = 'U', or the number of subdiagonals if <i>uplo</i> = 'L'; $kd \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, the number of columns in <i>B</i> ; $nrhs \geq 0$.
<i>ab, b</i>	REAL for spbsv DOUBLE PRECISION for dpbsv COMPLEX for cpbsv DOUBLE COMPLEX for zpbsv. Arrays: <i>ab</i> (<i>ldab</i> , *), <i>b</i> (<i>ldb</i> , *). The array <i>ab</i> contains the upper or the lower triangular part of the matrix <i>A</i> (as specified by <i>uplo</i>) in <i>band storage</i> (see Matrix Storage Schemes). The second dimension of <i>ab</i> must be at least $\max(1, n)$. The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$.
<i>ldab</i>	INTEGER. The first dimension of the array <i>ab</i> ; $ldab \geq kd + 1$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; $ldb \geq \max(1, n)$.

Output Parameters

<i>ab</i>	The upper or lower triangular part of <i>A</i> (in band storage) is overwritten by the Cholesky factor <i>U</i> or <i>L</i> , as specified by <i>uplo</i> , in the same storage format as <i>A</i> .
<i>b</i>	Overwritten by the solution matrix <i>x</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , the leading minor of order <i>i</i> (and therefore the matrix <i>A</i> itself) is not positive-definite, so the factorization could not be completed, and the solution has not been computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *pbsv* interface are as follows:

<i>a</i>	Stands for argument <i>ab</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(kd+1, n)$.
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

?pbsvx

*Uses the Cholesky factorization to compute the solution to the system of linear equations with a symmetric (Hermitian) positive-definite band matrix *A*, and provides error bounds on the solution.*

Syntax

FORTRAN 77:

```
call spbsvx( fact, uplo, n, kd, nrhs, ab, ldab, afb, ldafb, equed, s, b, ldb,
x, ldx, rcond, ferr, berr, work, iwork, info )

call dpbsvx( fact, uplo, n, kd, nrhs, ab, ldab, afb, ldafb, equed, s, b, ldb,
x, ldx, rcond, ferr, berr, work, iwork, info )

call cpbsvx( fact, uplo, n, kd, nrhs, ab, ldab, afb, ldafb, equed, s, b, ldb,
x, ldx, rcond, ferr, berr, work, iwork, info )

call zpbsvx( fact, uplo, n, kd, nrhs, ab, ldab, afb, ldafb, equed, s, b, ldb,
x, ldx, rcond, ferr, berr, work, iwork, info )
```

Fortran 95:

```
call pbsvx( a, b, x [,uplo] [,af] [,fact] [,equed] [,s] [,ferr] [,berr] [,rcond]
[,info] )
```

Description

This routine uses the Cholesky factorization $A=U^T*U$ (real flavors) / $A=U^H*U$ (complex flavors) or $A=L*L^T$ (real flavors) / $A=L*L^H$ (complex flavors) to compute the solution to a real or complex system of linear equations $A*X = B$, where *A* is a *n*-by-*n* symmetric or Hermitian positive definite band matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *x* are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?pbsvx` performs the following steps:

1. If `fact = 'E'`, real scaling factors s are computed to equilibrate the system:

$$\text{diag}(s) * A * \text{diag}(s) * \text{inv}(\text{diag}(s)) * X = \text{diag}(s) * B.$$

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(s) * A * \text{diag}(s)$ and B by $\text{diag}(s) * B$.

2. If `fact = 'N'` or `'E'`, the Cholesky decomposition is used to factor the matrix A (after equilibration if `fact = 'E'`) as

$$A = U^T * U \text{ (real)}, A = U^H * U \text{ (complex)}, \text{ if } \text{uplo} = 'U',$$

$$\text{or } A = L * L^T \text{ (real)}, A = L * L^H \text{ (complex)}, \text{ if } \text{uplo} = 'L',$$

where U is an upper triangular band matrix and L is a lower triangular band matrix.

3. If the leading i -by- i principal minor is not positive definite, then the routine returns with `info = i`. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, `info = n+1` is returned as a warning, but the routine still goes on to solve for x and compute error bounds as described below.
4. The system of equations is solved for x using the factored form of A .
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix x is premultiplied by $\text{diag}(s)$ so that it solves the original system before equilibration.

Input Parameters

`fact`

CHARACTER*1. Must be 'F', 'N', or 'E'.

Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.

If `fact = 'E'`: on entry, `afb` contains the factored form of A . If `equed = 'Y'`, the matrix A has been equilibrated with scaling factors given by s .

`ab` and `afb` will not be modified.

If `fact = 'N'`, the matrix A will be copied to `afb` and factored.

	If <i>fact</i> = 'E', the matrix <i>A</i> will be equilibrated if necessary, then copied to <i>afb</i> and factored.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of <i>A</i> is stored: If <i>uplo</i> = 'U', the upper triangle of <i>A</i> is stored. If <i>uplo</i> = 'L', the lower triangle of <i>A</i> is stored.
<i>n</i>	INTEGER. The order of matrix <i>A</i>; $n \geq 0$.
<i>kd</i>	INTEGER. The number of superdiagonals or subdiagonals in the matrix <i>A</i>; $kd \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, the number of columns in <i>B</i>; $nrhs \geq 0$.
<i>ab,afb,b,work</i>	REAL for spbsvx DOUBLE PRECISION for dpbsvx COMPLEX for cpbsvx DOUBLE COMPLEX for zpbsvx. Arrays: <i>ab</i>(<i>ldab</i>,*), <i>afb</i>(<i>ldab</i>,*), <i>b</i>(<i>ldb</i>,*), <i>work</i>(*). The array <i>ab</i> contains the upper or lower triangle of the matrix <i>A</i> in <i>band storage</i> (see Matrix Storage Schemes). If <i>fact</i> = 'F' and <i>equed</i> = 'Y', then <i>ab</i> must contain the equilibrated matrix <i>diag</i>(<i>s</i>)*<i>A</i>*<i>diag</i>(<i>s</i>). The second dimension of <i>ab</i> must be at least $\max(1, n)$. The array <i>afb</i> is an input argument if <i>fact</i> = 'F'. It contains the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization of the band matrix <i>A</i> in the same storage format as <i>A</i>. If <i>equed</i> = 'Y', then <i>afb</i> is the factored form of the equilibrated matrix <i>A</i>. The second dimension of <i>afb</i> must be at least $\max(1, n)$. The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$. <i>work</i>(*) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors, and at least $\max(1, 2*n)$ for complex flavors.
<i>ldab</i>	INTEGER. The first dimension of <i>ab</i>; $ldab \geq kd+1$.
<i>ldafb</i>	INTEGER. The first dimension of <i>afb</i>; $ldafb \geq kd+1$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i>; $ldb \geq \max(1, n)$.

<i>equed</i>	CHARACTER*1. Must be 'N' or 'Y'. <i>equed</i> is an input argument if <i>fact</i> = 'F'. It specifies the form of equilibration that was done: if <i>equed</i> = 'N', no equilibration was done (always true if <i>fact</i> = 'N') if <i>equed</i> = 'Y', equilibration was done, that is, <i>A</i> has been replaced by $diag(s) * A * diag(s)$.
<i>s</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION (<i>n</i>). The array <i>s</i> contains the scale factors for <i>A</i>. This array is an input argument if <i>fact</i> = 'F' only; otherwise it is an output argument. If <i>equed</i> = 'N', <i>s</i> is not accessed. If <i>fact</i> = 'F' and <i>equed</i> = 'Y', each element of <i>s</i> must be positive.
<i>ldx</i>	INTEGER. The first dimension of the output array <i>x</i>; $ldx \geq \max(1, n)$.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$; used in real flavors only.
<i>rwork</i>	REAL for cpbsvx DOUBLE PRECISION for zpbsvx. Workspace array, DIMENSION at least $\max(1, n)$; used in complex flavors only.

Output Parameters

<i>x</i>	REAL for spbsvx DOUBLE PRECISION for dpbsvx COMPLEX for cpbsvx DOUBLE COMPLEX for zpbsvx. Array, DIMENSION (<i>ldx</i>, *). If <i>info</i> = 0 or <i>info</i> = <i>n</i>+1, the array <i>x</i> contains the solution matrix <i>X</i> to the <i>original</i> system of equations. Note that if <i>equed</i> = 'Y', <i>A</i> and <i>B</i> are modified on exit, and the solution to the equilibrated system is $inv(diag(s)) * X$. The second dimension of <i>x</i> must be at least $\max(1, nrhs)$.
<i>ab</i>	On exit, if <i>fact</i> = 'E' and <i>equed</i> = 'Y', <i>A</i> is overwritten by $diag(s) * A * diag(s)$.

<i>afb</i>	If <i>fact</i> = 'N' or 'E', then <i>afb</i> is an output argument and on exit returns the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $A=U^T*U$ or $A=L*L^T$ (real routines), $A=U^H*U$ or $A=L*L^H$ (complex routines) of the original matrix <i>A</i> (if <i>fact</i> = 'N'), or of the equilibrated matrix <i>A</i> (if <i>fact</i> = 'E'). See the description of <i>ab</i> for the form of the equilibrated matrix.
<i>b</i>	Overwritten by <i>diag(s)*B</i> , if <i>equed</i> = 'Y'; not changed if <i>equed</i> = 'N'.
<i>s</i>	This array is an output argument if <i>fact</i> ≠ 'F'. See the description of <i>s</i> in <i>Input Arguments</i> section.
<i>rcond</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal condition number of the matrix <i>A</i> after equilibration (if done). If <i>rcond</i> is less than the machine precision (in particular, if <i>rcond</i> = 0), the matrix is singular to working precision. This condition is indicated by a return code of <i>info</i> > 0.
<i>ferr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector <i>x(j)</i> (the <i>j</i> -th column of the solution matrix <i>X</i>). If <i>xtrue</i> is the true solution corresponding to <i>x(j)</i> , <i>ferr(j)</i> is an estimated upper bound for the magnitude of the largest element in (<i>x(j)</i> - <i>xtrue</i>) divided by the magnitude of the largest element in <i>x(j)</i> . The estimate is as reliable as the estimate for <i>rcond</i> , and is almost always a slight overestimate of the true error.
<i>berr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector <i>x(j)</i> , that is, the smallest relative change in any element of <i>A</i> or <i>B</i> that makes <i>x(j)</i> an exact solution.
<i>equed</i>	If <i>fact</i> ≠ 'F', then <i>equed</i> is an output argument. It specifies the form of equilibration that was done (see the description of <i>equed</i> in <i>Input Arguments</i> section).
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

If $info = i$, and $i \leq n$, the leading minor of order i (and therefore the matrix A itself) is not positive definite, so the factorization could not be completed, and the solution and error bounds could not be computed; $rcond = 0$ is returned. If $info = i$, and $i = n + 1$, then U is nonsingular, but $rcond$ is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of $rcond$ would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `pbsvx` interface are as follows:

<i>a</i>	Stands for argument <i>ab</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(kd+1, n)$.
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.
<i>af</i>	Stands for argument <i>afb</i> in Fortran 77 interface. Holds the array <i>AF</i> of size $(kd+1, n)$.
<i>s</i>	Holds the vector of length (n) . Default value for each element is $s(i) = 1.0\_WP$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>fact</i>	Must be 'N', 'E', or 'F'. The default value is 'N'. If <i>fact</i> = 'F', then <i>af</i> must be present; otherwise, an error is returned.
<i>equed</i>	Must be 'N' or 'Y'. The default value is 'N'.

?ptsv

Computes the solution to the system of linear equations with a symmetric or Hermitian positive definite tridiagonal matrix A and multiple right-hand sides.

Syntax

FORTRAN 77:

```
call sptsv( n, nrhs, d, e, b, ldb, info )
call dptsv( n, nrhs, d, e, b, ldb, info )
call cptsv( n, nrhs, d, e, b, ldb, info )
call zptsv( n, nrhs, d, e, b, ldb, info )
```

Fortran 95:

```
call ptsv( d, e, b [,info] )
```

Description

This routine solves for X the real or complex system of linear equations $A^*X = B$, where A is an n -by- n symmetric/Hermitian positive-definite tridiagonal matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

A is factored as $A = L^*D^*L^T$ (real flavors) or $A = L^*D^*L^H$ (complex flavors), and the factored form of A is then used to solve the system of equations $A^*X = B$.

Input Parameters

n	INTEGER. The order of matrix A ; $n \geq 0$.
$nrhs$	INTEGER. The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.
d	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, dimension at least $\max(1, n)$. Contains the diagonal elements of the tridiagonal matrix A .
e, b	REAL for sptsv DOUBLE PRECISION for dptsv

COMPLEX for `cptsv`

DOUBLE COMPLEX for `zptsv`.

Arrays: $e(n - 1)$, $b(l\text{db}, *)$. The array e contains the $(n - 1)$ subdiagonal elements of A .

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.

ldb

INTEGER. The first dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

d

Overwritten by the n diagonal elements of the diagonal matrix D from the $L^*D^*L^T$ (real)/ $L^*D^*L^H$ (complex) factorization of A .

e

Overwritten by the $(n - 1)$ subdiagonal elements of the unit bidiagonal factor L from the factorization of A .

b

Overwritten by the solution matrix X .

$info$

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, the leading minor of order i (and therefore the matrix A itself) is not positive-definite, and the solution has not been computed. The factorization has not been completed unless $i = n$.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ptsv` interface are as follows:

d

Holds the vector of length (n) .

e

Holds the vector of length $(n-1)$.

b

Holds the matrix B of size $(n, nrhs)$.

?ptsvx

Uses factorization to compute the solution to the system of linear equations with a symmetric (Hermitian) positive definite tridiagonal matrix A , and provides error bounds on the solution.

Syntax

FORTRAN 77:

```
call sptsvx( fact, n, nrhs, d, e, df, ef, b, ldb, x, ldx, rcond, ferr, berr,
work, info )

call dptsvx( fact, n, nrhs, d, e, df, ef, b, ldb, x, ldx, rcond, ferr, berr,
work, info )

call cptsvx( fact, n, nrhs, d, e, df, ef, b, ldb, x, ldx, rcond, ferr, berr,
work, rwork, info )

call zptsvx( fact, n, nrhs, d, e, df, ef, b, ldb, x, ldx, rcond, ferr, berr,
work, rwork, info )
```

Fortran 95:

```
call ptsvx( d, e, b, x [,df] [,ef] [,fact] [,ferr] [,berr] [,rcond] [,info] )
```

Description

This routine uses the Cholesky factorization $A = L^*D*L^T$ (real)/ $A = L^*D*L^H$ (complex) to compute the solution to a real or complex system of linear equations $A*X = B$, where A is a n -by- n symmetric or Hermitian positive definite tridiagonal matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?ptsvx performs the following steps:

1. If *fact* = 'N', the matrix A is factored as $A = L^*D*L^T$ (real flavors)/ $A = L^*D*L^H$ (complex flavors), where L is a unit lower bidiagonal matrix and D is diagonal. The factorization can also be regarded as having the form $A = U^T*D*U$ (real flavors)/ $A = U^H*D*U$ (complex flavors).

2. If the leading i -by- i principal minor is not positive-definite, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $info = n+1$ is returned as a warning, but the routine still goes on to solve for x and compute error bounds as described below.
3. The system of equations is solved for x using the factored form of A .
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

<i>fact</i>	CHARACTER*1. Must be 'F' or 'N'. Specifies whether or not the factored form of the matrix A is supplied on entry. If <i>fact</i> = 'F': on entry, <i>df</i> and <i>ef</i> contain the factored form of A. Arrays <i>d</i>, <i>e</i>, <i>df</i>, and <i>ef</i> will not be modified. If <i>fact</i> = 'N', the matrix A will be copied to <i>df</i> and <i>ef</i>, and factored.
<i>n</i>	INTEGER. The order of matrix A; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, the number of columns in B; $nrhs \geq 0$.
<i>d</i> , <i>df</i> , <i>rwork</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays: <i>d</i>(<i>n</i>), <i>df</i>(<i>n</i>), <i>rwork</i>(<i>n</i>). The array <i>d</i> contains the n diagonal elements of the tridiagonal matrix A. The array <i>df</i> is an input argument if <i>fact</i> = 'F' and on entry contains the n diagonal elements of the diagonal matrix D from the $L^*D^*L^T$ (real)/$L^*D^*L^H$ (complex) factorization of A. The array <i>rwork</i> is a workspace array used for complex flavors only.
<i>e</i> , <i>ef</i> , <i>b</i> , <i>work</i>	REAL for sptsvx DOUBLE PRECISION for dptsvx COMPLEX for cptsvx DOUBLE COMPLEX for zptsvx. Arrays: <i>e</i>($n-1$), <i>ef</i>($n-1$), <i>b</i>(<i>ldb</i>*), <i>work</i>(*). The array <i>e</i> contains the $(n-1)$ subdiagonal elements of the tridiagonal matrix A.

The array *ef* is an input argument if *fact* = 'F' and on entry contains the $(n - 1)$ subdiagonal elements of the unit bidiagonal factor *L* from the L^*D*L^T (real)/ L^*D*L^H (complex) factorization of *A*.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.

The array *work* is a workspace array. The dimension of *work* must be at least $2*n$ for real flavors, and at least n for complex flavors.

ldb INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

ldx INTEGER. The leading dimension of *x*; $ldx \geq \max(1, n)$.

Output Parameters

x REAL for sptsvx
DOUBLE PRECISION for dptsvx
COMPLEX for cptsvx
DOUBLE COMPLEX for zptsvx.
Array, DIMENSION (*ldx*, *).
If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *x* to the system of equations. The second dimension of *x* must be at least $\max(1, nrhs)$.

df, *ef* These arrays are output arguments if *fact* = 'N'. See the description of *df*, *ef* in *Input Arguments* section.

rcond REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
An estimate of the reciprocal condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.

ferr REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Array, DIMENSION at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector *x*(*j*) (the *j*-th column of the solution matrix *X*). If *xtrue* is the true solution corresponding to *x*(*j*), *ferr*(*j*) is an estimated upper bound for the magnitude of the largest element in $(x(j) - xtrue)$ divided by the magnitude of the largest element in *x*(*j*). The estimate is as reliable as the estimate for *rcond*, and is almost always a slight overestimate of the true error.

<i>berr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of A or B that makes $x(j)$ an exact solution.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i-th parameter had an illegal value. If $info = i$, and $i \leq n$, the leading minor of order i (and therefore the matrix A itself) is not positive-definite, so the factorization could not be completed, and the solution and error bounds could not be computed; $rcond = 0$ is returned. If $info = i$, and $i = n + 1$, then U is nonsingular, but $rcond$ is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of $rcond$ would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ptsvx` interface are as follows:

<i>d</i>	Holds the vector of length (n) .
<i>e</i>	Holds the vector of length $(n-1)$.
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>x</i>	Holds the matrix X of size $(n, nrhs)$.
<i>df</i>	Holds the vector of length (n) .
<i>ef</i>	Holds the vector of length $(n-1)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>fact</i>	Must be 'N' or 'F'. The default value is 'N'. If $fact = 'F'$, then both arguments <i>af</i> and <i>ipiv</i> must be present; otherwise, an error is returned.

?sysv

Computes the solution to the system of linear equations with a real or complex symmetric matrix A and multiple right-hand sides.

Syntax

FORTRAN 77:

```
call ssysv( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call dsysv( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call csysv( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call zsysv( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
```

Fortran 95:

```
call sysv( a, b [,uplo] [,ipiv] [,info] )
```

Description

This routine solves for X the real or complex system of linear equations $A^*X = B$, where A is an n -by- n symmetric matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The diagonal pivoting method is used to factor A as $A = U^*D^*U^T$ or $A = L^*D^*L^T$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of A is then used to solve the system of equations $A^*X = B$.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns in B ; $nrhs \geq 0$.
<i>a, b, work</i>	REAL for ssysv

DOUBLE PRECISION for `dsysv`

COMPLEX for `csysv`

DOUBLE COMPLEX for `zsysv`.

Arrays: `a(lda,*)`, `b(ldb,*)`, `work(*)`.

The array `a` contains the upper or the lower triangular part of the symmetric matrix A (see `uplo`). The second dimension of `a` must be at least $\max(1, n)$.

The array `b` contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of `b` must be at least $\max(1, nrhs)$.

`work` is a workspace array, dimension at least $\max(1, lwork)$.

`lda` INTEGER. The first dimension of `a`; $lda \geq \max(1, n)$.

`ldb` INTEGER. The first dimension of `b`; $ldb \geq \max(1, n)$.

`lwork` INTEGER. The size of the `work` array; $lwork \geq 1$.

If `lwork` = -1, then a workspace query is assumed; the routine only calculates the optimal size of the `work` array, returns this value as the first entry of the `work` array, and no error message related to `lwork` is issued by `xerbla`. See *Application Notes* below for details and for the suggested value of `lwork`.

Output Parameters

`a` If `info` = 0, `a` is overwritten by the block-diagonal matrix D and the multipliers used to obtain the factor U (or L) from the factorization of A as computed by `?sytrf`.

`b` If `info` = 0, `b` is overwritten by the solution matrix X .

`ipiv` INTEGER.
Array, DIMENSION at least $\max(1, n)$. Contains details of the interchanges and the block structure of D , as determined by `?sytrf`.
If `ipiv(i)` = $k > 0$, then d_{ii} is a 1-by-1 diagonal block, and the i -th row and column of A was interchanged with the k -th row and column.
If `uplo` = 'U' and `ipiv(i)` = `ipiv(i-1)` = $-m < 0$, then D has a 2-by-2 block in rows/columns i and $i-1$, and $(i-1)$ -th row and column of A was interchanged with the m -th row and column.

	If $uplo = 'L'$ and $ipiv(i) = ipiv(i+1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i+1$, and $(i+1)$ -th row and column of A was interchanged with the m -th row and column.
<code>work(1)</code>	If $info = 0$, on exit <code>work(1)</code> contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.
<code>info</code>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value. If $info = i$, d_{ii} is 0. The factorization has been completed, but D is exactly singular, so the solution could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sysv` interface are as follows:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>b</code>	Holds the matrix B of size $(n, nrhs)$.
<code>ipiv</code>	Holds the vector of length (n) .
<code>uplo</code>	Must be <code>'U'</code> or <code>'L'</code> . The default value is <code>'U'</code> .

Application Notes

For better performance, try using $lwork = n * blocksize$, where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?sysvx

Uses the diagonal pivoting factorization to compute the solution to the system of linear equations with a real or complex symmetric matrix A , and provides error bounds on the solution.

Syntax

FORTRAN 77:

```
call ssysvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx,
             rcond, ferr, berr, work, lwork, iwork, info )

call dsysvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx,
             rcond, ferr, berr, work, lwork, iwork, info )

call csysvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx,
             rcond, ferr, berr, work, lwork, rwork, info )

call zsysvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx,
             rcond, ferr, berr, work, lwork, rwork, info )
```

Fortran 95:

```
call sysvx( a, b, x [,uplo] [,af] [,ipiv] [,fact] [,ferr] [,berr] [,rcond]
             [,info] )
```

Description

This routine uses the diagonal pivoting factorization to compute the solution to a real or complex system of linear equations $A \cdot X = B$, where A is a n -by- n symmetric matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?sysvx performs the following steps:

1. If $fact = 'N'$, the diagonal pivoting method is used to factor the matrix A . The form of the factorization is $A = U*D*U^T$ or $A = L*D*L^T$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.
2. If some $d_{i,i} = 0$, so that D is exactly singular, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $info = n+1$ is returned as a warning, but the routine still goes on to solve for x and compute error bounds as described below.
3. The system of equations is solved for x using the factored form of A .
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

<i>fact</i>	CHARACTER*1. Must be 'F' or 'N'. Specifies whether or not the factored form of the matrix A has been supplied on entry. If $fact = 'F'$: on entry, af and $ipiv$ contain the factored form of A. Arrays a, af, and $ipiv$ will not be modified. If $fact = 'N'$, the matrix A will be copied to af and factored.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If $uplo = 'U'$, the upper triangle of A is stored. If $uplo = 'L'$, the lower triangle of A is stored.
<i>n</i>	INTEGER. The order of matrix A; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, the number of columns in B; $nrhs \geq 0$.
<i>a,af,b,work</i>	REAL for <code>ssysvx</code> DOUBLE PRECISION for <code>dsysvx</code> COMPLEX for <code>csysvx</code> DOUBLE COMPLEX for <code>zsysvx</code>. Arrays: $a(lda,*)$, $af(ldaf,*)$, $b(ldb,*)$, $work(*)$. The array a contains the upper or the lower triangular part of the symmetric matrix A (see <i>uplo</i>). The second dimension of a must be at least $\max(1, n)$.

The array *af* is an input argument if *fact* = 'F'. It contains the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* from the factorization $A = U * D * U^T$ or $A = L * D * L^T$ as computed by [?sytrf](#). The second dimension of *af* must be at least $\max(1, n)$. The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least $\max(1, nrhs)$.

work(*) is a workspace array, dimension at least $\max(1, lwork)$.

<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldaf</i>	INTEGER. The first dimension of <i>af</i> ; $ldaf \geq \max(1, n)$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; $ldb \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The array <i>ipiv</i> is an input argument if <i>fact</i> = 'F'. It contains details of the interchanges and the block structure of <i>D</i> , as determined by ?sytrf . If $ipiv(i) = k > 0$, then d_{ii} is a 1-by-1 diagonal block, and the <i>i</i> -th row and column of <i>A</i> was interchanged with the <i>k</i> -th row and column. If $uplo = 'U'$ and $ipiv(i) = ipiv(i-1) = -m < 0$, then <i>D</i> has a 2-by-2 block in rows/columns <i>i</i> and <i>i-1</i> , and (<i>i-1</i>)-th row and column of <i>A</i> was interchanged with the <i>m</i> -th row and column. If $uplo = 'L'$ and $ipiv(i) = ipiv(i+1) = -m < 0$, then <i>D</i> has a 2-by-2 block in rows/columns <i>i</i> and <i>i+1</i> , and (<i>i+1</i>)-th row and column of <i>A</i> was interchanged with the <i>m</i> -th row and column.
<i>ldx</i>	INTEGER. The leading dimension of the output array <i>x</i> ; $ldx \geq \max(1, n)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <i>xerbla</i> . See <i>Application Notes</i> below for details and for the suggested value of <i>lwork</i> .
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$; used in real flavors only.
<i>rwork</i>	REAL for <i>csysvx</i> ; DOUBLE PRECISION for <i>zsysvx</i> .

Workspace array, `DIMENSION` at least `max(1, n)`; used in complex flavors only.

Output Parameters

<code>x</code>	REAL for <code>ssysvx</code> DOUBLE PRECISION for <code>dsysvx</code> COMPLEX for <code>csysvx</code> DOUBLE COMPLEX for <code>zsysvx</code>. Array, <code>DIMENSION (ldx, *)</code>. If <code>info = 0</code> or <code>info = n+1</code>, the array <code>x</code> contains the solution matrix <code>x</code> to the system of equations. The second dimension of <code>x</code> must be at least <code>max(1, nrhs)</code>.
<code>af, ipiv</code>	These arrays are output arguments if <code>fact = 'N'</code>. See the description of <code>af, ipiv</code> in <i>Input Arguments</i> section.
<code>rcond</code>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal condition number of the matrix <code>A</code>. If <code>rcond</code> is less than the machine precision (in particular, if <code>rcond = 0</code>), the matrix is singular to working precision. This condition is indicated by a return code of <code>info > 0</code>.
<code>ferr</code>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, <code>DIMENSION</code> at least <code>max(1, nrhs)</code>. Contains the estimated forward error bound for each solution vector <code>x(j)</code> (the <code>j</code>-th column of the solution matrix <code>X</code>). If <code>xtrue</code> is the true solution corresponding to <code>x(j)</code>, <code>ferr(j)</code> is an estimated upper bound for the magnitude of the largest element in <code>(x(j) - xtrue)</code> divided by the magnitude of the largest element in <code>x(j)</code>. The estimate is as reliable as the estimate for <code>rcond</code>, and is almost always a slight overestimate of the true error.
<code>berr</code>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, <code>DIMENSION</code> at least <code>max(1, nrhs)</code>. Contains the component-wise relative backward error for each solution vector <code>x(j)</code>, that is, the smallest relative change in any element of <code>A</code> or <code>B</code> that makes <code>x(j)</code> an exact solution.
<code>work(1)</code>	If <code>info=0</code>, on exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance. Use this <code>lwork</code> for subsequent runs.

info INTEGER. If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.
 If *info* = *i*, and $i \leq n$, then d_{ii} is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.
 If *info* = *i*, and $i = n + 1$, then *D* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *sysvx* interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (n, n) .
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.
<i>af</i>	Holds the matrix <i>AF</i> of size (n, n) .
<i>ipiv</i>	Holds the vector of length (n) .
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>fact</i>	Must be 'N' or 'F'. The default value is 'N'. If <i>fact</i> = 'F', then both arguments <i>af</i> and <i>ipiv</i> must be present; otherwise, an error is returned.

Application Notes

For real flavors, *lwork* must be at least $3*n$, and for complex flavors at least $2*n$. For better performance, try using $lwork = n*blocksize$, where *blocksize* is the optimal block size for ?sytrf.

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible `lwork` sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

zhesv

Computes the solution to the system of linear equations with a Hermitian matrix A and multiple right-hand sides.

Syntax

FORTRAN 77:

```
call chesv( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call zhesv( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
```

Fortran 95:

```
call hesv( a, b [,uplo] [,ipiv] [,info] )
```

Description

This routine solves for X the complex system of linear equations $A * X = B$, where A is an n -by- n symmetric matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The diagonal pivoting method is used to factor A as $A = U * D * U^H$ or $A = L * D * L^H$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of A is then used to solve the system of equations $A * X = B$.

Input Parameters

<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored and how A is factored: If <code>uplo</code> = 'U', the array a stores the upper triangular part of the matrix A, and A is factored as $U^*D^*U^H$. If <code>uplo</code> = 'L', the array a stores the lower triangular part of the matrix A, and A is factored as $L^*D^*L^H$.
<code>n</code>	INTEGER. The order of matrix A; $n \geq 0$.
<code>nrhs</code>	INTEGER. The number of right-hand sides, the number of columns in B; $nrhs \geq 0$.
<code>a, b, work</code>	COMPLEX for <code>chesv</code> DOUBLE COMPLEX for <code>zhesv</code>. Arrays: <code>a(lda,*)</code>, <code>b(ldb,*)</code>, <code>work(*)</code>. The array a contains the upper or the lower triangular part of the Hermitian matrix A (see <code>uplo</code>). The second dimension of a must be at least $\max(1, n)$. The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$. <code>work</code> is a workspace array, dimension at least $\max(1, lwork)$.
<code>lda</code>	INTEGER. The first dimension of a; $lda \geq \max(1, n)$.
<code>ldb</code>	INTEGER. The first dimension of b; $ldb \geq \max(1, n)$.
<code>lwork</code>	INTEGER. The size of the <code>work</code> array ($lwork \geq 1$). If <code>lwork</code> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <code>work</code> array, returns this value as the first entry of the <code>work</code> array, and no error message related to <code>lwork</code> is issued by <code>xerbla</code>. See <i>Application Notes</i> below for details and for the suggested value of <code>lwork</code>.

Output Parameters

<code>a</code>	If <code>info</code> = 0, a is overwritten by the block-diagonal matrix D and the multipliers used to obtain the factor U (or L) from the factorization of A as computed by <code>?hetrf</code>.
<code>b</code>	If <code>info</code> = 0, b is overwritten by the solution matrix X.

<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. Contains details of the interchanges and the block structure of D, as determined by ?hetrf. If $ipiv(i) = k > 0$, then d_{ii} is a 1-by-1 diagonal block, and the i-th row and column of A was interchanged with the k-th row and column. If $uplo = 'U'$ and $ipiv(i) = ipiv(i-1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i-1$, and $(i-1)$-th row and column of A was interchanged with the m-th row and column. If $uplo = 'L'$ and $ipiv(i) = ipiv(i+1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i+1$, and $(i+1)$-th row and column of A was interchanged with the m-th row and column.
<i>work(1)</i>	If $info = 0$, on exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i-th parameter had an illegal value. If $info = i$, d_{ii} is 0. The factorization has been completed, but D is exactly singular, so the solution could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hesv` interface are as follows:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>ipiv</i>	Holds the vector of length (n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For better performance, try using $lwork = n * blocksize$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hesvx

Uses the diagonal pivoting factorization to compute the solution to the complex system of linear equations with a Hermitian matrix A, and provides error bounds on the solution.

Syntax

FORTRAN 77:

```
call chesvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx,
             rcond, ferr, berr, work, lwork, rwork, info )

call zhesvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx,
             rcond, ferr, berr, work, lwork, rwork, info )
```

Fortran 95:

```
call hesvx( a, b, x [,uplo] [,af] [,ipiv] [,fact] [,ferr] [,berr] [,rcond]
             [,info] )
```

Description

This routine uses the diagonal pivoting factorization to compute the solution to a complex system of linear equations $A^*X = B$, where A is an n -by- n Hermitian matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?hesvx performs the following steps:

1. If $fact = 'N'$, the diagonal pivoting method is used to factor the matrix A . The form of the factorization is $A = U*D*U^H$ or $A = L*D*L^H$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.
2. If some $d_{i,i} = 0$, so that D is exactly singular, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $info = n+1$ is returned as a warning, but the routine still goes on to solve for x and compute error bounds as described below.
3. The system of equations is solved for x using the factored form of A .
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

<i>fact</i>	CHARACTER*1. Must be 'F' or 'N'. Specifies whether or not the factored form of the matrix A has been supplied on entry. If $fact = 'F'$: on entry, af and $ipiv$ contain the factored form of A. Arrays a, af, and $ipiv$ are not modified. If $fact = 'N'$, the matrix A is copied to af and factored.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored and how A is factored: If $uplo = 'U'$, the array a stores the upper triangular part of the Hermitian matrix A, and A is factored as $U*D*U^H$. If $uplo = 'L'$, the array a stores the lower triangular part of the Hermitian matrix A; A is factored as $L*D*L^H$.
<i>n</i>	INTEGER. The order of matrix A; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, the number of columns in B; $nrhs \geq 0$.
<i>a,af,b,work</i>	COMPLEX for <code>chesvx</code> DOUBLE COMPLEX for <code>zhesvx</code>. Arrays: $a(lda,*)$, $af(ldaf,*)$, $b(ldb,*)$, $work(*)$.

The array *a* contains the upper or the lower triangular part of the Hermitian matrix *A* (see *uplo*). The second dimension of *a* must be at least $\max(1, n)$.

The array *af* is an input argument if *fact* = 'F'. It contains the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* from the factorization $A = U^*D^*U^H$ or $A = L^*D^*L^H$ as computed by [?hetrf](#). The second dimension of *af* must be at least $\max(1, n)$.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least $\max(1, nrhs)$.

work(*) is a workspace array of dimension at least $\max(1, lwork)$.

lda INTEGER. The first dimension of *a*; $lda \geq \max(1, n)$.

ldaf INTEGER. The first dimension of *af*; $ldaf \geq \max(1, n)$.

ldb INTEGER. The first dimension of *b*; $ldb \geq \max(1, n)$.

ipiv INTEGER.

Array, DIMENSION at least $\max(1, n)$. The array *ipiv* is an input argument if *fact* = 'F'. It contains details of the interchanges and the block structure of *D*, as determined by [?hetrf](#).

If *ipiv*(*i*) = *k* > 0, then *d_{ii}* is a 1-by-1 diagonal block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column.

If *uplo* = 'U' and *ipiv*(*i*) = *ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*-1, and (*i*-1)-th row and column of *A* was interchanged with the *m*-th row and column.

If *uplo* = 'L' and *ipiv*(*i*) = *ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)-th row and column of *A* was interchanged with the *m*-th row and column.

ldx INTEGER. The leading dimension of the output array *x*; $ldx \geq \max(1, n)$.

lwork INTEGER. The size of the *work* array.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by *xerbla*. See *Application Notes* below for details and for the suggested value of *lwork*.

rwork REAL for *chesvx*

DOUBLE PRECISION for zhesvx.
 Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>x</i>	COMPLEX for chesvx DOUBLE COMPLEX for zhesvx. Array, DIMENSION (<i>ldx</i>, *). If <i>info</i> = 0 or <i>info</i> = <i>n</i>+1, the array <i>x</i> contains the solution matrix <i>x</i> to the system of equations. The second dimension of <i>x</i> must be at least $\max(1, nrhs)$.
<i>af</i> , <i>ipiv</i>	These arrays are output arguments if <i>fact</i> = 'N'. See the description of <i>af</i>, <i>ipiv</i> in <i>Input Arguments</i> section.
<i>rcond</i>	REAL for chesvx DOUBLE PRECISION for zhesvx. An estimate of the reciprocal condition number of the matrix <i>A</i>. If <i>rcond</i> is less than the machine precision (in particular, if <i>rcond</i> = 0), the matrix is singular to working precision. This condition is indicated by a return code of <i>info</i> > 0.
<i>ferr</i>	REAL for chesvx DOUBLE PRECISION for zhesvx. Array, DIMENSION at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector <i>x</i>(<i>j</i>) (the <i>j</i>-th column of the solution matrix <i>X</i>). If <i>xtrue</i> is the true solution corresponding to <i>x</i>(<i>j</i>), <i>ferr</i>(<i>j</i>) is an estimated upper bound for the magnitude of the largest element in (<i>x</i>(<i>j</i>) - <i>xtrue</i>) divided by the magnitude of the largest element in <i>x</i>(<i>j</i>). The estimate is as reliable as the estimate for <i>rcon</i>, and is almost always a slight overestimate of the true error.
<i>berr</i>	REAL for chesvx DOUBLE PRECISION for zhesvx. Array, DIMENSION at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector <i>x</i>(<i>j</i>), that is, the smallest relative change in any element of <i>A</i> or <i>B</i> that makes <i>x</i>(<i>j</i>) an exact solution.
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i>(1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, and $i \leq n$, then d_{ii} is exactly zero. The factorization has been completed, but the block diagonal matrix D is exactly singular, so the solution and error bounds could not be computed; $rcond = 0$ is returned.

If $info = i$, and $i = n + 1$, then D is nonsingular, but $rcond$ is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of $rcond$ would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hesvx` interface are as follows:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>x</i>	Holds the matrix X of size $(n, nrhs)$.
<i>af</i>	Holds the matrix AF of size (n, n) .
<i>ipiv</i>	Holds the vector of length (n) .
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>fact</i>	Must be 'N' or 'F'. The default value is 'N'. If <i>fact</i> = 'F', then both arguments <i>af</i> and <i>ipiv</i> must be present; otherwise, an error is returned.

Application Notes

The value of *lwork* must be at least $2*n$. For better performance, try using $lwork = n*blocksize$, where *blocksize* is the optimal block size for `?hetrf`.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?spsv

Computes the solution to the system of linear equations with a real or complex symmetric matrix A stored in packed format, and multiple right-hand sides.

Syntax

FORTRAN 77:

```
call sspsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call dpsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call cspsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call zspsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
```

Fortran 95:

```
call spsv( a, b [,uplo] [,ipiv] [,info] )
```

Description

This routine solves for *X* the real or complex system of linear equations $A^*X = B$, where *A* is an *n*-by-*n* symmetric matrix stored in packed format, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

The diagonal pivoting method is used to factor *A* as $A = U^*D^*U^T$ or $A = L^*D^*L^T$, where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, and *D* is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of A is then used to solve the system of equations $A^*X = B$.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.
<i>ap</i> , <i>b</i>	REAL for <i>sspsv</i> DOUBLE PRECISION for <i>dspsv</i> COMPLEX for <i>cspsv</i> DOUBLE COMPLEX for <i>zspsv</i> . Arrays: <i>ap</i> (*), <i>b</i> (<i>ldb</i> ,*). The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$. The array <i>ap</i> contains the factor U or L , as specified by <i>uplo</i> , in <i>packed storage</i> (see Matrix Storage Schemes). The array <i>b</i> contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; $ldb \geq \max(1, n)$.

Output Parameters

<i>ap</i>	The block-diagonal matrix D and the multipliers used to obtain the factor U (or L) from the factorization of A as computed by ?spturf , stored as a packed triangular matrix in the same storage format as A .
<i>b</i>	If <i>info</i> = 0, <i>b</i> is overwritten by the solution matrix X .
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. Contains details of the interchanges and the block structure of D , as determined by ?spturf . If <i>ipiv</i> (<i>i</i>) = <i>k</i> > 0, then d_{ii} is a 1-by-1 block, and the <i>i</i> -th row and column of A was interchanged with the <i>k</i> -th row and column.

If $uplo = 'U'$ and $ipiv(i) = ipiv(i-1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i-1$, and $(i-1)$ -th row and column of A was interchanged with the m -th row and column.

If $uplo = 'L'$ and $ipiv(i) = ipiv(i+1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i+1$, and $(i+1)$ -th row and column of A was interchanged with the m -th row and column.

info

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, d_{ii} is 0. The factorization has been completed, but D is exactly singular, so the solution could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `spsv` interface are as follows:

<i>a</i>	Stands for argument <i>ap</i> in Fortran 77 interface. Holds the array A of size $(n * (n+1) / 2)$.
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>ipiv</i>	Holds the vector of length (n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

?spsvx

Uses the diagonal pivoting factorization to compute the solution to the system of linear equations with a real or complex symmetric matrix A stored in packed format, and provides error bounds on the solution.

Syntax

FORTRAN 77:

```
call sspsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr,
berr, work, iwork, info )

call dspsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr,
berr, work, iwork, info )

call cspsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr,
berr, work, rwork, info )

call zspsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr,
berr, work, rwork, info )
```

Fortran 95:

```
call spsvx( a, b, x [,uplo] [,af] [,ipiv] [,fact] [,ferr] [,berr] [,rcond]
[,info] )
```

Description

This routine uses the diagonal pivoting factorization to compute the solution to a real or complex system of linear equations $A \cdot X = B$, where A is a n -by- n symmetric matrix stored in packed format, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?spsvx performs the following steps:

1. If `fact = 'N'`, the diagonal pivoting method is used to factor the matrix A . The form of the factorization is $A = U \cdot D \cdot U^T$ or $A = L \cdot D \cdot L^T$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

2. If some $d_{i,i} = 0$, so that D is exactly singular, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $info = n+1$ is returned as a warning, but the routine still goes on to solve for x and compute error bounds as described below.
3. The system of equations is solved for x using the factored form of A .
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

<i>fact</i>	CHARACTER*1. Must be 'F' or 'N'. Specifies whether or not the factored form of the matrix A has been supplied on entry. If <i>fact</i> = 'F': on entry, <i>afp</i> and <i>ipiv</i> contain the factored form of A. Arrays <i>ap</i>, <i>afp</i>, and <i>ipiv</i> are not modified. If <i>fact</i> = 'N', the matrix A is copied to <i>afp</i> and factored.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored and how A is factored: If <i>uplo</i> = 'U', the array <i>ap</i> stores the upper triangular part of the symmetric matrix A, and A is factored as $U^*D^*U^T$. If <i>uplo</i> = 'L', the array <i>ap</i> stores the lower triangular part of the symmetric matrix A; A is factored as $L^*D^*L^T$.
<i>n</i>	INTEGER. The order of matrix A; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, the number of columns in B; $nrhs \geq 0$.
<i>ap,afp,b,work</i>	REAL for sspsvx DOUBLE PRECISION for dpsvx COMPLEX for cspsvx DOUBLE COMPLEX for zspsvx. Arrays: <i>ap</i>(*), <i>afp</i>(*), <i>b</i>(ldb,*), <i>work</i>(*). The array <i>ap</i> contains the upper or lower triangle of the symmetric matrix A in <i>packed storage</i> (see Matrix Storage Schemes).

The array *afp* is an input argument if *fact* = 'F'. It contains the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* from the factorization $A = U * D * U^T$ or $A = L * D * L^T$ as computed by [?spturf](#), in the same storage format as *A*.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.

work(*) is a workspace array.

The dimension of arrays *ap* and *afp* must be at least $\max(1, n(n+1)/2)$; the second dimension of *b* must be at least $\max(1, nrhs)$; the dimension of *work* must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

ldb

INTEGER. The first dimension of *b*; $ldb \geq \max(1, n)$.

ipiv

INTEGER.

Array, DIMENSION at least $\max(1, n)$. The array *ipiv* is an input argument if *fact* = 'F'. It contains details of the interchanges and the block structure of *D*, as determined by [?spturf](#).

If *ipiv*(*i*) = *k* > 0, then *d_{ii}* is a 1-by-1 diagonal block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column.

If *uplo* = 'U' and *ipiv*(*i*) = *ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*-1, and (*i*-1)-th row and column of *A* was interchanged with the *m*-th row and column.

If *uplo* = 'L' and *ipiv*(*i*) = *ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)-th row and column of *A* was interchanged with the *m*-th row and column.

ldx

INTEGER. The leading dimension of the output array *x*; $ldx \geq \max(1, n)$.

iwork

INTEGER. Workspace array, DIMENSION at least $\max(1, n)$; used in real flavors only.

rwork

REAL for *cspsvx*

DOUBLE PRECISION for *zspsvx*.

Workspace array, DIMENSION at least $\max(1, n)$; used in complex flavors only.

Output Parameters

x

REAL for *sspsvx*

DOUBLE PRECISION for *dspsvx*

	COMPLEX for <code>cspsvx</code> DOUBLE COMPLEX for <code>zspsvx</code> . Array, DIMENSION (<code>ldx</code> , *). If <code>info = 0</code> or <code>info = n+1</code> , the array <code>x</code> contains the solution matrix <code>X</code> to the system of equations. The second dimension of <code>x</code> must be at least <code>max(1, nrhs)</code> .
<code>afp, ipiv</code>	These arrays are output arguments if <code>fact = 'N'</code> . See the description of <code>afp, ipiv</code> in <i>Input Arguments</i> section.
<code>rcond</code>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal condition number of the matrix <code>A</code> . If <code>rcond</code> is less than the machine precision (in particular, if <code>rcond = 0</code>), the matrix is singular to working precision. This condition is indicated by a return code of <code>info > 0</code> .
<code>ferr, berr</code>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least <code>max(1, nrhs)</code> . Contain the component-wise forward and relative backward errors, respectively, for each solution vector.
<code>info</code>	INTEGER. If <code>info = 0</code> , the execution is successful. If <code>info = -i</code> , the <code>i</code> -th parameter had an illegal value. If <code>info = i</code> , and <code>i ≤ n</code> , then <code>d_{ii}</code> is exactly zero. The factorization has been completed, but the block diagonal matrix <code>D</code> is exactly singular, so the solution and error bounds could not be computed; <code>rcond = 0</code> is returned. If <code>info = i</code> , and <code>i = n + 1</code> , then <code>D</code> is nonsingular, but <code>rcond</code> is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of <code>rcond</code> would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `spsvx` interface are as follows:

<i>a</i>	Stands for argument <i>ap</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(n * (n+1) / 2)$.
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.
<i>af</i>	Stands for argument <i>afp</i> in Fortran 77 interface. Holds the array <i>AF</i> of size $(n * (n+1) / 2)$.
<i>ipiv</i>	Holds the vector of length (n) .
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>fact</i>	Must be 'N' or 'F'. The default value is 'N'. If <i>fact</i> = 'F', then both arguments <i>af</i> and <i>ipiv</i> must be present; otherwise, an error is returned.

?hpsv

Computes the solution to the system of linear equations with a Hermitian matrix A stored in packed format, and multiple right-hand sides.

Syntax

FORTRAN 77:

```
call chpsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call zhpsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
```

Fortran 95:

```
call hpsv( a, b [,uplo] [,ipiv] [,info] )
```

Description

This routine solves for *x* the system of linear equations $A * X = B$, where *A* is an *n*-by-*n* Hermitian matrix stored in packed format, the columns of matrix *B* are individual right-hand sides, and the columns of *x* are the corresponding solutions.

The diagonal pivoting method is used to factor *A* as $A = U * D * U^H$ or $A = L * D * L^H$, where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, and *D* is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of A is then used to solve the system of equations $A^*X = B$.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	INTEGER. The order of matrix A; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns in B; $nrhs \geq 0$.
<i>ap, b</i>	COMPLEX for <code>chpsv</code> DOUBLE COMPLEX for <code>zhpsv</code>. Arrays: <i>ap</i>(*), <i>b</i>(<i>ldb</i>,*). The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$. The array <i>ap</i> contains the factor U or L, as specified by <i>uplo</i>, in <i>packed storage</i> (see Matrix Storage Schemes). The array <i>b</i> contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i>; $ldb \geq \max(1, n)$.

Output Parameters

<i>ap</i>	The block-diagonal matrix D and the multipliers used to obtain the factor U (or L) from the factorization of A as computed by ?hptrf, stored as a packed triangular matrix in the same storage format as A.
<i>b</i>	If <i>info</i> = 0, <i>b</i> is overwritten by the solution matrix X.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. Contains details of the interchanges and the block structure of D, as determined by ?hptrf. If <i>ipiv</i>(<i>i</i>) = <i>k</i> > 0, then d_{ii} is a 1-by-1 block, and the <i>i</i>-th row and column of A was interchanged with the <i>k</i>-th row and column. If <i>uplo</i> = 'U' and <i>ipiv</i>(<i>i</i>) = <i>ipiv</i>(<i>i</i>-1) = -<i>m</i> < 0, then D has a 2-by-2 block in rows/columns <i>i</i> and <i>i</i>-1, and (<i>i</i>-1)-th row and column of A was interchanged with the <i>m</i>-th row and column.

If $uplo = 'L'$ and $ipiv(i) = ipiv(i+1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i+1$, and $(i+1)$ -th row and column of A was interchanged with the m -th row and column.

info

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, d_{ii} is 0. The factorization has been completed, but D is exactly singular, so the solution could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hpsv` interface are as follows:

<i>a</i>	Stands for argument <i>ap</i> in Fortran 77 interface. Holds the array A of size $(n*(n+1)/2)$.
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>ipiv</i>	Holds the vector of length (n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

?hpsvx

Uses the diagonal pivoting factorization to compute the solution to the system of linear equations with a Hermitian matrix A stored in packed format, and provides error bounds on the solution.

Syntax

FORTRAN 77:

```
call chpsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr,
berr, work, rwork, info )
```

```
call zhpsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr,
berr, work, rwork, info )
```

Fortran 95:

```
call hpsvx( a, b, x [,uplo] [,af] [,ipiv] [,fact] [,ferr] [,berr] [,rcond]
[,info] )
```

Description

This routine uses the diagonal pivoting factorization to compute the solution to a complex system of linear equations $A^*X = B$, where A is a n -by- n Hermitian matrix stored in packed format, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?hpsvx` performs the following steps:

1. If `fact = 'N'`, the diagonal pivoting method is used to factor the matrix A . The form of the factorization is $A = U^*D^*U^H$ or $A = L^*D^*L^H$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is a Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.
2. If some $d_{i,i} = 0$, so that D is exactly singular, then the routine returns with `info = i`. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, `info = n+1` is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
3. The system of equations is solved for X using the factored form of A .
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

<i>fact</i>	CHARACTER*1. Must be 'F' or 'N'. Specifies whether or not the factored form of the matrix A has been supplied on entry. If <code>fact = 'F'</code> : on entry, <code>afp</code> and <code>ipiv</code> contain the factored form of A . Arrays <code>ap</code> , <code>afp</code> , and <code>ipiv</code> are not modified. If <code>fact = 'N'</code> , the matrix A is copied to <code>afp</code> and factored.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If `uplo = 'U'`, the array `ap` stores the upper triangular part of the Hermitian matrix A , and A is factored as $U^*D^*U^H$.

If `uplo = 'L'`, the array `ap` stores the lower triangular part of the Hermitian matrix A , and A is factored as $L^*D^*L^H$.

`n` INTEGER. The order of matrix A ; $n \geq 0$.

`nrhs` INTEGER. The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.

`ap,afp,b,work` COMPLEX for `chpsvx`
DOUBLE COMPLEX for `zhpsvx`.
Arrays: `ap(*)`, `afp(*)`, `b(ldb,*)`, `work(*)`.
The array `ap` contains the upper or lower triangle of the Hermitian matrix A in *packed storage* (see [Matrix Storage Schemes](#)).
The array `afp` is an input argument if `fact = 'F'`. It contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization $A = U^*D^*U^H$ or $A = L^*D^*L^H$ as computed by [?hptrf](#), in the same storage format as A .
The array `b` contains the matrix B whose columns are the right-hand sides for the systems of equations.
`work(*)` is a workspace array.
The dimension of arrays `ap` and `afp` must be at least $\max(1, n(n+1)/2)$; the second dimension of `b` must be at least $\max(1, nrhs)$; the dimension of `work` must be at least $\max(1, 2*n)$.

`ldb` INTEGER. The first dimension of `b`; $ldb \geq \max(1, n)$.

`ipiv` INTEGER.
Array, DIMENSION at least $\max(1, n)$. The array `ipiv` is an input argument if `fact = 'F'`. It contains details of the interchanges and the block structure of D , as determined by [?hptrf](#).
If `ipiv(i) = k > 0`, then d_{ii} is a 1-by-1 diagonal block, and the i -th row and column of A was interchanged with the k -th row and column.
If `uplo = 'U'` and `ipiv(i) = ipiv(i-1) = -m < 0`, then D has a 2-by-2 block in rows/columns i and $i-1$, and $(i-1)$ -th row and column of A was interchanged with the m -th row and column.
If `uplo = 'L'` and `ipiv(i) = ipiv(i+1) = -m < 0`, then D has a 2-by-2 block in rows/columns i and $i+1$, and $(i+1)$ -th row and column of A was interchanged with the m -th row and column.

ldx INTEGER. The leading dimension of the output array *x*; $ldx \geq \max(1, n)$.

rwork REAL for `chpsvx`
DOUBLE PRECISION for `zhpsvx`.
Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

x COMPLEX for `chpsvx`
DOUBLE COMPLEX for `zhpsvx`.
Array, DIMENSION (*ldx*, *).
If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *x* to the system of equations. The second dimension of *x* must be at least $\max(1, nrhs)$.

afp, ipiv These arrays are output arguments if *fact* = 'N'. See the description of *afp, ipiv* in *Input Arguments* section.

rcond REAL for `chpsvx`
DOUBLE PRECISION for `zhpsvx`.
An estimate of the reciprocal condition number of the matrix *A*. If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.

ferr REAL for `chpsvx`
DOUBLE PRECISION for `zhpsvx`.
Array, DIMENSION at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector *x*(*j*) (the *j*-th column of the solution matrix *x*). If *xtrue* is the true solution corresponding to *x*(*j*), *ferr*(*j*) is an estimated upper bound for the magnitude of the largest element in (*x*(*j*) - *xtrue*) divided by the magnitude of the largest element in *x*(*j*). The estimate is as reliable as the estimate for *rcond*, and is almost always a slight overestimate of the true error.

berr REAL for `chpsvx`
DOUBLE PRECISION for `zhpsvx`.
Array, DIMENSION at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector *x*(*j*), that is, the smallest relative change in any element of *A* or *B* that makes *x*(*j*) an exact solution.

info INTEGER. If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.
 If *info* = *i*, and $i \leq n$, then d_{ii} is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.
 If *info* = *i*, and $i = n + 1$, then *D* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hpsvx` interface are as follows:

<i>a</i>	Stands for argument <i>ap</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(n^*(n+1)/2)$.
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.
<i>af</i>	Stands for argument <i>ap</i> in Fortran 77 interface. Holds the array <i>AF</i> of size $(n^*(n+1)/2)$.
<i>ipiv</i>	Holds the vector of length (n) .
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>fact</i>	Must be 'N' or 'F'. The default value is 'N'. If <i>fact</i> = 'F', then both arguments <i>af</i> and <i>ipiv</i> must be present; otherwise, an error is returned.

LAPACK Routines: Least Squares and Eigenvalue Problems

4

This chapter describes the Intel® Math Kernel Library implementation of routines from the LAPACK package that are used for solving linear least squares problems, eigenvalue and singular value problems, as well as performing a number of related computational tasks.

Sections in this chapter include descriptions of LAPACK [computational routines](#) and [driver routines](#). For full reference on LAPACK routines and related information see [\[LUG\]](#).

Least Squares Problems. A typical *least squares problem* is as follows: given a matrix A and a vector b , find the vector x that minimizes the sum of squares $\sum_i (Ax)_i - b_i)^2$ or, equivalently, find the vector x that minimizes the 2-norm $\|Ax - b\|_2$.

In the most usual case, A is an m -by- n matrix with $m \geq n$ and $\text{rank}(A) = n$. This problem is also referred to as finding the *least squares solution* to an *overdetermined* system of linear equations (here we have more equations than unknowns). To solve this problem, you can use the QR factorization of the matrix A (see [QR Factorization](#)).

If $m < n$ and $\text{rank}(A) = m$, there exist an infinite number of solutions x which exactly satisfy $Ax = b$, and thus minimize the norm $\|Ax - b\|_2$. In this case it is often useful to find the unique solution that minimizes $\|x\|_2$. This problem is referred to as finding the *minimum-norm solution* to an *underdetermined* system of linear equations (here we have more unknowns than equations). To solve this problem, you can use the LQ factorization of the matrix A (see [LQ Factorization](#)).

In the general case you may have a *rank-deficient least squares problem*, with $\text{rank}(A) < \min(m, n)$: find the *minimum-norm least squares solution* that minimizes both $\|x\|_2$ and $\|Ax - b\|_2^2$. In this case (or when the rank of A is in doubt) you can use the QR factorization with pivoting or *singular value decomposition* (see [Singular Value Decomposition](#)).

Eigenvalue Problems. The eigenvalue problems (from German *eigen* "own") are stated as follows: given a matrix A , find the *eigenvalues* λ and the corresponding *eigenvectors* z that satisfy the equation

$$Az = \lambda z \text{ (right eigenvectors } z)$$

or the equation

$$z^H A = \lambda z^H \text{ (left eigenvectors } z).$$

If A is a real symmetric or complex Hermitian matrix, the above two equations are equivalent, and the problem is called a *symmetric eigenvalue problem*. Routines for solving this type of problems are described in the section [Symmetric Eigenvalue Problems](#).

Routines for solving eigenvalue problems with nonsymmetric or non-Hermitian matrices are described in the section [Nonsymmetric Eigenvalue Problems](#).

The library also includes routines that handle *generalized symmetric-definite eigenvalue problems*: find the eigenvalues λ and the corresponding eigenvectors x that satisfy one of the following equations:

$$Az = \lambda Bz, ABz = \lambda z, \text{ or } BAz = \lambda z,$$

where A is symmetric or Hermitian, and B is symmetric positive-definite or Hermitian positive-definite. Routines for reducing these problems to standard symmetric eigenvalue problems are described in the section [Generalized Symmetric-Definite Eigenvalue Problems](#).

To solve a particular problem, you usually call several computational routines. Sometimes you need to combine the routines of this chapter with other LAPACK routines described in Chapter 3 as well as with BLAS routines described in Chapter 2.

For example, to solve a set of least squares problems minimizing $\|Ax - b\|^2$ for all columns b of a given matrix B (where A and B are real matrices), you can call `?geqrf` to form the factorization $A = QR$, then call `?ormqr` to compute $C = Q^H B$ and finally call the BLAS routine `?trsm` to solve for X the system of equations $RX = C$.

Another way is to call an appropriate driver routine that performs several tasks in one call. For example, to solve the least squares problem the driver routine `?gels` can be used.



WARNING. LAPACK routines require that input matrices do not contain `INF` or `NaN` values. When input data is inappropriate for LAPACK, problems may arise, including possible hangs.

Starting from release 8.0, Intel MKL along with Fortran 77 interface to LAPACK computational and driver routines supports also Fortran 95 interface which uses simplified routine calls with shorter argument lists. The calling sequence for Fortran 95 interface is given in the syntax section of the routine description immediately after Fortran 77 calls.

Routine Naming Conventions

For each routine in this chapter, when calling it from the Fortran 77 program you can use the LAPACK name.

LAPACK names have the structure `xyyzzz`, which is described below.

The initial letter `x` indicates the data type:

s	real, single precision
c	complex, single precision
d	real, double precision
z	complex, double precision

The second and third letters *yy* indicate the matrix type and storage scheme:

bd	bidiagonal matrix
ge	general matrix
gb	general band matrix
hs	upper Hessenberg matrix
or	(real) orthogonal matrix
op	(real) orthogonal matrix (packed storage)
un	(complex) unitary matrix
up	(complex) unitary matrix (packed storage)
pt	symmetric or Hermitian positive-definite tridiagonal matrix
sy	symmetric matrix
sp	symmetric matrix (packed storage)
sb	(real) symmetric band matrix
st	(real) symmetric tridiagonal matrix
he	Hermitian matrix
hp	Hermitian matrix (packed storage)
hb	(complex) Hermitian band matrix
tr	triangular or quasi-triangular matrix.

The last three letters *zzz* indicate the computation performed, for example:

qrf	form the QR factorization
lqf	form the LQ factorization.

Thus, the routine `sgeqrf` forms the QR factorization of general real matrices in single precision; the corresponding routine for complex matrices is `cgeqrf`.

Names of the LAPACK computational and driver routines for Fortran 95 interface in Intel MKL are the same as Fortran 77 names but without the first letter that indicates the data type. For example, the name of the routine that forms the QR factorization of general real matrices in Fortran 95 interface is `geqrf`. Handling of different data types is done through defining a specific internal parameter referring to a module block with named constants for single and double precision.

For details on the design of Fortran 95 interface for LAPACK computational and driver routines in Intel MKL and for the general information on how the optional arguments are reconstructed, see [Fortran 95 Interface Conventions](#) in chapter 3 .

Matrix Storage Schemes

LAPACK routines use the following matrix storage schemes:

- *Full storage*: a matrix A is stored in a two-dimensional array a , with the matrix element a_{ij} stored in the array element $a(i, j)$.
- *Packed storage* scheme allows you to store symmetric, Hermitian, or triangular matrices more compactly: the upper or lower triangle of the matrix is packed by columns in a one-dimensional array.
- *Band storage*: an m -by- n band matrix with kl sub-diagonals and ku super-diagonals is stored compactly in a two-dimensional array ab with $kl+ku+1$ rows and n columns. Columns of the matrix are stored in the corresponding columns of the array, and *diagonals* of the matrix are stored in rows of the array.

In Chapters 3 and 4, arrays that hold matrices in packed storage have names ending in p ; arrays with matrices in band storage have names ending in b . For more information on matrix storage schemes, see “[Matrix Arguments](#)” in Appendix B.

Mathematical Notation

In addition to the mathematical notation used in previous chapters, descriptions of routines in this chapter use the following notation:

λ_i	<i>Eigenvalues</i> of the matrix A (for the definition of eigenvalues, see Eigenvalue Problems).
σ_i	<i>Singular values</i> of the matrix A . They are equal to square roots of the eigenvalues of $A^H A$. (For more information, see Singular Value Decomposition).
$\ x\ _2$	The <i>2-norm</i> of the vector x : $\ x\ _2 = (\sum_i x_i ^2)^{1/2} = \ x\ _E$.
$\ A\ _2$	The <i>2-norm</i> (or <i>spectral norm</i>) of the matrix A . $\ A\ _2 = \max_i \sigma_i$, $\ A\ _2^2 = \max_{\ x\ =1} (Ax \cdot Ax)$.
$\ A\ _E$	The <i>Euclidean norm</i> of the matrix A : $\ A\ _E^2 = \sum_i \sum_j a_{ij} ^2$ (for vectors, the Euclidean norm and the 2-norm are equal: $\ x\ _E = \ x\ _2$).
$q(x, y)$	The <i>acute angle between vectors</i> x and y : $\cos q(x, y) = x \cdot y / (\ x\ _2 \ y\ _2)$.

Computational Routines

In the sections that follow, the descriptions of LAPACK computational routines are given. These routines perform distinct computational tasks that can be used for:

Orthogonal Factorizations

Singular Value Decomposition

Symmetric Eigenvalue Problems

Generalized Symmetric-Definite Eigenvalue Problems

Nonsymmetric Eigenvalue Problems

Generalized Nonsymmetric Eigenvalue Problems

Generalized Singular Value Decomposition

See also the respective [driver routines](#).

Orthogonal Factorizations

This section describes the LAPACK routines for the QR (RQ) and LQ (QL) factorization of matrices. Routines for the RZ factorization as well as for generalized QR and RQ factorizations are also included.

QR Factorization. Assume that A is an m -by- n matrix to be factored.

If $m \geq n$, the QR factorization is given by

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix} = (Q_1, Q_2) \begin{pmatrix} R \\ 0 \end{pmatrix}$$

where R is an n -by- n upper triangular matrix with real diagonal elements, and Q is an m -by- m orthogonal (or unitary) matrix.

You can use the QR factorization for solving the following least squares problem: minimize $\|Ax - b\|^2$ where A is a full-rank m -by- n matrix ($m \geq n$). After factoring the matrix, compute the solution x by solving $Rx = (Q_1)^T b$.

If $m < n$, the QR factorization is given by

$$A = QR = Q(R_1 R_2)$$

where R is trapezoidal, R_1 is upper triangular and R_2 is rectangular.

The LAPACK routines do not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ *elementary reflectors*. Routines are provided to work with Q in this representation.

LQ Factorization LQ factorization of an m -by- n matrix A is as follows. If $m \leq n$,

$$A = (L, 0) Q = (L, 0) \begin{pmatrix} Q_1 \\ Q_2 \end{pmatrix} = (LQ1)$$

where L is an m -by- m lower triangular matrix with real diagonal elements, and Q is an n -by- n orthogonal (or unitary) matrix.

If $m > n$, the LQ factorization is

$$A = \begin{pmatrix} L_1 \\ L_2 \end{pmatrix} Q$$

where L_1 is an n -by- n lower triangular matrix, L_2 is rectangular, and Q is an n -by- n orthogonal (or unitary) matrix.

You can use the LQ factorization to find the minimum-norm solution of an underdetermined system of linear equations $Ax = b$ where A is an m -by- n matrix of rank m ($m < n$). After factoring the matrix, compute the solution vector x as follows: solve $L_1 y = b$ for y , and then compute $x = (Q_1)^H y$.

[Table 4-1](#) lists LAPACK routines (Fortran 77 interface) that perform orthogonal factorization of matrices. Respective routine names in Fortran 95 interface are without the first symbol (see [Routine Naming Conventions](#)).

Table 4-1 Computational Routines for Orthogonal Factorization

Matrix type, factorization	Factorize without pivoting	Factorize with pivoting	Generate matrix Q	Apply matrix Q
general matrices, QR factorization	?geqrf	?geqpf ?geqp3	?orgqr ?ungqr	?ormqr ?unmqr
general matrices, RQ factorization	?gerqf		?orgrq ?ungrq	?ormrq ?unmrq

Matrix type, factorization	Factorize without pivoting	Factorize with pivoting	Generate matrix Q	Apply matrix Q
general matrices, LQ factorization	?gelqf		?orglq ?unglq	?ormlq ?unmlq
general matrices, QL factorization	?geqlf		?orgql ?ungql	?ormql ?unmql
trapezoidal matrices, RZ factorization	?tzzrf			?ormrz ?unmrz
pair of matrices, generalized QR factorization	?ggqrf			
pair of matrices, generalized RQ factorization	?ggrqf			

?geqrf

Computes the QR factorization of a general *m*-by-*n* matrix.

Syntax

FORTRAN 77:

```
call sgeqrf(m, n, a, lda, tau, work, lwork, info)
call dgeqrf(m, n, a, lda, tau, work, lwork, info)
call cgeqrf(m, n, a, lda, tau, work, lwork, info)
call zgeqrf(m, n, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call geqrf(a [, tau] [,info])
```

Description

The routine forms the *QR* factorization of a general *m*-by-*n* matrix *A* (see [Orthogonal Factorizations](#)). No pivoting is performed.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of $\min(m, n)$ *elementary reflectors*. Routines are provided to work with *Q* in this representation.

Input Parameters

<i>m</i>	INTEGER. The number of rows in the matrix <i>A</i> ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in <i>A</i> ($n \geq 0$).
<i>a</i> , <i>work</i>	REAL for <code>sgeqrf</code> DOUBLE PRECISION for <code>dgeqrf</code> COMPLEX for <code>cgeqrf</code> DOUBLE COMPLEX for <code>zgeqrf</code> . Arrays: <i>a</i> (<i>lda</i> ,*) contains the matrix <i>A</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; at least $\max(1, m)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array ($lwork \geq n$). If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <code>xerbla</code> . See Application Notes for the suggested value of <i>lwork</i> .

Output Parameters

<i>a</i>	Overwritten by the factorization data as follows: If $m \geq n$, the elements below the diagonal are overwritten by the details of the unitary matrix <i>Q</i> , and the upper triangle is overwritten by the corresponding elements of the upper triangular matrix <i>R</i> . If $m < n$, the strictly lower triangular part is overwritten by the details of the unitary matrix <i>Q</i> , and the remaining elements are overwritten by the corresponding elements of the <i>m</i> -by- <i>n</i> upper trapezoidal matrix <i>R</i> .
<i>tau</i>	REAL for <code>sgeqrf</code> DOUBLE PRECISION for <code>dgeqrf</code> COMPLEX for <code>cgeqrf</code> DOUBLE COMPLEX for <code>zgeqrf</code> . Array, DIMENSION at least $\max(1, \min(m, n))$. Contains additional information on the matrix <i>Q</i> .

work(1) If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *geqrf* interface are the following:

a Holds the matrix *A* of size (*m*,*n*).

tau Holds the vector of length min(*m*,*n*)

Application Notes

For better performance, try using *lwork* = *n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed factorization is the exact factorization of a matrix $A + E$, where

$$\|E\|_2 = O(\epsilon) \|A\|_2.$$

The approximate number of floating-point operations for real flavors is

$$\begin{aligned} (4/3)n^3 & \quad \text{if } m = n, \\ (2/3)n^2(3m-n) & \quad \text{if } m > n, \\ (2/3)m^2(3n-m) & \quad \text{if } m < n. \end{aligned}$$

The number of operations for complex flavors is 4 times greater.

To solve a set of least squares problems minimizing $\|A^*x - b\|_2$ for all columns b of a given matrix B , you can call the following:

`?geqrf` (this routine) to factorize $A = QR$;
`?ormqr` to compute $C = Q^T * B$ (for real matrices);
`?unmqr` to compute $C = Q^H * B$ (for complex matrices);
`?trsm` (a BLAS routine) to solve $R^*X = C$.

(The columns of the computed x are the least squares solution vectors x .)

To compute the elements of Q explicitly, call

`?orgqr` (for real matrices)
`?ungqr` (for complex matrices).

?geqpf

Computes the QR factorization of a general m -by- n matrix with pivoting.

Syntax

FORTRAN 77:

```
call sgeqpf(m, n, a, lda, jpvt, tau, work, info)
call dgeqpf(m, n, a, lda, jpvt, tau, work, info)
call cgeqpf(m, n, a, lda, jpvt, tau, work, rwork, info)
call zgeqpf(m, n, a, lda, jpvt, tau, work, rwork, info)
```

Fortran 95:

```
call geqpf(a, jpvt [,tau] [,info])
```

Description

This routine is deprecated and has been replaced by routine `?geqp3`.

The routine `?geqpf` forms the QR factorization of a general m -by- n matrix A with column pivoting: $A^*P = Q^*R$ (see [Orthogonal Factorizations](#)). Here P denotes an n -by- n permutation matrix.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ *elementary reflectors*. Routines are provided to work with Q in this representation.

Input Parameters

m	INTEGER. The number of rows in the matrix A ($m \geq 0$).
n	INTEGER. The number of columns in A ($n \geq 0$).
$a, work$	REAL for <code>sgeqpf</code> DOUBLE PRECISION for <code>dgeqpf</code> COMPLEX for <code>cgeqpf</code> DOUBLE COMPLEX for <code>zgeqpf</code> . Arrays: $a(la,*)$ contains the matrix A . The second dimension of a must be at least $\max(1, n)$. $work(lwork)$ is a workspace array. The size of the $work$ array must be at least $\max(1, 3*n)$ for real flavors and at least $\max(1, n)$ for complex flavors.
lda	INTEGER. The first dimension of a ; at least $\max(1, m)$.
$jpvt$	INTEGER. Array, DIMENSION at least $\max(1, n)$. On entry, if $jpvt(i) > 0$, the i -th column of A is moved to the beginning of A^*P before the computation, and fixed in place during the computation. If $jpvt(i) = 0$, the i th column of A is a free column (that is, it may be interchanged during the computation with any other free column).
$rwork$	REAL for <code>cgeqpf</code> DOUBLE PRECISION for <code>zgeqpf</code> . A workspace array, DIMENSION at least $\max(1, 2*n)$.

Output Parameters

a	Overwritten by the factorization data as follows:

If $m \geq n$, the elements below the diagonal are overwritten by the details of the unitary (orthogonal) matrix Q , and the upper triangle is overwritten by the corresponding elements of the upper triangular matrix R .

If $m < n$, the strictly lower triangular part is overwritten by the details of the matrix Q , and the remaining elements are overwritten by the corresponding elements of the m -by- n upper trapezoidal matrix R .

tau REAL for sgeqpf
 DOUBLE PRECISION for dgeqpf
 COMPLEX for cgeqpf
 DOUBLE COMPLEX for zgeqpf.

Array, DIMENSION at least $\max(1, \min(m, n))$. Contains additional information on the matrix Q .

jpvt Overwritten by details of the permutation matrix P in the factorization $A*P = Q*R$. More precisely, the columns of $A*P$ are the columns of A in the following order:
 $jpvt(1), jpvt(2), \dots, jpvt(n)$.

info INTEGER.
 If $info = 0$, the execution is successful.
 If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `geqpf` interface are the following:

<i>a</i>	Holds the matrix A of size (m, n) .
<i>jpvt</i>	Holds the vector of length (n) .
<i>tau</i>	Holds the vector of length $\min(m, n)$

Application Notes

The computed factorization is the exact factorization of a matrix $A + E$, where

$$\|E\|_2 = O(\epsilon) \|A\|_2.$$

The approximate number of floating-point operations for real flavors is

$$\begin{aligned} (4/3)n^3 & \quad \text{if } m = n, \\ (2/3)n^2(3m-n) & \quad \text{if } m > n, \\ (2/3)m^2(3n-m) & \quad \text{if } m < n. \end{aligned}$$

The number of operations for complex flavors is 4 times greater.

To solve a set of least squares problems minimizing $\|A^*x - b\|_2$ for all columns b of a given matrix B , you can call the following:

`?geqpf` (this routine) to factorize $A^*P = Q^*R$;
`?ormqr` to compute $C = Q^T * B$ (for real matrices);
`?unmqr` to compute $C = Q^H * B$ (for complex matrices);
`?trsm` (a BLAS routine) to solve $R^*X = C$.

(The columns of the computed x are the permuted least squares solution vectors x ; the output array `jpvt` specifies the permutation order.)

To compute the elements of Q explicitly, call

`?orgqr` (for real matrices)
`?ungqr` (for complex matrices).

?geqp3

Computes the QR factorization of a general m -by- n matrix with column pivoting using level 3 BLAS.

Syntax

FORTRAN 77:

```
call sgeqp3(m, n, a, lda, jpvt, tau, work, lwork, info)
call dgeqp3(m, n, a, lda, jpvt, tau, work, lwork, info)
call cgeqp3(m, n, a, lda, jpvt, tau, work, lwork, rwork, info)
call zgeqp3(m, n, a, lda, jpvt, tau, work, lwork, rwork, info)
```

Fortran 95:

```
call geqp3(a, jpvt [,tau] [,info])
```

Description

The routine forms the QR factorization of a general m -by- n matrix A with column pivoting: $A^*P = Q^*R$ (see [Orthogonal Factorizations](#)) using Level 3 BLAS. Here P denotes an n -by- n permutation matrix. Use this routine instead of `?geqp3` for better performance.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ *elementary reflectors*. Routines are provided to work with Q in this representation.

Input Parameters

m	INTEGER. The number of rows in the matrix A ($m \geq 0$).
n	INTEGER. The number of columns in A ($n \geq 0$).
$a, work$	REAL for <code>sgeqp3</code> DOUBLE PRECISION for <code>dgeqp3</code> COMPLEX for <code>cgeqp3</code> DOUBLE COMPLEX for <code>zgeqp3</code> . Arrays: $a(lda,*)$ contains the matrix A . The second dimension of a must be at least $\max(1, n)$. $work$ is a workspace array, its dimension $\max(1, lwork)$.
lda	INTEGER. The first dimension of a ; at least $\max(1, m)$.
$lwork$	INTEGER. The size of the $work$ array; must be at least $\max(1, 3*n+1)$ for real flavors, and at least $\max(1, n+1)$ for complex flavors. If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by <code>xerbla</code> . See Application Notes below for details.
$jpvt$	INTEGER. Array, DIMENSION at least $\max(1, n)$. On entry, if $jpvt(i) \neq 0$, the i -th column of A is moved to the beginning of AP before the computation, and fixed in place during the computation.

If $jpvt(i) = 0$, the i -th column of A is a free column (that is, it may be interchanged during the computation with any other free column).

rwork

REAL for cgeqp3

DOUBLE PRECISION for zgeqp3.

A workspace array, DIMENSION at least $\max(1, 2*n)$. Used in complex flavors only.

Output Parameters

a

Overwritten by the factorization data as follows:

If $m \geq n$, the elements below the diagonal are overwritten by the details of the unitary (orthogonal) matrix Q , and the upper triangle is overwritten by the corresponding elements of the upper triangular matrix R .

If $m < n$, the strictly lower triangular part is overwritten by the details of the matrix Q , and the remaining elements are overwritten by the corresponding elements of the m -by- n upper trapezoidal matrix R .

tau

REAL for sgeqp3

DOUBLE PRECISION for dgeqp3

COMPLEX for cgeqp3

DOUBLE COMPLEX for zgeqp3.

Array, DIMENSION at least $\max(1, \min(m, n))$. Contains scalar factors of the elementary reflectors for the matrix Q .

jpvt

Overwritten by details of the permutation matrix P in the factorization $A*P = Q*R$. More precisely, the columns of AP are the columns of A in the following order:

$jpvt(1), jpvt(2), \dots, jpvt(n)$.

info

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `geqp3` interface are the following:

<code>a</code>	Holds the matrix A of size (m, n) .
<code>jpvt</code>	Holds the vector of length (n) .
<code>tau</code>	Holds the vector of length $\min(m, n)$

Application Notes

To solve a set of least squares problems minimizing $\|A^*x - b\|_2$ for all columns b of a given matrix B , you can call the following:

<code>?geqp3</code> (this routine)	to factorize $A^*P = Q^*R$;
<code>?ormqr</code>	to compute $C = Q^T * B$ (for real matrices);
<code>?unmqr</code>	to compute $C = Q^H * B$ (for complex matrices);
<code>?trsm</code> (a BLAS routine)	to solve $R^*X = C$.

(The columns of the computed x are the permuted least squares solution vectors x ; the output array `jpvt` specifies the permutation order.)

To compute the elements of Q explicitly, call

<code>?orgqr</code>	(for real matrices)
<code>?ungqr</code>	(for complex matrices).

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible `lwork` sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?orgqr

Generates the real orthogonal matrix Q of the QR factorization formed by ?geqrf.

Syntax

FORTRAN 77:

```
call sorgqr(m, n, k, a, lda, tau, work, lwork, info)
call dorgqr(m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call orgqr(a, tau [,info])
```

Description

The routine generates the whole or part of m -by- m orthogonal matrix Q of the QR factorization formed by the routines [sgeqrf/dgeqrf](#) or [sgeqpf/dgeqpf](#). Use this routine after a call to [sgeqrf/dgeqrf](#) or [sgeqpf/dgeqpf](#).

Usually Q is determined from the QR factorization of an m by p matrix A with $m \geq p$. To compute the whole matrix Q , use:

```
call ?orgqr(m, m, p, a, lda, tau, work, lwork, info)
```

To compute the leading p columns of Q (which form an orthonormal basis in the space spanned by the columns of A):

```
call ?orgqr(m, p, p, a, lda, tau, work, lwork, info)
```

To compute the matrix Q^k of the QR factorization of leading k columns of the matrix A :

```
call ?orgqr(m, m, k, a, lda, tau, work, lwork, info)
```

To compute the leading k columns of Q^k (which form an orthonormal basis in the space spanned by leading k columns of the matrix A):

```
call ?orgqr(m, k, k, a, lda, tau, work, lwork, info)
```

Input Parameters

m	INTEGER. The order of the orthogonal matrix Q ($m \geq 0$).
n	INTEGER. The number of columns of Q to be computed

$(0 \leq n \leq m)$.

k INTEGER. The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

a, *tau*, *work* REAL for sorgqr
DOUBLE PRECISION for dorgqr

Arrays:
a(*lda*,*) and *tau*(*) are the arrays returned by sgeqrf / dgeqrf or sgeqpf / dgeqpf.
The second dimension of *a* must be at least $\max(1, n)$.
The dimension of *tau* must be at least $\max(1, k)$.
work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The first dimension of *a*; at least $\max(1, m)$.

lwork INTEGER. The size of the *work* array ($lwork \geq n$).
If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.
See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a Overwritten by *n* leading columns of the *m*-by-*m* orthogonal matrix Q .

work(1) If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `orgqr` interface are the following:

a Holds the matrix *A* of size (m, n) .
tau Holds the vector of length (k)

Application Notes

For better performance, try using $lwork = n * blocksize$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed *Q* differs from an exactly orthogonal matrix by a matrix *E* such that

$$\|E\|_2 = O(\epsilon) \|A\|_2 \text{ where } \epsilon \text{ is the machine precision.}$$

The total number of floating-point operations is approximately $4 * m * n * k - 2 * (m + n) * k^2 + (4/3) * k^3$.

If $n = k$, the number is approximately $(2/3) * n^2 * (3m - n)$.

The complex counterpart of this routine is [?ungqr](#).

?ormqr

Multiplies a real matrix by the orthogonal matrix Q of the QR factorization formed by ?geqrf or ?geqpf.

Syntax

FORTRAN 77:

```
call sormqr(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call dormqr(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call ormqr(a, tau, c [,side] [,trans] [,info])
```

Description

The routine multiplies a real matrix C by Q or Q^T , where Q is the orthogonal matrix Q of the QR factorization formed by the routines [sgeqrf/dgeqrf](#) or [sgeqpf/dgeqpf](#).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products Q^*C , Q^T*C , $C*Q$, or $C*Q^T$ (overwriting the result on C).

Input Parameters

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^T is applied to C from the left. If <i>side</i> = 'R', Q or Q^T is applied to C from the right.
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'T'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'T', the routine multiplies C by Q^T .
<i>m</i>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: $0 \leq k \leq m$ if <i>side</i> = 'L'; $0 \leq k \leq n$ if <i>side</i> = 'R'.

<i>a</i> , <i>tau</i> , <i>c</i> , <i>work</i>	REAL for <i>sgeqrf</i> DOUBLE PRECISION for <i>dgeqrf</i>. Arrays: <i>a</i>(<i>lda</i>,*) and <i>tau</i>(*) are the arrays returned by <i>sgeqrf</i> / <i>dgeqrf</i> or <i>sgeqpf</i> / <i>dgeqpf</i>. The second dimension of <i>a</i> must be at least $\max(1, k)$. The dimension of <i>tau</i> must be at least $\max(1, k)$. <i>c</i>(<i>ldc</i>,*) contains the matrix <i>c</i>. The second dimension of <i>c</i> must be at least $\max(1, n)$ <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The first dimension of <i>a</i>. Constraints: <i>lda</i> $\geq \max(1, m)$ if <i>side</i> = 'L'; <i>lda</i> $\geq \max(1, n)$ if <i>side</i> = 'R'.
<i>ldc</i>	INTEGER. The first dimension of <i>c</i>. Constraint: <i>ldc</i> $\geq \max(1, m)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array. Constraints: <i>lwork</i> $\geq \max(1, n)$ if <i>side</i> = 'L'; <i>lwork</i> $\geq \max(1, m)$ if <i>side</i> = 'R'. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla. See <i>Application Notes</i> for the suggested value of <i>lwork</i>.

Output Parameters

<i>c</i>	Overwritten by the product Q^*C , $Q^T C$, C^*Q , or C^*Q^T (as specified by <i>side</i> and <i>trans</i>).
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ormqr` interface are the following:

<code>a</code>	Holds the matrix A of size (r,k) . $r = m$ if <code>side = 'L'</code> . $r = n$ if <code>side = 'R'</code> .
<code>tau</code>	Holds the vector of length (k) .
<code>c</code>	Holds the matrix C of size (m,n) .
<code>side</code>	Must be 'L' or 'R'. The default value is 'L'.
<code>trans</code>	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using `lwork = n*blocksize` (if `side = 'L'`) or `lwork = m*blocksize` (if `side = 'R'`) where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible `lwork` sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is [?unmqr](#).

?ungqr

Generates the complex unitary matrix Q of the QR factorization formed by ?geqrf.

Syntax

FORTRAN 77:

```
call cungqr(m, n, k, a, lda, tau, work, lwork, info)
call zungqr(m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call ungqr(a, tau [,info])
```

Description

The routine generates the whole or part of m -by- m unitary matrix Q of the QR factorization formed by the routines [cgeqrf/zgeqrf](#) or [cgeqpz/zgeqpz](#). Use this routine after a call to [cgeqrf/zgeqrf](#) or [cgeqpz/zgeqpz](#).

Usually Q is determined from the QR factorization of an m by p matrix A with $m \geq p$. To compute the whole matrix Q , use:

```
call ?ungqr(m, m, p, a, lda, tau, work, lwork, info)
```

To compute the leading p columns of Q (which form an orthonormal basis in the space spanned by the columns of A):

```
call ?ungqr(m, p, p, a, lda, tau, work, lwork, info)
```

To compute the matrix Q^k of the QR factorization of the leading k columns of the matrix A :

```
call ?ungqr(m, m, k, a, lda, tau, work, lwork, info)
```

To compute the leading k columns of Q^k (which form an orthonormal basis in the space spanned by the leading k columns of the matrix A):

```
call ?ungqr(m, k, k, a, lda, tau, work, lwork, info)
```

Input Parameters

m	INTEGER. The order of the unitary matrix Q ($m \geq 0$).
n	INTEGER. The number of columns of Q to be computed

	$(0 \leq n \leq m)$.
k	INTEGER. The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).
$a, \tau, work$	COMPLEX for <code>cungqr</code> DOUBLE COMPLEX for <code>zungqr</code> Arrays: $a(lda,*)$ and $\tau(*)$ are the arrays returned by <code>cgeqrf/zgeqrf</code> or <code>cgeqpz/zgeqpz</code> . The second dimension of a must be at least $\max(1, n)$. The dimension of τ must be at least $\max(1, k)$. $work$ is a workspace array, its dimension $\max(1, lwork)$.
lda	INTEGER. The first dimension of a ; at least $\max(1, m)$.
$lwork$	INTEGER. The size of the $work$ array ($lwork \geq n$). If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by <code>xerbla</code> . See <i>Application Notes</i> for the suggested value of $lwork$.

Output Parameters

a	Overwritten by n leading columns of the m -by- m unitary matrix Q .
$work(1)$	If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ungqr` interface are the following:

a	Holds the matrix A of size (m, n) .

tau Holds the vector of length (*k*).

Application Notes

For better performance, try using $lwork = n * blocksize$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set $lwork = -1$.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed *Q* differs from an exactly unitary matrix by a matrix *E* such that $\|E\|_2 = O(\epsilon) * \|A\|_2$, where ϵ is the machine precision.

The total number of floating-point operations is approximately $16 * m * n * k - 8 * (m + n) * k^2 + (16/3) * k^3$.

If $n = k$, the number is approximately $(8/3) * n^2 * (3m - n)$.

The real counterpart of this routine is [?orgqr](#).

?unmqr

Multiplies a complex matrix by the unitary matrix Q of the QR factorization formed by ?geqrf.

Syntax

FORTRAN 77:

```
call cunmqr(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call zunmqr(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call unmqr(a, tau, c [,side] [,trans] [,info])
```

Description

The routine multiplies a rectangular complex matrix c by Q or Q_H , where Q is the unitary matrix Q of the QR factorization formed by the routines [cgeqrf/zgeqrf](#) or [cgeqpf/zgeqpf](#).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products Q^*C , $Q^H * C$, $C * Q$, or $C * Q^H$ (overwriting the result on c).

Input Parameters

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q_H is applied to c from the left. If <i>side</i> = 'R', Q or Q_H is applied to c from the right.
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'C'. If <i>trans</i> = 'N', the routine multiplies c by Q. If <i>trans</i> = 'C', the routine multiplies c by Q^H.
<i>m</i>	INTEGER. The number of rows in the matrix c ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in c ($n \geq 0$).
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q. Constraints: $0 \leq k \leq m$ if <i>side</i> = 'L'; $0 \leq k \leq n$ if <i>side</i> = 'R'.
<i>a, c, tau, work</i>	COMPLEX for cgeqrf DOUBLE COMPLEX for zgeqrf. Arrays: $a(la,*)$ and $tau(*)$ are the arrays returned by cgeqrf / zgeqrf or cgeqpf / zgeqpf. The second dimension of a must be at least $\max(1, k)$. The dimension of tau must be at least $\max(1, k)$. $c ldc,*)$ contains the matrix c. The second dimension of c must be at least $\max(1, n)$ $work$ is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The first dimension of a. Constraints:

$lda \geq \max(1, m)$ if $side = 'L'$;
 $lda \geq \max(1, n)$ if $side = 'R'$.
 ldc INTEGER. The first dimension of c . Constraint:
 $ldc \geq \max(1, m)$.
 $lwork$ INTEGER. The size of the $work$ array. Constraints:
 $lwork \geq \max(1, n)$ if $side = 'L'$;
 $lwork \geq \max(1, m)$ if $side = 'R'$.
 If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#). See *Application notes* for the suggested value of $lwork$.

Output Parameters

c Overwritten by the product Q^*C , $Q^H C$, C^*Q , or C^*Q^H (as specified by $side$ and $trans$).
 $work(1)$ If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.
 $info$ INTEGER.
 If $info = 0$, the execution is successful.
 If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `unmqr` interface are the following:

a Holds the matrix A of size (r, k) .
 $r = m$ if $side = 'L'$.
 $r = n$ if $side = 'R'$.
 tau Holds the vector of length (k) .
 c Holds the matrix C of size (m, n) .

<i>side</i>	Must be 'L' or 'R'. The default value is 'L'.
<i>trans</i>	Must be 'N' or 'C'. The default value is 'N'.

Application Notes

For better performance, try using $lwork = n * blocksize$ (if $side = 'L'$) or $lwork = m * blocksize$ (if $side = 'R'$) where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of $lwork$ for the first run, or set $lwork = -1$.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if $lwork$ is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is [?ormqr](#).

?gelqf

Computes the LQ factorization of a general m -by- n matrix.

Syntax

FORTRAN 77:

```
call sgelqf(m, n, a, lda, tau, work, lwork, info)
call dgelqf(m, n, a, lda, tau, work, lwork, info)
call cgelqf(m, n, a, lda, tau, work, lwork, info)
call zgelqf(m, n, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call gelqf(a [, tau] [,info])
```

Description

The routine forms the LQ factorization of a general m -by- n matrix A (see [Orthogonal Factorizations](#)). No pivoting is performed.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ *elementary reflectors*. Routines are provided to work with Q in this representation.

Input Parameters

m INTEGER. The number of rows in the matrix A ($m \geq 0$).

n INTEGER. The number of columns in A ($n \geq 0$).

$a, work$ REAL for `sgelqf`
 DOUBLE PRECISION for `dgelqf`
 COMPLEX for `cgelqf`
 DOUBLE COMPLEX for `zgelqf`.
Arrays:
 $a(lda,*)$ contains the matrix A .
 The second dimension of a must be at least $\max(1, n)$.
 $work$ is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The first dimension of a ; at least $\max(1, m)$.

$lwork$ INTEGER. The size of the $work$ array; at least $\max(1, m)$.
 If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by `xerbla`.
 See *Application Notes* for the suggested value of $lwork$.

Output Parameters

a Overwritten by the factorization data as follows:
 If $m \leq n$, the elements above the diagonal are overwritten by the details of the unitary (orthogonal) matrix Q , and the lower triangle is overwritten by the corresponding elements of the lower triangular matrix L .

If $m > n$, the strictly upper triangular part is overwritten by the details of the matrix Q , and the remaining elements are overwritten by the corresponding elements of the m -by- n lower trapezoidal matrix L .

<i>tau</i>	REAL for <code>sgelqf</code> DOUBLE PRECISION for <code>dgelqf</code> COMPLEX for <code>cgelqf</code> DOUBLE COMPLEX for <code>zgelqf</code>. Array, DIMENSION at least $\max(1, \min(m, n))$. Contains additional information on the matrix Q.
<i>work(1)</i>	If <i>info</i> = 0, on exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gelqf` interface are the following:

<i>a</i>	Holds the matrix A of size (m, n) .
<i>tau</i>	Holds the vector of length $\min(m, n)$.

Application Notes

For better performance, try using $lwork = m * blocksize$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work(1)*) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed factorization is the exact factorization of a matrix $A + E$, where

$$\|E\|_2 = O(\epsilon) \|A\|_2.$$

The approximate number of floating-point operations for real flavors is

$$\begin{aligned} (4/3)n^3 & \quad \text{if } m = n, \\ (2/3)n^2(3m-n) & \quad \text{if } m > n, \\ (2/3)m^2(3n-m) & \quad \text{if } m < n. \end{aligned}$$

The number of operations for complex flavors is 4 times greater.

To find the minimum-norm solution of an underdetermined least squares problem minimizing $\|A^*x - b\|_2$ for all columns b of a given matrix B , you can call the following:

`?gelqf` (this routine) to factorize $A = L^*Q$;

`?trsm` (a BLAS routine) to solve $L^*Y = B$ for Y ;

`?ormlq` to compute $X = (Q_1)^T * Y$ (for real matrices);

`?unmlq` to compute $X = (Q_1)^H * Y$ (for complex matrices).

(The columns of the computed X are the minimum-norm solution vectors x . Here A is an m -by- n matrix with $m < n$; Q_1 denotes the first m columns of Q).

To compute the elements of Q explicitly, call

`?orglq` (for real matrices)

`?unglq` (for complex matrices).

?orglq

Generates the real orthogonal matrix Q of the LQ factorization formed by ?gelqf.

Syntax

FORTRAN 77:

```
call sorglq(m, n, k, a, lda, tau, work, lwork, info)
call dorglq(m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call orglq(a, tau [,info])
```

Description

The routine generates the whole or part of n -by- n orthogonal matrix Q of the LQ factorization formed by the routines [sgelqf](#)/[gelqf](#). Use this routine after a call to [sgelqf](#)/[dgelqf](#).

Usually Q is determined from the LQ factorization of an p -by- n matrix A with $n \geq p$. To compute the whole matrix Q , use:

```
call ?orglq(n, n, p, a, lda, tau, work, lwork, info)
```

To compute the leading p rows of Q , which form an orthonormal basis in the space spanned by the rows of A , use:

```
call ?orglq(p, n, p, a, lda, tau, work, lwork, info)
```

To compute the matrix Q^k of the LQ factorization of A 's leading k rows, use:

```
call ?orglq(n, n, k, a, lda, tau, work, lwork, info)
```

To compute the leading k rows of Q^k , which form an orthonormal basis in the space spanned by A 's leading k rows, use:

```
call ?orgqr(k, n, k, a, lda, tau, work, lwork, info)
```

Input Parameters

m INTEGER. The number of rows of Q to be computed
($0 \leq m \leq n$).

n INTEGER. The order of the orthogonal matrix Q ($n \geq m$).

<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq m$).
<i>a</i> , <i>tau</i> , <i>work</i>	REAL for <code>sorglq</code> DOUBLE PRECISION for <code>dorglq</code> Arrays: <i>a</i> (<i>lda</i> ,*) and <i>tau</i> (*) are the arrays returned by <code>sgelqf/dgelqf</code> . The second dimension of <i>a</i> must be at least $\max(1, n)$. The dimension of <i>tau</i> must be at least $\max(1, k)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; at least $\max(1, m)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array; at least $\max(1, m)$. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <code>xerbla</code> . See <i>Application Notes</i> for the suggested value of <i>lwork</i> .

Output Parameters

<i>a</i>	Overwritten by <i>m</i> leading rows of the <i>n</i> -by- <i>n</i> orthogonal matrix Q .
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `orglq` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>m</i> , <i>n</i>).
<i>tau</i>	Holds the vector of length (<i>k</i>).

Application Notes

For better performance, try using $lwork = m * blocksize$, where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed Q differs from an exactly orthogonal matrix by a matrix E such that $\|E\|_2 = O(\epsilon) * \|A\|_2$, where ϵ is the machine precision.

The total number of floating-point operations is approximately $4 * m * n * k - 2 * (m + n) * k^2 + (4/3) * k^3$.

If $m = k$, the number is approximately $(2/3) * m^2 * (3n - m)$.

The complex counterpart of this routine is [?unglq](#).

?ormlq

Multiplies a real matrix by the orthogonal matrix Q of the LQ factorization formed by ?gelqf.

Syntax

FORTRAN 77:

```
call sormlq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call dormlq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call ormlq(a, tau, c [,side] [,trans] [,info])
```

Description

The routine multiplies a real m -by- n matrix C by Q or Q^T , where Q is the orthogonal matrix Q of the LQ factorization formed by the routine [sgelqf](#)/[dgelqf](#).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products Q^*C , $Q^T C$, C^*Q , or $C^T Q$ (overwriting the result on C).

Input Parameters

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^T is applied to C from the left. If <i>side</i> = 'R', Q or Q^T is applied to C from the right.
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'T'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'T', the routine multiplies C by Q^T .
<i>m</i>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: $0 \leq k \leq m$ if <i>side</i> = 'L'; $0 \leq k \leq n$ if <i>side</i> = 'R'.
<i>a</i> , <i>c</i> , <i>tau</i> , <i>work</i>	REAL for sormlq DOUBLE PRECISION for dormlq . Arrays: <i>a</i> (<i>lda</i> ,*) and <i>tau</i> (*) are arrays returned by ?gelqf . The second dimension of <i>a</i> must be: at least $\max(1, m)$ if <i>side</i> = 'L'; at least $\max(1, n)$ if <i>side</i> = 'R'. The dimension of <i>tau</i> must be at least $\max(1, k)$. <i>c</i> (<i>ldc</i> ,*) contains the matrix C . The second dimension of <i>c</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The first dimension of *a*; $lda \geq \max(1, k)$.

ldc INTEGER. The first dimension of *c*; $ldc \geq \max(1, m)$.

lwork INTEGER. The size of the *work* array. Constraints:
 $lwork \geq \max(1, n)$ if *side* = 'L';
 $lwork \geq \max(1, m)$ if *side* = 'R'.
 If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#).
 See *Application Notes* for the suggested value of *lwork*.

Output Parameters

c Overwritten by the product Q^*C , $Q^T * C$, C^*Q , or C^*Q^T (as specified by *side* and *trans*).

work(1) If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ormlq` interface are the following:

a Holds the matrix *A* of size (*k*,*m*).

tau Holds the vector of length (*k*).

c Holds the matrix *C* of size (*m*,*n*).

side Must be 'L' or 'R'. The default value is 'L'.

trans Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using $lwork = n * blocksize$ (if $side = 'L'$) or $lwork = m * blocksize$ (if $side = 'R'$) where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is [?unmlq](#).

?unglq

Generates the complex unitary matrix Q of the LQ factorization formed by [?gelqf](#).

Syntax

FORTRAN 77:

```
call cunqlq(m, n, k, a, lda, tau, work, lwork, info)
call zunqlq(m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call unqlq(a, tau [,info])
```

Description

The routine generates the whole or part of n -by- n unitary matrix Q of the LQ factorization formed by the routines [cgelqf](#)/[zgelqf](#). Use this routine after a call to [cgelqf](#)/[zgelqf](#).

Usually Q is determined from the LQ factorization of an p -by- n matrix A with $n < p$. To compute the whole matrix Q , use:

```
call ?unglq(n, n, p, a, lda, tau, work, lwork, info)
```

To compute the leading p rows of Q , which form an orthonormal basis in the space spanned by the rows of A , use:

```
call ?unglq(p, n, p, a, lda, tau, work, lwork, info)
```

To compute the matrix Q^k of the LQ factorization of the leading k rows of the matrix A , use:

```
call ?unglq(n, n, k, a, lda, tau, work, lwork, info)
```

To compute the leading k rows of Q^k , which form an orthonormal basis in the space spanned by the leading k rows of the matrix A , use:

```
call ?ungqr(k, n, k, a, lda, tau, work, lwork, info)
```

Input Parameters

m	INTEGER. The number of rows of Q to be computed ($0 \leq m \leq n$).
n	INTEGER. The order of the unitary matrix Q ($n \geq m$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq m$).
$a, \tau, work$	COMPLEX for <code>cunglq</code> DOUBLE COMPLEX for <code>zunglq</code> Arrays: $a(lda,*)$ and $\tau(*)$ are the arrays returned by <code>sgelqf/dgelqf</code> . The second dimension of a must be at least $\max(1, n)$. The dimension of τ must be at least $\max(1, k)$. $work$ is a workspace array, its dimension $\max(1, lwork)$.
lda	INTEGER. The first dimension of a ; at least $\max(1, m)$.
$lwork$	INTEGER. The size of the $work$ array; at least $\max(1, m)$. If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by <code>xerbla</code> . See <i>Application Notes</i> for the suggested value of $lwork$.

Output Parameters

<i>a</i>	Overwritten by <i>m</i> leading rows of the <i>n</i> -by- <i>n</i> unitary matrix <i>Q</i> .
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `unglq` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>m</i> , <i>n</i>).
<i>tau</i>	Holds the vector of length (<i>k</i>).

Application Notes

For better performance, try using *lwork* = *m***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed Q differs from an exactly unitary matrix by a matrix E such that $\|E\|_2 = O(\epsilon) * \|A\|_2$, where ϵ is the machine precision.

The total number of floating-point operations is approximately $16 * m * n * k - 8 * (m + n) * k^2 + (16/3) * k^3$.

If $m = k$, the number is approximately $(8/3) * m^2 * (3n - m)$.

The real counterpart of this routine is [?orglq](#).

?unmlq

Multiplies a complex matrix by the unitary matrix Q of the LQ factorization formed by ?gelqf.

Syntax

FORTRAN 77:

```
call cunmlq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call zunmlq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call unmlq(a, tau, c [,side] [,trans] [,info])
```

Description

The routine multiplies a real m -by- n matrix C by Q or Q^H , where Q is the unitary matrix Q of the LQ factorization formed by the routine [cgelqf/zgelqf](#).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products $Q * C$, $Q^H * C$, $C * Q$, or $C * Q^H$ (overwriting the result on C).

Input Parameters

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^H is applied to C from the left. If <i>side</i> = 'R', Q or Q^H is applied to C from the right.
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'C'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'C', the routine multiplies C by Q_H .

m INTEGER. The number of rows in the matrix C ($m \geq 0$).
 n INTEGER. The number of columns in C ($n \geq 0$).
 k INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints:
 $0 \leq k \leq m$ if $side = 'L'$;
 $0 \leq k \leq n$ if $side = 'R'$.
 $a, c, tau, work$ COMPLEX for `cunmlq`
DOUBLE COMPLEX for `zunmlq`.
Arrays:
 $a(lda,*)$ and $tau(*)$ are arrays returned by `?gelqf`.
The second dimension of a must be:
at least $\max(1, m)$ if $side = 'L'$;
at least $\max(1, n)$ if $side = 'R'$.
The dimension of tau must be at least $\max(1, k)$.
 $c(ldc,*)$ contains the matrix C .
The second dimension of c must be at least $\max(1, n)$
 $work$ is a workspace array, its dimension $\max(1, lwork)$.
 lda INTEGER. The first dimension of a ; $lda \geq \max(1, k)$.
 ldc INTEGER. The first dimension of c ; $ldc \geq \max(1, m)$.
 $lwork$ INTEGER. The size of the $work$ array. Constraints:
 $lwork \geq \max(1, n)$ if $side = 'L'$;
 $lwork \geq \max(1, m)$ if $side = 'R'$.
If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by `xerbla`.
See *Application Notes* for the suggested value of $lwork$.

Output Parameters

c Overwritten by the product Q^*C , $Q^H * C$, C^*Q , or C^*Q^H (as specified by $side$ and $trans$).

<code>work(1)</code>	If <code>info = 0</code> , on exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance. Use this <code>lwork</code> for subsequent runs.
<code>info</code>	INTEGER. If <code>info = 0</code> , the execution is successful. If <code>info = -i</code> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `unmlq` interface are the following:

<code>a</code>	Holds the matrix <i>A</i> of size (<i>k</i> , <i>m</i>).
<code>tau</code>	Holds the vector of length (<i>k</i>).
<code>c</code>	Holds the matrix <i>C</i> of size (<i>m</i> , <i>n</i>).
<code>side</code>	Must be 'L' or 'R'. The default value is 'L'.
<code>trans</code>	Must be 'N' or 'C'. The default value is 'N'.

Application Notes

For better performance, try using `lwork = n*blocksize` (if `side = 'L'`) or `lwork = m*blocksize` (if `side = 'R'`) where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of `lwork` for the first run, or set `lwork = -1`.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If `lwork = -1`, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if `lwork` is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is [?ormlq](#).

?geqlf

Computes the QL factorization of a general m -by- n matrix.

Syntax

FORTRAN 77:

```
call sgeqlf(m, n, a, lda, tau, work, lwork, info)
call dgeqlf(m, n, a, lda, tau, work, lwork, info)
call cgeqlf(m, n, a, lda, tau, work, lwork, info)
call zgeqlf(m, n, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call geqlf(a [, tau] [,info])
```

Description

The routine forms the QL factorization of a general m -by- n matrix A . No pivoting is performed.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ *elementary reflectors*. Routines are provided to work with Q in this representation.

Input Parameters

m	INTEGER. The number of rows in the matrix A ($m \geq 0$).
n	INTEGER. The number of columns in A ($n \geq 0$).
$a, work$	REAL for sgeqlf DOUBLE PRECISION for dgeqlf COMPLEX for cgeqlf DOUBLE COMPLEX for zgeqlf. Arrays: $a(lda,*)$ contains the matrix A . The second dimension of a must be at least $\max(1, n)$. $work$ is a workspace array, its dimension $\max(1, lwork)$.
lda	INTEGER. The first dimension of a ; at least $\max(1, m)$.
$lwork$	INTEGER. The size of the $work$ array; at least $\max(1, n)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#). See *Application Notes* for the suggested value of $lwork$.

Output Parameters

a	Overwritten on exit by the factorization data as follows: if $m \geq n$, the lower triangle of the subarray $a(m-n+1:m, 1:n)$ contains the n -by- n lower triangular matrix L ; if $m \leq n$, the elements on and below the $(n-m)$ -th superdiagonal contain the m -by- n lower trapezoidal matrix L ; in both cases, the remaining elements, with the array tau , represent the orthogonal/unitary matrix Q as a product of elementary reflectors.
tau	REAL for sgeqlf DOUBLE PRECISION for dgeqlf COMPLEX for cgeqlf DOUBLE COMPLEX for zgeqlf. Array, DIMENSION at least $\max(1, \min(m, n))$. Contains scalar factors of the elementary reflectors for the matrix Q .
$work(1)$	If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `geqlf` interface are the following:

a	Holds the matrix A of size (m, n) .
tau	Holds the vector of length $\min(m, n)$.

Application Notes

For better performance, try using `lwork = n*blocksize`, where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible `lwork` sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Related routines include:

<code>?orgql</code>	to generate matrix Q (for real matrices);
<code>?ungql</code>	to generate matrix Q (for complex matrices);
<code>?ormql</code>	to apply matrix Q (for real matrices);
<code>?unmql</code>	to apply matrix Q (for complex matrices).

?orgql

Generates the real matrix Q of the QL factorization formed by ?geqlf.

Syntax

FORTRAN 77:

```
call sorgql(m, n, k, a, lda, tau, work, lwork, info)
call dorgql(m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call orgql(a, tau [,info])
```

Description

The routine generates an m -by- n real matrix Q with orthonormal columns, which is defined as the last n columns of a product of k elementary reflectors $H(i)$ of order m : $Q = H(k) * \dots * H(2) * H(1)$ as returned by the routines [sgeqlf/dgeqlf](#). Use this routine after a call to [sgeqlf/dgeqlf](#).

Input Parameters

m	INTEGER. The number of rows of the matrix Q ($m \geq 0$).
n	INTEGER. The number of columns of the matrix Q ($m \geq n \geq 0$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix Q ($n \geq k \geq 0$).
$a, \tau, work$	REAL for sorgql DOUBLE PRECISION for dorgql Arrays: $a(lda,*)$, $\tau(*)$. On entry, the $(n - k + i)$th column of a must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by sgeqlf/dgeqlf in the last k columns of its array argument a; $\tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by sgeqlf/dgeqlf; The second dimension of a must be at least $\max(1, n)$. The dimension of τ must be at least $\max(1, k)$. $work$ is a workspace array, its dimension $\max(1, lwork)$.
lda	INTEGER. The first dimension of a ; at least $\max(1, m)$.
$lwork$	INTEGER. The size of the $work$ array; at least $\max(1, n)$. If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by xerbla. See <i>Application Notes</i> for the suggested value of $lwork$.

Output Parameters

<i>a</i>	Overwritten by the m -by- n matrix Q .
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `orgql` interface are the following:

<i>a</i>	Holds the matrix A of size (m, n) .
<i>tau</i>	Holds the vector of length (k) .

Application Notes

For better performance, try using *lwork* = $n \times \text{blocksize}$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is [?ungql](#).

?ungql

Generates the complex matrix Q of the QL factorization formed by ?geqlf.

Syntax

FORTRAN 77:

```
call cungql(m, n, k, a, lda, tau, work, lwork, info)
call zungql(m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call ungql(a, tau [,info])
```

Description

The routine generates an m -by- n complex matrix Q with orthonormal columns, which is defined as the last n columns of a product of k elementary reflectors $H(i)$ of order m : $Q = H(k) * \dots * H(2) * H(1)$ as returned by the routines [cgeqlf/zgeqlf](#). Use this routine after a call to [cgeqlf/zgeqlf](#).

Input Parameters

m	INTEGER. The number of rows of the matrix Q ($m \geq 0$).
n	INTEGER. The number of columns of the matrix Q ($m \geq n \geq 0$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix Q ($n \geq k \geq 0$).
$a, \tau, work$	COMPLEX for cungql DOUBLE COMPLEX for zungql Arrays: $a(lda,*)$, $\tau(*)$, $work(lwork)$. On entry, the $(n - k + i)$th column of a must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by cgeqlf/zgeqlf in the last k columns of its array argument a; $\tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by cgeqlf/zgeqlf;

The second dimension of a must be at least $\max(1, n)$.
 The dimension of τ must be at least $\max(1, k)$.
 $work$ is a workspace array, its dimension $\max(1, lwork)$.
lda INTEGER. The first dimension of a ; at least $\max(1, m)$.
lwork INTEGER. The size of the $work$ array; at least $\max(1, n)$.
 If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).
 See *Application Notes* for the suggested value of $lwork$.

Output Parameters

a Overwritten by the m -by- n matrix Q .
 $work(1)$ If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.
info INTEGER.
 If $info = 0$, the execution is successful.
 If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ungql` interface are the following:

a Holds the matrix A of size (m, n) .
 τ Holds the vector of length (k) .

Application Notes

For better performance, try using $lwork = n * blocksize$, where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of $lwork$ for the first run, or set $lwork = -1$.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If `lwork = -1`, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if `lwork` is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is [?orgql](#).

?ormql

Multiplies a real matrix by the orthogonal matrix Q of the QL factorization formed by ?geqlf.

Syntax

FORTRAN 77:

```
call sormql(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call dormql(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call ormql(a, tau, c [,side] [,trans] [,info])
```

Description

This routine multiplies a real m -by- n matrix C by Q or Q^T , where Q is the orthogonal matrix Q of the QL factorization formed by the routine [sgeqlf/dgeqlf](#).

Depending on the parameters `side` and `trans`, the routine [?ormql](#) can form one of the matrix products Q^*C , Q^T*C , $C*Q$, or $C*Q^T$ (overwriting the result over C).

Input Parameters

<code>side</code>	CHARACTER*1. Must be either 'L' or 'R'. If <code>side = 'L'</code> , Q or Q^T is applied to C from the left. If <code>side = 'R'</code> , Q or Q^T is applied to C from the right.
<code>trans</code>	CHARACTER*1. Must be either 'N' or 'T'.

If *trans* = 'N', the routine multiplies *c* by Q .
 If *trans* = 'T', the routine multiplies *c* by Q^T .

m INTEGER. The number of rows in the matrix *c* ($m \geq 0$).

n INTEGER. The number of columns in *c* ($n \geq 0$).

k INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints:
 $0 \leq k \leq m$ if *side* = 'L';
 $0 \leq k \leq n$ if *side* = 'R'.

a, *tau*, *c*, *work* REAL for sormql
 DOUBLE PRECISION for dormql.
 Arrays: *a*(*lda*,*), *tau*(*), *c*(*ldc*,*).
 On entry, the *i*th column of *a* must contain the vector which defines the elementary reflector H_i , for $i = 1, 2, \dots, k$, as returned by sgeqlf/dgeqlf in the last *k* columns of its array argument *a*.
 The second dimension of *a* must be at least $\max(1, k)$.
tau(*i*) must contain the scalar factor of the elementary reflector H_i , as returned by sgeqlf/dgeqlf.
 The dimension of *tau* must be at least $\max(1, k)$.
c(*ldc*,*) contains the *m*-by-*n* matrix *C*.
 The second dimension of *c* must be at least $\max(1, n)$.
work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The first dimension of *a*;
 if *side* = 'L', $lda \geq \max(1, m)$;
 if *side* = 'R', $lda \geq \max(1, n)$.

ldc INTEGER. The first dimension of *c*; $ldc \geq \max(1, m)$.

lwork INTEGER. The size of the *work* array. Constraints:
 $lwork \geq \max(1, n)$ if *side* = 'L';
 $lwork \geq \max(1, m)$ if *side* = 'R'.

If `lwork = -1`, then a workspace query is assumed; the routine only calculates the optimal size of the `work` array, returns this value as the first entry of the `work` array, and no error message related to `lwork` is issued by `xerbla`. See *Application Notes* for the suggested value of `lwork`.

Output Parameters

<code>c</code>	Overwritten by the product Q^*C , Q^T*C , $C*Q$, or $C*Q^T$ (as specified by <code>side</code> and <code>trans</code>).
<code>work(1)</code>	If <code>info = 0</code> , on exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance. Use this <code>lwork</code> for subsequent runs.
<code>info</code>	INTEGER. If <code>info = 0</code> , the execution is successful. If <code>info = -i</code> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ormql` interface are the following:

<code>a</code>	Holds the matrix <i>A</i> of size (<i>r</i> , <i>k</i>). $r = m$ if <code>side = 'L'</code> . $r = n$ if <code>side = 'R'</code> .
<code>tau</code>	Holds the vector of length (<i>k</i>).
<code>c</code>	Holds the matrix <i>C</i> of size (<i>m</i> , <i>n</i>).
<code>side</code>	Must be 'L' or 'R'. The default value is 'L'.
<code>trans</code>	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using `lwork = n*blocksize` (if `side = 'L'`) or `lwork = m*blocksize` (if `side = 'R'`) where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible `lwork` sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is [?unmql](#).

[?unmql](#)

Multiplies a complex matrix by the unitary matrix Q of the QL factorization formed by [?geqlf](#).

Syntax

FORTRAN 77:

```
call cunmql(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call zunmql(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call unmql(a, tau, c [,side] [,trans] [,info])
```

Description

The routine multiplies a complex m -by- n matrix c by Q or Q^H , where Q is the unitary matrix Q of the QL factorization formed by the routine [cgeqlf/zgeqlf](#).

Depending on the parameters `side` and `trans`, the routine [?unmql](#) can form one of the matrix products $Q^H C$, $Q^H * C$, $C * Q$, or $C * Q^H$ (overwriting the result over c).

Input Parameters

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^H is applied to c from the left. If <i>side</i> = 'R', Q or Q^H is applied to c from the right.
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'C'. If <i>trans</i> = 'N', the routine multiplies c by Q. If <i>trans</i> = 'C', the routine multiplies c by Q^H.
<i>m</i>	INTEGER. The number of rows in the matrix c ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in c ($n \geq 0$).
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q. Constraints: $0 \leq k \leq m$ if <i>side</i> = 'L'; $0 \leq k \leq n$ if <i>side</i> = 'R'.
<i>a, tau, c, work</i>	COMPLEX for cunmq1 DOUBLE COMPLEX for zunmq1. Arrays: <i>a</i>(<i>lda</i>,*), <i>tau</i>(*), <i>c</i>(<i>ldc</i>,*), <i>work</i>(<i>lwork</i>). On entry, the <i>i</i>-th column of <i>a</i> must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by <i>cgeqlf/zgeqlf</i> in the last <i>k</i> columns of its array argument <i>a</i>. The second dimension of <i>a</i> must be at least $\max(1, k)$. <i>tau</i>(<i>i</i>) must contain the scalar factor of the elementary reflector $H(i)$, as returned by <i>cgeqlf/zgeqlf</i>. The dimension of <i>tau</i> must be at least $\max(1, k)$. <i>c</i>(<i>ldc</i>,*) contains the <i>m</i>-by-<i>n</i> matrix C. The second dimension of <i>c</i> must be at least $\max(1, n)$ <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldc</i>	INTEGER. The first dimension of <i>c</i> ; $ldc \geq \max(1, m)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array. Constraints: $lwork \geq \max(1, n)$ if <i>side</i> = 'L'; $lwork \geq \max(1, m)$ if <i>side</i> = 'R'.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the `work` array, returns this value as the first entry of the `work` array, and no error message related to `lwork` is issued by `xerbla`. See *Application Notes* for the suggested value of `lwork`.

Output Parameters

<code>c</code>	Overwritten by the product Q^*C , $Q^H C$, C^*Q , or C^*Q^H (as specified by <code>side</code> and <code>trans</code>).
<code>work(1)</code>	If <code>info = 0</code> , on exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance. Use this <code>lwork</code> for subsequent runs.
<code>info</code>	INTEGER. If <code>info = 0</code> , the execution is successful. If <code>info = -i</code> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `unmql` interface are the following:

<code>a</code>	Holds the matrix <i>A</i> of size (r, k) . $r = m$ if <code>side = 'L'</code> . $r = n$ if <code>side = 'R'</code> .
<code>tau</code>	Holds the vector of length (k) .
<code>c</code>	Holds the matrix <i>C</i> of size (m, n) .
<code>side</code>	Must be 'L' or 'R'. The default value is 'L'.
<code>trans</code>	Must be 'N' or 'C'. The default value is 'N'.

Application Notes

For better performance, try using `lwork = n*blocksize` (if `side = 'L'`) or `lwork = m*blocksize` (if `side = 'R'`) where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of `lwork` for the first run, or set `lwork = -1`.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If `lwork = -1`, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if `lwork` is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is [?ormql](#).

?gerqf

Computes the RQ factorization of a general m -by- n matrix.

Syntax

FORTRAN 77:

```
call sgerqf(m, n, a, lda, tau, work, lwork, info)
call dgerqf(m, n, a, lda, tau, work, lwork, info)
call cgerqf(m, n, a, lda, tau, work, lwork, info)
call zgerqf(m, n, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call gerqf(a [, tau] [,info])
```

Description

The routine forms the RQ factorization of a general m -by- n matrix A . No pivoting is performed.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ *elementary reflectors*. Routines are provided to work with Q in this representation.

Input Parameters

m	INTEGER. The number of rows in the matrix A ($m \geq 0$).
n	INTEGER. The number of columns in A ($n \geq 0$).
$a, work$	REAL for sgerqf DOUBLE PRECISION for dgerqf COMPLEX for cgerqf DOUBLE COMPLEX for zgerqf. Arrays: $a(lda,*)$ contains the m -by- n matrix A . The second dimension of a must be at least $\max(1, n)$. $work$ is a workspace array, its dimension $\max(1, lwork)$.
lda	INTEGER. The first dimension of a ; at least $\max(1, m)$.
$lwork$	INTEGER. The size of the $work$ array; $lwork \geq \max(1, m)$. If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by xerbla. See Application Notes for the suggested value of $lwork$.

Output Parameters

a	Overwritten on exit by the factorization data as follows: if $m \leq n$, the upper triangle of the subarray $a(1:m, n-m+1:n)$ contains the m -by- m upper triangular matrix R ; if $m \geq n$, the elements on and above the $(m-n)$ th subdiagonal contain the m -by- n upper trapezoidal matrix R ; in both cases, the remaining elements, with the array tau , represent the orthogonal/unitary matrix Q as a product of $\min(m, n)$ elementary reflectors.
tau	REAL for sgerqf DOUBLE PRECISION for dgerqf COMPLEX for cgerqf DOUBLE COMPLEX for zgerqf.

	Array, <code>DIMENSION</code> at least $\max(1, \min(m, n))$. Contains scalar factors of the elementary reflectors for the matrix Q .
<code>work(1)</code>	If <code>info = 0</code> , on exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	INTEGER. If <code>info = 0</code> , the execution is successful. If <code>info = -i</code> , the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gerqf` interface are the following:

<code>a</code>	Holds the matrix A of size (m, n) .
<code>tau</code>	Holds the vector of length $\min(m, n)$.

Application Notes

For better performance, try using `lwork = m*blocksize`, where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible `lwork` sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Related routines include:

<code>?orgrq</code>	to generate matrix Q (for real matrices);
<code>?ungrq</code>	to generate matrix Q (for complex matrices);
<code>?ormrq</code>	to apply matrix Q (for real matrices);
<code>?unmrq</code>	to apply matrix Q (for complex matrices).

?orgrq

Generates the real matrix Q of the RQ factorization formed by ?gerqf.

Syntax

FORTRAN 77:

```
call sorgrq(m, n, k, a, lda, tau, work, lwork, info)
call dorgrq(m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call orgrq(a, tau [,info])
```

Description

The routine generates an m -by- n real matrix Q with orthonormal rows, which is defined as the last m rows of a product of k elementary reflectors $H(i)$ of order n : $Q = H(1) * H(2) * \dots * H(k)$ as returned by the routines `sgerqf/dgerqf`. Use this routine after a call to `sgerqf/dgerqf`.

Input Parameters

m	INTEGER. The number of rows of the matrix Q ($m \geq 0$).
n	INTEGER. The number of columns of the matrix Q ($n \geq m$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix Q ($m \geq k \geq 0$).
$a, \tau, work$	REAL for <code>sorgrq</code> DOUBLE PRECISION for <code>dorgrq</code> Arrays: $a(lda,*)$, $\tau(*)$.

On entry, the $(m - k + i)$ -th row of a must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by `sgerqf/dgerqf` in the last k rows of its array argument a ;

$\tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by `sgerqf/dgerqf`;

The second dimension of a must be at least $\max(1, n)$.

The dimension of τ must be at least $\max(1, k)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda

INTEGER. The first dimension of a ; at least $\max(1, m)$.

$lwork$

INTEGER. The size of the $work$ array; at least $\max(1, m)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by `xerbla`.

See *Application Notes* for the suggested value of $lwork$.

Output Parameters

a

Overwritten by the m -by- n matrix Q .

$work(1)$

If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `orgrq` interface are the following:

a

Holds the matrix A of size (m, n) .

τ

Holds the vector of length (k) .

Application Notes

For better performance, try using `lwork = m*blocksize`, where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible `lwork` sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is [?ungrq](#).

?ungrq

Generates the complex matrix Q of the RQ factorization formed by [?gerqf](#).

Syntax

FORTRAN 77:

```
call cungrq(m, n, k, a, lda, tau, work, lwork, info)
call zungrq(m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call ungrq(a, tau [,info])
```

Description

The routine generates an m -by- n complex matrix Q with orthonormal rows, which is defined as the last m rows of a product of k elementary reflectors $H(i)$ of order n : $Q = H(1)H^*H(2)H^* \dots H(k)H$ as returned by the routines [sgerqf/dgerqf](#). Use this routine after a call to [sgerqf/dgerqf](#).

Input Parameters

m	INTEGER. The number of rows of the matrix Q ($m \geq 0$).
n	INTEGER. The number of columns of the matrix Q ($n \geq m$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix Q ($m \geq k \geq 0$).
$a, \tau, work$	REAL for cungrq DOUBLE PRECISION for zungrq Arrays: $a(lda,*)$, $\tau(*)$, $work(lwork)$. On entry, the $(m - k + i)$th row of a must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by sgerqf/dgerqf in the last k rows of its array argument a; $\tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by sgerqf/dgerqf; The second dimension of a must be at least $\max(1, n)$. The dimension of τ must be at least $\max(1, k)$. $work$ is a workspace array, its dimension $\max(1, lwork)$.
lda	INTEGER. The first dimension of a ; at least $\max(1, m)$.
$lwork$	INTEGER. The size of the $work$ array; at least $\max(1, m)$. If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by xerbla. See <i>Application Notes</i> for the suggested value of $lwork$.

Output Parameters

a	Overwritten by the m -by- n matrix Q .

`work(1)` If `info = 0`, on exit `work(1)` contains the minimum value of `lwork` required for optimum performance. Use this `lwork` for subsequent runs.

`info` INTEGER.
 If `info = 0`, the execution is successful.
 If `info = -i`, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ungrq` interface are the following:

`a` Holds the matrix *A* of size (*m*, *n*).

`tau` Holds the vector of length (*k*).

Application Notes

For better performance, try using `lwork = m*blocksize`, where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of `lwork` for the first run, or set `lwork = -1`.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If `lwork = -1`, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if `lwork` is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is `?orgrq`.

?ormrq

Multiplies a real matrix by the orthogonal matrix Q of the RQ factorization formed by ?gerqf.

Syntax

FORTRAN 77:

```
call sormrq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call dormrq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call ormqr(a, tau, c [,side] [,trans] [,info])
```

Description

The routine multiplies a real m -by- n matrix C by Q or Q^T , where Q is the real orthogonal matrix defined as a product of k elementary reflectors H_i : $Q = H_1 H_2 \dots H_k$ as returned by the RQ factorization routine [sgerqf/dgerqf](#).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products Q^*C , $Q^T C$, C^*Q , or C^*Q^T (overwriting the result over C).

Input Parameters

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^T is applied to C from the left. If <i>side</i> = 'R', Q or Q^T is applied to C from the right.
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'T'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'T', the routine multiplies C by Q^T .
<i>m</i>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: $0 \leq k \leq m$, if <i>side</i> = 'L'; $0 \leq k \leq n$, if <i>side</i> = 'R'.

a, *tau*, *c*, *work*

REAL for `sormrq`
DOUBLE PRECISION for `dormrq`.
Arrays: *a*(*lda*,*), *tau*(*), *c*(*ldc*,*).

On entry, the *i*th row of *a* must contain the vector which defines the elementary reflector H_i , for $i = 1, 2, \dots, k$, as returned by `sgerqf`/`dgerqf` in the last *k* rows of its array argument *a*.
The second dimension of *a* must be at least $\max(1, m)$ if *side* = 'L', and at least $\max(1, n)$ if *side* = 'R'.
tau(*i*) must contain the scalar factor of the elementary reflector H_i , as returned by `sgerqf`/`dgerqf`.
The dimension of *tau* must be at least $\max(1, k)$.
c(*ldc*,*) contains the *m*-by-*n* matrix *C*.
The second dimension of *c* must be at least $\max(1, n)$.
work is a workspace array, its dimension $\max(1, lwork)$.

lda

INTEGER. The first dimension of *a*; $lda \geq \max(1, k)$.

ldc

INTEGER. The first dimension of *c*; $ldc \geq \max(1, m)$.

lwork

INTEGER. The size of the *work* array. Constraints:
 $lwork \geq \max(1, n)$ if *side* = 'L';
 $lwork \geq \max(1, m)$ if *side* = 'R'.
If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by `xerbla`.
See *Application Notes* for the suggested value of *lwork*.

Output Parameters

c

Overwritten by the product Q^*C , $Q^T C$, C^*Q , or C^*Q^T (as specified by *side* and *trans*).

work(1)

If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info

INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ormrq` interface are the following:

<code>a</code>	Holds the matrix A of size (k, m) .
<code>tau</code>	Holds the vector of length (k) .
<code>c</code>	Holds the matrix C of size (m, n) .
<code>side</code>	Must be 'L' or 'R'. The default value is 'L'.
<code>trans</code>	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using `lwork = n*blocksize` (if `side = 'L'`) or `lwork = m*blocksize` (if `side = 'R'`) where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible `lwork` sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is [?unmrq](#).

zunmrq

Multiplies a complex matrix by the unitary matrix Q of the RQ factorization formed by `zgerqf`.

Syntax

FORTRAN 77:

```
call cunmrq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call zunmrq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call unmrq(a, tau, c [,side] [,trans] [,info])
```

Description

The routine multiplies a complex m -by- n matrix C by Q or Q^H , where Q is the complex unitary matrix defined as a product of k elementary reflectors $H(i)$ of order n : $Q = H(1)H^*H(2)H^* \dots H(k)H^*$. H has returned by the RQ factorization routine [cgerqf/zgerqf](#).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products Q^*C , $Q^H * C$, $C * Q$, or $C * Q^H$ (overwriting the result over C).

Input Parameters

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^H is applied to C from the left. If <i>side</i> = 'R', Q or Q^H is applied to C from the right.
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'C'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'C', the routine multiplies C by Q^H .
<i>m</i>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: $0 \leq k \leq m$, if <i>side</i> = 'L'; $0 \leq k \leq n$, if <i>side</i> = 'R'.

a, *tau*, *c*, *work*

COMPLEX for cunmrq
DOUBLE COMPLEX for zunmrq.
Arrays: *a*(*lda*,*), *tau*(*), *c*(*ldc*,*), *work*(*lwork*).
On entry, the *i*th row of *a* must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by cgerqf/zgerqf in the last *k* rows of its array argument *a*.
The second dimension of *a* must be at least $\max(1, m)$ if *side* = 'L', and at least $\max(1, n)$ if *side* = 'R'.
tau(*i*) must contain the scalar factor of the elementary reflector $H(i)$, as returned by cgerqf/zgerqf.
The dimension of *tau* must be at least $\max(1, k)$.
c(*ldc*,*) contains the *m*-by-*n* matrix *C*.
The second dimension of *c* must be at least $\max(1, n)$
work is a workspace array, its dimension $\max(1, lwork)$.

lda

INTEGER. The first dimension of *a*; $lda \geq \max(1, k)$.

ldc

INTEGER. The first dimension of *c*; $ldc \geq \max(1, m)$.

lwork

INTEGER. The size of the *work* array. Constraints:
 $lwork \geq \max(1, n)$ if *side* = 'L';
 $lwork \geq \max(1, m)$ if *side* = 'R'.
If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.
See *Application Notes* for the suggested value of *lwork*.

Output Parameters

c

Overwritten by the product Q^*C , $Q^H * C$, C^*Q , or C^*Q^H (as specified by *side* and *trans*).

work(1)

If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info

INTEGER.
If *info* = 0, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `unmrq` interface are the following:

<code>a</code>	Holds the matrix A of size (k, m) .
<code>tau</code>	Holds the vector of length (k) .
<code>c</code>	Holds the matrix C of size (m, n) .
<code>side</code>	Must be 'L' or 'R'. The default value is 'L'.
<code>trans</code>	Must be 'N' or 'C'. The default value is 'N'.

Application Notes

For better performance, try using $lwork = n * blocksize$ (if $side = 'L'$) or $lwork = m * blocksize$ (if $side = 'R'$) where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of $lwork$ for the first run, or set $lwork = -1$.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if $lwork$ is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is `?ormrq`.

?tzzrf

Reduces the upper trapezoidal matrix A to upper triangular form.

Syntax

FORTRAN 77:

```
call stzzrf(m, n, a, lda, tau, work, lwork, info)
call dtzzrf(m, n, a, lda, tau, work, lwork, info)
call ctzzrf(m, n, a, lda, tau, work, lwork, info)
call ztzzrf(m, n, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call tzzrf(a [, tau] [,info])
```

Description

This routine reduces the m -by- n ($m \leq n$) real/complex upper trapezoidal matrix A to upper triangular form by means of orthogonal/unitary transformations. The upper trapezoidal matrix A is factored as

$$A = (R \ 0) * Z,$$

where Z is an n -by- n orthogonal/unitary matrix and R is an m -by- m upper triangular matrix.

See ?larz that applies an elementary reflector returned by ?tzzrf to a general matrix.

Input Parameters

m	INTEGER. The number of rows in the matrix A ($m \geq 0$).
n	INTEGER. The number of columns in A ($n \geq m$).
$a, work$	REAL for stzzrf DOUBLE PRECISION for dtzzrf COMPLEX for ctzzrf DOUBLE COMPLEX for ztzzrf. Arrays: $a(lda,*)$, $work(lwork)$. The leading m -by- n upper trapezoidal part of the array a contains the matrix A to be factorized.

The second dimension of a must be at least $\max(1, n)$.
 $work$ is a workspace array, its dimension $\max(1, lwork)$.
 lda INTEGER. The first dimension of a ; at least $\max(1, m)$.
 $lwork$ INTEGER. The size of the $work$ array;
 $lwork \geq \max(1, m)$.
 If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).
 See *Application Notes* for the suggested value of $lwork$.

Output Parameters

a Overwritten on exit by the factorization data as follows:
 the leading m -by- m upper triangular part of a contains the upper triangular matrix R , and elements $m+1$ to n of the first m rows of a , with the array tau , represent the orthogonal matrix Z as a product of m elementary reflectors.

tau REAL for `stzrzf`
 DOUBLE PRECISION for `dtzrzf`
 COMPLEX for `ctzrzf`
 DOUBLE COMPLEX for `ztzrzf`.
 Array, DIMENSION at least $\max(1, m)$. Contains scalar factors of the elementary reflectors for the matrix Z .

$work(1)$ If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

$info$ INTEGER.
 If $info = 0$, the execution is successful.
 If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tzrzf` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (m,n) .
<i>tau</i>	Holds the vector of length (m) .

Application Notes

For better performance, try using `lwork = m*blocksize`, where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of `lwork` for the first run, or set `lwork = -1`.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If `lwork = -1`, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if `lwork` is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Related routines include:

<code>?ormrz</code>	to apply matrix <i>Q</i> (for real matrices)
<code>?unmrz</code>	to apply matrix <i>Q</i> (for complex matrices).

?ormrz

Multiplies a real matrix by the orthogonal matrix defined from the factorization formed by ?tzzrf.

Syntax

FORTRAN 77:

```
call sormrz(side, trans, m, n, k, l, a, lda, tau, c, ldc, work, lwork, info)
call dormrz(side, trans, m, n, k, l, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call ormrz(a, tau, c, l [, side] [,trans] [,info])
```


Description

The routine multiplies a real m -by- n matrix c by Q or Q^T , where Q is the real orthogonal matrix defined as a product of k elementary reflectors $H(i)$ of order n : $Q = H(1) * H(2) * \dots * H(k)$ as returned by the factorization routine `stzrzf/dtzrzf`.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products $Q * C$, $Q^T * C$, $C * Q$, or $C * Q^T$ (overwriting the result over c).

The matrix Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

Input Parameters

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^T is applied to c from the left. If <i>side</i> = 'R', Q or Q^T is applied to c from the right.
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'T'. If <i>trans</i> = 'N', the routine multiplies c by Q . If <i>trans</i> = 'T', the routine multiplies c by Q^T .
<i>m</i>	INTEGER. The number of rows in the matrix c ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in c ($n \geq 0$).
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: $0 \leq k \leq m$, if <i>side</i> = 'L'; $0 \leq k \leq n$, if <i>side</i> = 'R'.
<i>l</i>	INTEGER. The number of columns of the matrix A containing the meaningful part of the Householder reflectors. Constraints: $0 \leq l \leq m$, if <i>side</i> = 'L'; $0 \leq l \leq n$, if <i>side</i> = 'R'.
<i>a, tau, c, work</i>	REAL for <code>sormrz</code> DOUBLE PRECISION for <code>dormrz</code> . Arrays: $a(lda,*)$, $tau(*)$, $c(ldc,*)$. On entry, the i th row of a must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by <code>stzrzf/dtzrzf</code> in the last k rows of its array argument a .

The second dimension of *a* must be at least $\max(1, m)$ if *side* = 'L', and at least $\max(1, n)$ if *side* = 'R'.
tau(*i*) must contain the scalar factor of the elementary reflector $H(i)$, as returned by `stzrzf/dtzrzf`.
The dimension of *tau* must be at least $\max(1, k)$.
c(*ldc*,*) contains the *m*-by-*n* matrix *C*.
The second dimension of *c* must be at least $\max(1, n)$.
work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The first dimension of *a*; $lda \geq \max(1, k)$.

ldc INTEGER. The first dimension of *c*; $ldc \geq \max(1, m)$.

lwork INTEGER. The size of the *work* array. Constraints:
 $lwork \geq \max(1, n)$ if *side* = 'L';
 $lwork \geq \max(1, m)$ if *side* = 'R'.
If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by `xerbla`.
See *Application Notes* for the suggested value of *lwork*.

Output Parameters

c Overwritten by the product Q^*C , $Q^T * C$, C^*Q , or C^*Q^T (as specified by *side* and *trans*).

work(1) If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ormrz` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (k, m) .
<i>tau</i>	Holds the vector of length (k) .
<i>c</i>	Holds the matrix <i>C</i> of size (m, n) .
<i>side</i>	Must be 'L' or 'R'. The default value is 'L'.
<i>trans</i>	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using $lwork = n * blocksize$ (if *side* = 'L') or $lwork = m * blocksize$ (if *side* = 'R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is [?unmrz](#).

[?unmrz](#)

Multiplies a complex matrix by the unitary matrix defined from the factorization formed by [?tzrzf](#).

Syntax

FORTRAN 77:

```
call cunmrz(side, trans, m, n, k, l, a, lda, tau, c, ldc, work, lwork, info)
call zunmrz(side, trans, m, n, k, l, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call unmrz(a, tau, c, l [,side] [,trans] [,info])
```

Description

The routine multiplies a complex m -by- n matrix C by Q or Q^H , where Q is the unitary matrix defined as a product of k elementary reflectors $H(i)$:

$Q = H(1)H^* H(2)H^* \dots H(k)H$ as returned by the factorization routine [ctzrzf/ztrzf](#).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products Q^*C , $Q^H C$, C^*Q , or C^*Q^H (overwriting the result over C).

The matrix Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

Input Parameters

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^H is applied to C from the left. If <i>side</i> = 'R', Q or Q^H is applied to C from the right.
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'C'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'C', the routine multiplies C by Q^H .
<i>m</i>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: $0 \leq k \leq m$, if <i>side</i> = 'L'; $0 \leq k \leq n$, if <i>side</i> = 'R'.
<i>l</i>	INTEGER. The number of columns of the matrix A containing the meaningful part of the Householder reflectors. Constraints: $0 \leq l \leq m$, if <i>side</i> = 'L'; $0 \leq l \leq n$, if <i>side</i> = 'R'.
<i>a, tau, c, work</i>	COMPLEX for cunmrz DOUBLE COMPLEX for zunmrz. Arrays: $a(lda,*)$, $tau(*)$, $c ldc,*)$, $work(lwork)$.

On entry, the i th row of a must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by `ctzrzf/ztzrzf` in the last k rows of its array argument a .

The second dimension of a must be at least $\max(1, m)$ if $side = 'L'$, and at least $\max(1, n)$ if $side = 'R'$.

$\tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by `ctzrzf/ztzrzf`.

The dimension of τ must be at least $\max(1, k)$.

$c(ldc,*)$ contains the m -by- n matrix C .

The second dimension of c must be at least $\max(1, n)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The first dimension of a ; $lda \geq \max(1, k)$.

ldc INTEGER. The first dimension of c ; $ldc \geq \max(1, m)$.

$lwork$ INTEGER. The size of the $work$ array. Constraints:

$lwork \geq \max(1, n)$ if $side = 'L'$;

$lwork \geq \max(1, m)$ if $side = 'R'$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).

See *Application Notes* for the suggested value of $lwork$.

Output Parameters

c Overwritten by the product Q^*C , $Q^H * C$, C^*Q , or C^*Q^H (as specified by $side$ and $trans$).

$work(1)$ If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

$info$ INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `unmrz` interface are the following:

<code>a</code>	Holds the matrix A of size (k, m) .
<code>tau</code>	Holds the vector of length (k) .
<code>c</code>	Holds the matrix C of size (m, n) .
<code>side</code>	Must be 'L' or 'R'. The default value is 'L'.
<code>trans</code>	Must be 'N' or 'C'. The default value is 'N'.

Application Notes

For better performance, try using `lwork = n*blocksize` (if `side = 'L'`) or `lwork = m*blocksize` (if `side = 'R'`) where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of `lwork` for the first run, or set `lwork = -1`.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If `lwork = -1`, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if `lwork` is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is [?ormrz](#).

?ggqrf

Computes the generalized QR factorization of two matrices.

Syntax

FORTRAN 77:

```
call sggqrf(n, m, p, a, lda, taua, b, ldb, taub, work, lwork, info)
call dggqrf(n, m, p, a, lda, taua, b, ldb, taub, work, lwork, info)
call cggqrf(n, m, p, a, lda, taua, b, ldb, taub, work, lwork, info)
call zggqrf(n, m, p, a, lda, taua, b, ldb, taub, work, lwork, info)
```

Fortran 95:

```
call ggqrf(a, b [,taua] [,taub] [,info])
```

Description

The routine forms the generalized QR factorization of an n -by- m matrix A and an n -by- p matrix B as $A = Q^*R$, $B = Q^*T^*Z$, where Q is an n -by- n orthogonal/unitary matrix, Z is a p -by- p orthogonal/unitary matrix, and R and T assume one of the forms:

$$R = \begin{matrix} & m \\ & m \\ n - m & \begin{pmatrix} R_{11} \\ 0 \end{pmatrix} \end{matrix}, \quad \text{if } n \geq m$$

or

$$R = \begin{matrix} n & m - n \\ n & \begin{pmatrix} R_{11} & R_{12} \end{pmatrix} \end{matrix}, \quad \text{if } n < m$$

where R_{11} is upper triangular, and

$$T = \begin{matrix} & p-n & n \\ n & \begin{pmatrix} 0 & T_{12} \end{pmatrix} \end{matrix}, \text{ if } n \leq p,$$

$$T = \begin{matrix} & p \\ n-p & \begin{pmatrix} T_{11} \\ T_{21} \end{pmatrix} \end{matrix}, \text{ if } n > p,$$

where T_{12} or T_{21} is a p -by- p upper triangular matrix.

In particular, if B is square and nonsingular, the GQR factorization of A and B implicitly gives the QR factorization of $B^{-1}A$ as:

$$B^{-1} * A = Z^H * (T^{-1} * R)$$

Input Parameters

n	INTEGER. The number of rows of the matrices A and B ($n \geq 0$).
m	INTEGER. The number of columns in A ($m \geq 0$).
p	INTEGER. The number of columns in B ($p \geq 0$).
$a, b, work$	REAL for sggqrf DOUBLE PRECISION for dggqrf COMPLEX for cggqrf DOUBLE COMPLEX for zggqrf. Arrays: $a(lda,*)$ contains the matrix A . The second dimension of a must be at least $\max(1, m)$. $b(l db,*)$ contains the matrix B . The second dimension of b must be at least $\max(1, p)$. $work$ is a workspace array, its dimension $\max(1, lwork)$.
lda	INTEGER. The first dimension of a ; at least $\max(1, n)$.
ldb	INTEGER. The first dimension of b ; at least $\max(1, n)$.
$lwork$	INTEGER. The size of the $work$ array; must be at least $\max(1, n, m, p)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#). See *Application Notes* for the suggested value of *lwork*.

Output Parameters

<i>a, b</i>	Overwritten by the factorization data as follows: on exit, the elements on and above the diagonal of the array <i>a</i> contain the $\min(n,m)$-by-<i>m</i> upper trapezoidal matrix R (R is upper triangular if $n \geq m$); the elements below the diagonal, with the array <i>taua</i>, represent the orthogonal/unitary matrix Q as a product of $\min(n,m)$ elementary reflectors ; if $n \leq p$, the upper triangle of the subarray <i>b</i>(1:<i>n</i>, <i>p</i>-<i>n</i>+1:<i>p</i>) contains the <i>n</i>-by-<i>n</i> upper triangular matrix T; if $n > p$, the elements on and above the (<i>n</i>-<i>p</i>)th subdiagonal contain the <i>n</i>-by-<i>p</i> upper trapezoidal matrix T; the remaining elements, with the array <i>taub</i>, represent the orthogonal/unitary matrix Z as a product of elementary reflectors.
<i>taua, taub</i>	REAL for sggqrf DOUBLE PRECISION for dggqrf COMPLEX for cggqrf DOUBLE COMPLEX for zggqrf. Arrays, DIMENSION at least $\max(1, \min(n, m))$ for <i>taua</i> and at least $\max(1, \min(n, p))$ for <i>taub</i>. The array <i>taua</i> contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Q. The array <i>taub</i> contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Z.
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i>(1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ggqrf` interface are the following:

<code>a</code>	Holds the matrix A of size (n, m) .
<code>b</code>	Holds the matrix B of size (n, p) .
<code>taua</code>	Holds the vector of length $\min(n, m)$.
<code>taub</code>	Holds the vector of length $\min(n, p)$.

Application Notes

For better performance, try using $lwork \geq \max(n, m, p) * \max(nb1, nb2, nb3)$, where $nb1$ is the optimal blocksize for the QR factorization of an n -by- m matrix, $nb2$ is the optimal blocksize for the RQ factorization of an n -by- p matrix, and $nb3$ is the optimal blocksize for a call of `?ormqr/?unmqr`.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?ggrqf

Computes the generalized RQ factorization of two matrices.

Syntax

FORTRAN 77:

```

call sggrqf (m, p, n, a, lda, taua, b, ldb, taub, work, lwork, info)
call dggrqf (m, p, n, a, lda, taua, b, ldb, taub, work, lwork, info)
call cggrqf (m, p, n, a, lda, taua, b, ldb, taub, work, lwork, info)
call zggrqf (m, p, n, a, lda, taua, b, ldb, taub, work, lwork, info)

```

Fortran 95:

```

call ggrqf(a, b [,taua] [,taub] [,info])

```

Description

The routine forms the generalized RQ factorization of an m -by- n matrix A and an p -by- n matrix B as $A = R^*Q$, $B = Z^*T^*Q$, where Q is an n -by- n orthogonal/unitary matrix, Z is a p -by- p orthogonal/unitary matrix, and R and T assume one of the forms:

$$R = \begin{matrix} & n-m & m \\ m & \begin{pmatrix} 0 & R_{12} \end{pmatrix} \end{matrix}, \quad \text{if } m \leq n,$$

or

$$R = \begin{matrix} & n \\ m-n & \begin{pmatrix} R_{11} \\ R_{21} \end{pmatrix} \\ n & \end{matrix}, \quad \text{if } m > n,$$

where R_{11} or R_{21} is upper triangular, and

$$T = \begin{pmatrix} n & \\ & p - n \end{pmatrix} \begin{pmatrix} T_{11} \\ 0 \end{pmatrix}, \quad \text{if } p \geq n,$$

or

$$T = \begin{pmatrix} p & n - p \\ & \end{pmatrix} \begin{pmatrix} T_{11} & T_{12} \end{pmatrix}, \quad \text{if } p < n,$$

where T_{11} is upper triangular.

In particular, if B is square and nonsingular, the *GRQ* factorization of A and B implicitly gives the *RQ* factorization of A^*B^{-1} as:

$$A^*B^{-1} = (R^*T^{-1})^*Z^H$$

Input Parameters

m	INTEGER. The number of rows of the matrix A ($m \geq 0$).
p	INTEGER. The number of rows in B ($p \geq 0$).
n	INTEGER. The number of columns of the matrices A and B ($n \geq 0$).
$a, b, work$	REAL for sggrqf DOUBLE PRECISION for dggrqf COMPLEX for cggrqf DOUBLE COMPLEX for zggrqf. Arrays: $a(lda,*)$ contains the m -by- n matrix A . The second dimension of a must be at least $\max(1, n)$. $b(l db,*)$ contains the p -by- n matrix B . The second dimension of b must be at least $\max(1, n)$. $work$ is a workspace array, its dimension $\max(1, lwork)$.
lda	INTEGER. The first dimension of a ; at least $\max(1, m)$.

ldb INTEGER. The first dimension of *b*; at least $\max(1, p)$.

lwork INTEGER. The size of the *work* array; must be at least $\max(1, n, m, p)$.
 If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#).
 See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a, b Overwritten by the factorization data as follows:
 on exit, if $m \leq n$, the upper triangle of the subarray *a*(1:*m*, *n-m+1*:*n*) contains the *m*-by-*m* upper triangular matrix *R*;
 if $m > n$, the elements on and above the (*m-n*)th subdiagonal contain the *m*-by-*n* upper trapezoidal matrix *R*;
 the remaining elements, with the array *taua*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors; the elements on and above the diagonal of the array *b* contain the $\min(p, n)$ -by-*n* upper trapezoidal matrix *T* (*T* is upper triangular if $p \geq n$); the elements below the diagonal, with the array *taub*, represent the orthogonal/unitary matrix *Z* as a product of elementary reflectors.

taua, taub REAL for sggrqf
 DOUBLE PRECISION for dggrqf
 COMPLEX for cggrqf
 DOUBLE COMPLEX for zggrqf.
 Arrays, DIMENSION at least $\max(1, \min(m, n))$ for *taua* and at least $\max(1, \min(p, n))$ for *taub*.
 The array *taua* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Q*.
 The array *taub* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Z*.

work(1) If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ggrqf` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>m</i> , <i>n</i>).
<i>b</i>	Holds the matrix <i>A</i> of size (<i>p</i> , <i>n</i>).
<i>taua</i>	Holds the vector of length min(<i>m</i> , <i>n</i>).
<i>taub</i>	Holds the vector of length min(<i>p</i> , <i>n</i>).

Application Notes

For better performance, try using

$$lwork \geq \max(n, m, p) * \max(nb1, nb2, nb3),$$

where *nb1* is the optimal blocksize for the *RQ* factorization of an *m*-by-*n* matrix, *nb2* is the optimal blocksize for the *QR* factorization of an *p*-by-*n* matrix, and *nb3* is the optimal blocksize for a call of `?ormrq/?unmrq`.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork*= -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork*= -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Singular Value Decomposition

This section describes LAPACK routines for computing the *singular value decomposition* (SVD) of a general m -by- n matrix A :

$$A = U\Sigma V^H.$$

In this decomposition, U and V are unitary (for complex A) or orthogonal (for real A); Σ is an m -by- n diagonal matrix with real diagonal elements σ_i :

$$\sigma_1 < \sigma_2 < \dots < \sigma_{\min(m, n)} < 0.$$

The diagonal elements σ_i are *singular values* of A . The first $\min(m, n)$ columns of the matrices U and V are, respectively, *left* and *right singular vectors* of A . The singular values and singular vectors satisfy

$$AV_i = \sigma_i U_i \text{ and } A^H U_i = \sigma_i V_i$$

where U_i and V_i are the i -th columns of U and V , respectively.

To find the SVD of a general matrix A , call the LAPACK routine `?gebrd` or `?gbbbrd` for reducing A to a bidiagonal matrix B by a unitary (orthogonal) transformation: $A = QB P^H$. Then call `?bdsqr`, which forms the SVD of a bidiagonal matrix: $B = U_1 \Sigma V_1^H$.

Thus, the sought-for SVD of A is given by $A = U\Sigma V^H = (QU_1)\Sigma(V_1^H P^H)$.

Table 4-2 lists LAPACK routines (Fortran 77 interface) that perform singular value decomposition of matrices. Respective routine names in Fortran 95 interface are without the first symbol (see [Routine Naming Conventions](#)).

Table 4-2 Computational Routines for Singular Value Decomposition (SVD)

Operation	Real matrices	Complex matrices
Reduce A to a bidiagonal matrix B : $A = QB P^H$ (full storage)	<code>?gebrd</code>	<code>?gebrd</code>
Reduce A to a bidiagonal matrix B : $A = QB P^H$ (band storage)	<code>?gbbbrd</code>	<code>?gbbbrd</code>
Generate the orthogonal (unitary) matrix Q or P	<code>?orgbr</code>	<code>?ungbr</code>
Apply the orthogonal (unitary) matrix Q or P	<code>?ormbr</code>	<code>?unmbr</code>

Operation	Real matrices	Complex matrices
Form singular value decomposition of the bidiagonal matrix B : $B = U\Sigma V^H$	?bdsqr ?bdsdc	?bdsqr

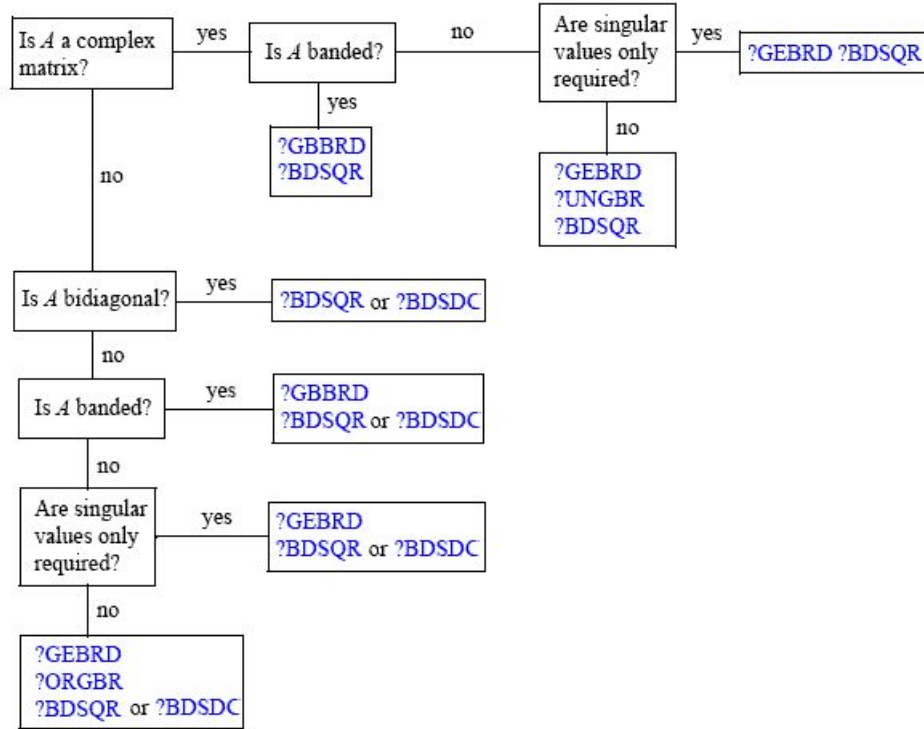
Figure 4-1 Decision Tree: Singular Value Decomposition

Figure 4-1 “Decision Tree: Singular Value Decomposition” presents a decision tree that helps you choose the right sequence of routines for SVD, depending on whether you need singular values only or singular vectors as well, whether A is real or complex, and so on.

You can use the SVD to find a minimum-norm solution to a (possibly) rank-deficient least squares problem of minimizing $\|Ax - b\|^2$. The effective rank k of the matrix A can be determined as the number of singular values which exceed a suitable threshold. The minimum-norm solution is

$$x = V_k (\Sigma_k)^{-1} c$$

where Σ_k is the leading k -by- k submatrix of Σ , the matrix V_k consists of the first k columns of $V = PV_1$, and the vector c consists of the first k elements of $U^H b = U_1^H Q^H b$.

?gebrd

Reduces a general matrix to bidiagonal form.

Syntax

FORTRAN 77:

```
call sgebrd(m, n, a, lda, d, e, tauq, taup, work, lwork, info)
call dgebrd(m, n, a, lda, d, e, tauq, taup, work, lwork, info)
call cgebrd(m, n, a, lda, d, e, tauq, taup, work, lwork, info)
call zgebrd(m, n, a, lda, d, e, tauq, taup, work, lwork, info)
```

Fortran 95:

```
call gebrd(a [, d] [,e] [,tauq] [,taup] [,info])
```

Description

The routine reduces a general m -by- n matrix A to a bidiagonal matrix B by an orthogonal (unitary) transformation.

$$A = QB P^H = Q \begin{pmatrix} B_1 \\ 0 \end{pmatrix} P^H = Q_1 B_1 P^H ,$$

If $m \geq n$, the reduction is given by

where B_1 is an n -by- n upper diagonal matrix, Q and P are orthogonal or, for a complex A , unitary matrices; Q_1 consists of the first n columns of Q .

If $m < n$, the reduction is given by

$$A = Q^* B^* P^H = Q^* (B_1 0) P^H = Q_1^* B_1 P_1^H ,$$

where B_1 is an m -by- m lower diagonal matrix, Q and P are orthogonal or, for a complex A , unitary matrices; P_1 consists of the first m rows of P .

The routine does not form the matrices Q and P explicitly, but represents them as products of elementary reflectors. Routines are provided to work with the matrices Q and P in this representation:

If the matrix A is real,

- to compute Q and P explicitly, call [?orgbr](#).

- to multiply a general matrix by Q or P , call `?ormbr`.

If the matrix A is complex,

- to compute Q and P explicitly, call `?ungbr`.
- to multiply a general matrix by Q or P , call `?unmbr`.

Input Parameters

m INTEGER. The number of rows in the matrix A ($m \geq 0$).

n INTEGER. The number of columns in A ($n \geq 0$).

a, *work* REAL for `sgebrd`
 DOUBLE PRECISION for `dgebrd`
 COMPLEX for `cgebrd`
 DOUBLE COMPLEX for `zgebrd`.

Arrays:
a(*lda*,*) contains the matrix A .
 The second dimension of *a* must be at least $\max(1, n)$.
work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The first dimension of *a*; at least $\max(1, m)$.

lwork INTEGER.
 The dimension of *work*; at least $\max(1, m, n)$.
 If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by `xerbla`.
 See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a If $m \geq n$, the diagonal and first super-diagonal of *a* are overwritten by the upper bidiagonal matrix B . Elements below the diagonal are overwritten by details of Q , and the remaining elements are overwritten by details of P .

	If $m < n$, the diagonal and first sub-diagonal of a are overwritten by the lower bidiagonal matrix B. Elements above the diagonal are overwritten by details of P, and the remaining elements are overwritten by details of Q.
d	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Array, DIMENSION at least $\max(1, \min(m, n))$. Contains the diagonal elements of B.
e	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Array, DIMENSION at least $\max(1, \min(m, n) - 1)$. Contains the off-diagonal elements of B.
τ_{uq}, τ_{up}	REAL for sgebrd DOUBLE PRECISION for dgebrd COMPLEX for cgebrd DOUBLE COMPLEX for zgebrd. Arrays, DIMENSION at least $\max(1, \min(m, n))$. Contain further details of the matrices Q and P.
$work(1)$	If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gebrd` interface are the following:

a	Holds the matrix A of size (m, n) .
d	Holds the vector of length $\min(m, n)$.
e	Holds the vector of length $\min(m, n) - 1$.
τ_{uq}	Holds the vector of length $\min(m, n)$.
τ_{up}	Holds the vector of length $\min(m, n)$.

Application Notes

For better performance, try using $lwork = (m + n) * blocksize$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrices *Q*, *B*, and *P* satisfy $QB P^H = A + E$, where $\|E\|_2 = c(n)\epsilon \|A\|_2$, $c(n)$ is a modestly increasing function of *n*, and ϵ is the machine precision.

The approximate number of floating-point operations for real flavors is

$$(4/3) * n^2 * (3 * m - n) \text{ for } m \geq n,$$

$$(4/3) * m^2 * (3 * n - m) \text{ for } m < n.$$

The number of operations for complex flavors is four times greater.

If *n* is much less than *m*, it can be more efficient to first form the *QR* factorization of *A* by calling [?geqrf](#) and then reduce the factor *R* to bidiagonal form. This requires approximately $2 * n^2 * (m + n)$ floating-point operations.

If *m* is much less than *n*, it can be more efficient to first form the *LQ* factorization of *A* by calling [?gelqf](#) and then reduce the factor *L* to bidiagonal form. This requires approximately $2 * m^2 * (m + n)$ floating-point operations.

?gbbbrd

Reduces a general band matrix to bidiagonal form.

Syntax

FORTRAN 77:

```
call sgbbrd(vect, m, n, ncc, kl, ku, ab, ldab, d, e, q, ldq, pt, ldpt, c, ldc,
work, info)

call dgbbrd(vect, m, n, ncc, kl, ku, ab, ldab, d, e, q, ldq, pt, ldpt, c, ldc,
work, info)

call cgbbrd(vect, m, n, ncc, kl, ku, ab, ldab, d, e, q, ldq, pt, ldpt, c, ldc,
work, rwork, info)

call zgbbrd(vect, m, n, ncc, kl, ku, ab, ldab, d, e, q, ldq, pt, ldpt, c, ldc,
work, rwork, info)
```

Fortran 95:

```
call gbbbrd(a [, c] [,d] [,e] [,q] [,pt] [,kl] [,m] [,info])
```

Description

This routine reduces an m -by- n band matrix A to upper bidiagonal matrix B : $A = Q * B * P^H$. Here the matrices Q and P are orthogonal (for real A) or unitary (for complex A). They are determined as products of Givens rotation matrices, and may be formed explicitly by the routine if required. The routine can also update a matrix C as follows: $C = Q^H * C$.

Input Parameters

<i>vect</i>	CHARACTER*1. Must be 'N' or 'Q' or 'P' or 'B'. If <i>vect</i> = 'N', neither Q nor P^H is generated. If <i>vect</i> = 'Q', the routine generates the matrix Q . If <i>vect</i> = 'P', the routine generates the matrix P^H . If <i>vect</i> = 'B', the routine generates both Q and P^H .
<i>m</i>	INTEGER. The number of rows in the matrix A ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in A ($n \geq 0$).
<i>ncc</i>	INTEGER. The number of columns in C ($ncc \geq 0$).

<i>kl</i>	INTEGER. The number of sub-diagonals within the band of <i>A</i> ($kl \geq 0$).
<i>ku</i>	INTEGER. The number of super-diagonals within the band of <i>A</i> ($ku \geq 0$).
<i>ab, c, work</i>	REAL for <i>sgbbrd</i> DOUBLE PRECISION for <i>dgbbrd</i> COMPLEX for <i>cgbbrd</i> DOUBLE COMPLEX for <i>zgbbrd</i> . Arrays: <i>ab(ldab,*)</i> contains the matrix <i>A</i> in band storage (see Matrix Storage Schemes). The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>c(ldc,*)</i> contains an <i>m</i> -by- <i>ncc</i> matrix <i>C</i> . If <i>ncc</i> = 0, the array <i>c</i> is not referenced. The second dimension of <i>c</i> must be at least $\max(1, ncc)$. <i>work(*)</i> is a workspace array. The dimension of <i>work</i> must be at least $2 \cdot \max(m, n)$ for real flavors, or $\max(m, n)$ for complex flavors.
<i>ldab</i>	INTEGER. The first dimension of the array <i>ab</i> ($ldab \geq kl + ku + 1$).
<i>ldq</i>	INTEGER. The first dimension of the output array <i>q</i> . $ldq \geq \max(1, m)$ if <i>vect</i> = 'Q' or 'B', $ldq \geq 1$ otherwise.
<i>ldpt</i>	INTEGER. The first dimension of the output array <i>pt</i> . $ldpt \geq \max(1, n)$ if <i>vect</i> = 'P' or 'B', $ldpt \geq 1$ otherwise.
<i>ldc</i>	INTEGER. The first dimension of the array <i>c</i> . $ldc \geq \max(1, m)$ if <i>ncc</i> > 0; $ldc \geq 1$ if <i>ncc</i> = 0.
<i>rwork</i>	REAL for <i>cgbbrd</i> DOUBLE PRECISION for <i>zgbbrd</i> . A workspace array, DIMENSION at least $\max(m, n)$.

Output Parameters

<i>ab</i>	Overwritten by values generated during the reduction.
<i>d</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors.

	Array, <code>DIMENSION</code> at least $\max(1, \min(m, n))$. Contains the diagonal elements of the matrix B .
e	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Array, <code>DIMENSION</code> at least $\max(1, \min(m, n) - 1)$. Contains the off-diagonal elements of B .
q, pt	REAL for sgebrd DOUBLE PRECISION for dgebrd COMPLEX for cgebrd DOUBLE COMPLEX for zgebrd. Arrays: $q(ldq, *)$ contains the output m -by- m matrix Q . The second dimension of q must be at least $\max(1, m)$. $p(ldpt, *)$ contains the output n -by- n matrix P^T . The second dimension of pt must be at least $\max(1, n)$.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gbbrd` interface are the following:

a	Stands for argument ab in Fortran 77 interface . Holds the array A of size $(kl+ku+1, n)$.
c	Holds the matrix C of size (m, ncc) .
d	Holds the vector of length $\min(m, n)$.
e	Holds the vector of length $\min(m, n)-1$.
q	Holds the matrix Q of size (m, m) .
pt	Holds the matrix PT of size (n, n) .
m	If omitted, assumed $m = n$.
kl	If omitted, assumed $kl = ku$.
ku	Restored as $ku = lda - kl - 1$.

`vect`

Restored based on the presence of arguments q and pt as follows:

`vect` = 'B', if both q and pt are present,

`vect` = 'Q', if q is present and pt omitted, `vect` = 'P',

if q is omitted and pt present, `vect` = 'N', if both q and pt are omitted.

Application Notes

The computed matrices Q , B , and P satisfy $Q^*B^*P^H = A + E$, where $\|E\|_2 = c(n)\varepsilon \|A\|_2$, $c(n)$ is a modestly increasing function of n , and ε is the machine precision.

If $m = n$, the total number of floating-point operations for real flavors is approximately the sum of:

$6n^2(kl + ku)$ if `vect` = 'N' and $ncc = 0$,

$3n^2ncc(kl + ku - 1)/(kl + ku)$ if C is updated, and

$3n^3(kl + ku - 1)/(kl + ku)$ if either Q or P^H is generated (double this if both).

To estimate the number of operations for complex flavors, use the same formulas with the coefficients 20 and 10 (instead of 6 and 3).

?orgbr

Generates the real orthogonal matrix Q or P^T determined by ?gebrd.

Syntax

FORTRAN 77:

```
call sorgbr(vect, m, n, k, a, lda, tau, work, lwork, info)
```

```
call dorgbr(vect, m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call orgbr(a, tau [,vect] [,info])
```

Description

The routine generates the whole or part of the orthogonal matrices Q and P_T formed by the routines [sgebrd](#)/[dgebrd](#). Use this routine after a call to [sgebrd](#)/[dgebrd](#). All valid combinations of arguments are described in *Input parameters*. In most cases you need the following:

To compute the whole m -by- m matrix Q :

```
call ?orgbr('Q', m, m, n, a ... )
```

(note that the array a must have at least m columns).

To form the n leading columns of Q if $m > n$:

```
call ?orgbr('Q', m, n, n, a ... )
```

To compute the whole n -by- n matrix P^T :

```
call ?orgbr('P', n, n, m, a ... )
```

(note that the array a must have at least n rows).

To form the m leading rows of P^T if $m < n$:

```
call ?orgbr('P', m, n, m, a ... )
```

Input Parameters

<i>vect</i>	CHARACTER*1. Must be 'Q' or 'P'. If <i>vect</i> = 'Q', the routine generates the matrix Q . If <i>vect</i> = 'P', the routine generates the matrix P^T .
<i>m</i>	INTEGER. The number of required rows of Q or P^T .
<i>n</i>	INTEGER. The number of required columns of Q or P^T .
<i>k</i>	INTEGER. One of the dimensions of A in ?gebrd: If <i>vect</i> = 'Q', the number of columns in A ; If <i>vect</i> = 'P', the number of rows in A . Constraints: $m \geq 0$, $n \geq 0$, $k \geq 0$. For <i>vect</i> = 'Q': $k \leq n \leq m$ if $m > k$, or $m = n$ if $m \leq k$. For <i>vect</i> = 'P': $k \leq m \leq n$ if $n > k$, or $m = n$ if $n \leq k$.
<i>a, work</i>	REAL for sorgbr DOUBLE PRECISION for dorgbr. Arrays: <i>a(lda,*)</i> is the array a as returned by ?gebrd. The second dimension of a must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The first dimension of a ; at least $\max(1, m)$.
<i>tau</i>	REAL for sorgbr DOUBLE PRECISION for dorgbr.

For $\text{vect} = 'Q'$, the array tauq as returned by `?gebrd`.
 For $\text{vect} = 'P'$, the array taup as returned by `?gebrd`.
 The dimension of tau must be at least $\max(1, \min(m, k))$
 for $\text{vect} = 'Q'$, or $\max(1, \min(m, k))$ for $\text{vect} = 'P'$.
 lwork INTEGER. The size of the work array.
 If $\text{lwork} = -1$, then a workspace query is assumed; the
 routine only calculates the optimal size of the work array,
 returns this value as the first entry of the work array, and
 no error message related to lwork is issued by `xerbla`.
 See *Application Notes* for the suggested value of lwork .

Output Parameters

a Overwritten by the orthogonal matrix Q or P^T (or the leading
 rows or columns thereof) as specified by vect , m , and n .
 $\text{work}(1)$ If $\text{info} = 0$, on exit $\text{work}(1)$ contains the minimum value
 of lwork required for optimum performance. Use this lwork
 for subsequent runs.
 info INTEGER.
 If $\text{info} = 0$, the execution is successful.
 If $\text{info} = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `orgbr` interface are the following:

a Holds the matrix A of size (m, n) .
 tau Holds the vector of length $\min(m, k)$ where
 $k = m$, if $\text{vect} = 'P'$,
 $k = n$, if $\text{vect} = 'Q'$.
 vect Must be $'Q'$ or $'P'$. The default value is $'Q'$.

Application Notes

For better performance, try using $lwork = \min(m, n) * blocksize$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix *Q* differs from an exactly orthogonal matrix by a matrix *E* such that $\|E\|_2 = O(\epsilon)$.

The approximate numbers of floating-point operations for the cases listed in *Description* are as follows:

To form the whole of *Q*:

$$(4/3) * n * (3m^2 - 3m * n + n^2) \text{ if } m > n;$$

$$(4/3) * m^3 \text{ if } m \leq n.$$

To form the *n* leading columns of *Q* when $m > n$:

$$(2/3) * n^2 * (3m - n^2) \text{ if } m > n.$$

To form the whole of P^T :

$$(4/3) * n^3 \text{ if } m \geq n;$$

$$(4/3) * m * (3n^2 - 3m * n + m^2) \text{ if } m < n.$$

To form the *m* leading columns of P^T when $m < n$:

$$(2/3) * n^2 * (3m - n^2) \text{ if } m > n.$$

The complex counterpart of this routine is [?ungbr](#).

[?ormbr](#)

Multiplies an arbitrary real matrix by the real orthogonal matrix Q or P^T determined by [?gebrd](#).

Syntax

FORTRAN 77:

```
call sormbr(vect, side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork,
info)

call dormbr(vect, side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork,
info)
```

Fortran 95:

```
call ormbr(a, tau, c [,vect] [,side] [,trans] [,info])
```

Description

Given an arbitrary real matrix C , this routine forms one of the matrix products Q^*C , $Q^T C$, C^*Q , C^*Q^T , P^*C , $P^T C$, C^*P , C^*P^T , where Q and P are orthogonal matrices computed by a call to [sgebrd](#)/[dgebrd](#). The routine overwrites the product on C .

Input Parameters

In the descriptions below, r denotes the order of Q or P^T :

If $side = 'L'$, $r = m$; if $side = 'R'$, $r = n$.

$vect$	CHARACTER*1. Must be 'Q' or 'P'. If $vect = 'Q'$, then Q or Q^T is applied to C . If $vect = 'P'$, then P or P_T is applied to C .
$side$	CHARACTER*1. Must be 'L' or 'R'. If $side = 'L'$, multipliers are applied to C from the left. If $side = 'R'$, they are applied to C from the right.
$trans$	CHARACTER*1. Must be 'N' or 'T'. If $trans = 'N'$, then Q or P is applied to C . If $trans = 'T'$, then Q^T or P^T is applied to C .
m	INTEGER. The number of rows in C .

<i>n</i>	INTEGER. The number of columns in <i>C</i> .
<i>k</i>	INTEGER. One of the dimensions of <i>A</i> in ?gebrd: If <i>vect</i> = 'Q', the number of columns in <i>A</i> ; If <i>vect</i> = 'P', the number of rows in <i>A</i> . Constraints: $m \geq 0, n \geq 0, k \geq 0$.
<i>a, c, work</i>	REAL for sormbr DOUBLE PRECISION for dormbr. Arrays: <i>a(lda,*)</i> is the array <i>a</i> as returned by ?gebrd. Its second dimension must be at least $\max(1, \min(r, k))$ for <i>vect</i> = 'Q', or $\max(1, r)$ for <i>vect</i> = 'P'. <i>c ldc,*)</i> holds the matrix <i>C</i> . Its second dimension must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> . Constraints: $lda \geq \max(1, r)$ if <i>vect</i> = 'Q'; $lda \geq \max(1, \min(r, k))$ if <i>vect</i> = 'P'.
<i>ldc</i>	INTEGER. The first dimension of <i>c</i> ; $ldc \geq \max(1, m)$.
<i>tau</i>	REAL for sormbr DOUBLE PRECISION for dormbr. Array, DIMENSION at least $\max(1, \min(r, k))$. For <i>vect</i> = 'Q', the array <i>tauq</i> as returned by ?gebrd. For <i>vect</i> = 'P', the array <i>taup</i> as returned by ?gebrd.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array. Constraints: $lwork \geq \max(1, n)$ if <i>side</i> = 'L'; $lwork \geq \max(1, m)$ if <i>side</i> = 'R'. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla. See <i>Application Notes</i> for the suggested value of <i>lwork</i> .

Output Parameters

<i>c</i>	Overwritten by the product $Q^*C, Q^{T*}C, C^*Q, C^*Q^T, P^*C, P^{T*}C, C^*P$, or C^*P^T , as specified by <i>vect</i> , <i>side</i> , and <i>trans</i> .

<code>work(1)</code>	If <code>info = 0</code> , on exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance. Use this <code>lwork</code> for subsequent runs.
<code>info</code>	INTEGER. If <code>info = 0</code> , the execution is successful. If <code>info = -i</code> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ormbr` interface are the following:

<code>a</code>	Holds the matrix <i>A</i> of size $(r, \min(nq, k))$ where $r = nq$, if <code>vect = 'Q'</code> , $r = \min(nq, k)$, if <code>vect = 'P'</code> , $nq = m$, if <code>side = 'L'</code> , $nq = n$, if <code>side = 'R'</code> , $k = m$, if <code>vect = 'P'</code> , $k = n$, if <code>vect = 'Q'</code> .
<code>tau</code>	Holds the vector of length $\min(nq, k)$.
<code>c</code>	Holds the matrix <i>C</i> of size (m, n) .
<code>vect</code>	Must be 'Q' or 'P'. The default value is 'Q'.
<code>side</code>	Must be 'L' or 'R'. The default value is 'L'.
<code>trans</code>	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using

`lwork = n*blocksize` for `side = 'L'`, or

`lwork = m*blocksize` for `side = 'R'`,

where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed product differs from the exact product by a matrix *E* such that $\|E\|_2 = O(\epsilon) * \|C\|_2$.

The total number of floating-point operations is approximately

$2*n*k(2*m - k)$ if *side* = 'L' and $m \geq k$;

$2*m*k(2*n - k)$ if *side* = 'R' and $n \geq k$;

$2*m^2*n$ if *side* = 'L' and $m < k$;

$2*n^2*m$ if *side* = 'R' and $n < k$.

The complex counterpart of this routine is [?unmbr](#).

?ungbr

Generates the complex unitary matrix Q or P^H determined by ?gebrd.

Syntax

FORTRAN 77:

```
call cungrb(vect, m, n, k, a, lda, tau, work, lwork, info)
call zungrb(vect, m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call ungbr(a, tau [,vect] [,info])
```


Description

The routine generates the whole or part of the unitary matrices Q and P_H formed by the routines [cgebrd/zgebrd](#). Use this routine after a call to [cgebrd/zgebrd](#). All valid combinations of arguments are described in *Input Parameters*; in most cases you need the following:

To compute the whole m -by- m matrix Q , use:

```
call ?ungbr('Q', m, m, n, a ... )
```

(note that the array a must have at least m columns).

To form the n leading columns of Q if $m > n$, use:

```
call ?ungbr('Q', m, n, n, a ... )
```

To compute the whole n -by- n matrix P^H , use:

```
call ?ungbr('P', n, n, m, a ... )
```

(note that the array a must have at least n rows).

To form the m leading rows of P_H if $m < n$, use:

```
call ?ungbr('P', m, n, m, a ... )
```

Input Parameters

$vect$	CHARACTER*1. Must be 'Q' or 'P'. If $vect = 'Q'$, the routine generates the matrix Q . If $vect = 'P'$, the routine generates the matrix P^H .
m	INTEGER. The number of required rows of Q or P^H .
n	INTEGER. The number of required columns of Q or P^H .
k	INTEGER. One of the dimensions of A in ?gebrd : If $vect = 'Q'$, the number of columns in A ; If $vect = 'P'$, the number of rows in A . Constraints: $m \geq 0, n \geq 0, k \geq 0$. For $vect = 'Q'$: $k \leq n \leq m$ if $m > k$, or $m = n$ if $m \leq k$. For $vect = 'P'$: $k \leq m \leq n$ if $n > k$, or $m = n$ if $n \leq k$.
$a, work$	COMPLEX for cungbr DOUBLE COMPLEX for zungbr . Arrays: $a(lda,*)$ is the array a as returned by ?gebrd . The second dimension of a must be at least $\max(1, n)$.

work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The first dimension of *a*; at least $\max(1, m)$.

tau COMPLEX for *cungbr*
DOUBLE COMPLEX for *zungbr*.
For *vect* = 'Q', the array *tauq* as returned by ?gebrd.
For *vect* = 'P', the array *taup* as returned by ?gebrd.
The dimension of *tau* must be at least $\max(1, \min(m, k))$ for *vect* = 'Q', or $\max(1, \min(m, k))$ for *vect* = 'P'.

lwork INTEGER. The size of the *work* array.
Constraint: $lwork < \max(1, \min(m, n))$.
If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by *xerbla*.
See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a Overwritten by the orthogonal matrix *Q* or P^T (or the leading rows or columns thereof) as specified by *vect*, *m*, and *n*.

work(1) If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *ungbr* interface are the following:

a Holds the matrix *A* of size (m, n) .

tau Holds the vector of length $\min(m, k)$ where $k = m$, if *vect* = 'P',

$k = n$, if $vect = 'Q'$.
 $vect$ Must be 'Q' or 'P'. The default value is 'Q'.

Application Notes

For better performance, try using $lwork = \min(m, n) * blocksize$, where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of $lwork$ for the first run, or set $lwork = -1$.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if $lwork$ is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix Q differs from an exactly orthogonal matrix by a matrix E such that $\|E\|_2 = O(\epsilon)$.

The approximate numbers of possible floating-point operations are listed below:

To compute the whole matrix Q :

$$(16/3)n(3m^2 - 3m*n + n^2) \text{ if } m > n;$$

$$(16/3)m^3 \text{ if } m \leq n.$$

To form the n leading columns of Q when $m > n$:

$$(8/3)n^2(3m - n^2).$$

To compute the whole matrix P^H :

$$(16/3)n^3 \text{ if } m \geq n;$$

$$(16/3)m(3n^2 - 3m*n + m^2) \text{ if } m < n.$$

To form the m leading columns of P^H when $m < n$:

$(8/3)n^2(3m - n^2)$ if $m > n$.

The real counterpart of this routine is [?orgbr](#).

?unmbr

Multiplies an arbitrary complex matrix by the unitary matrix Q or P determined by ?gebrd.

Syntax

FORTRAN 77:

```
call cunmbr(vect, side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork,
info)

call zunmbr(vect, side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork,
info)
```

Fortran 95:

```
call unmbr(a, tau, c [,vect] [,side] [,trans] [,info])
```

Description

Given an arbitrary complex matrix C , this routine forms one of the matrix products Q^*C , Q^H*C , $C*Q$, $C*Q^H$, P^*C , P^H*C , $C*P$, or $C*P^H$, where Q and P are unitary matrices computed by a call to [cgebrd](#)/[zgebrd](#). The routine overwrites the product on C .

Input Parameters

In the descriptions below, r denotes the order of Q or P^H :

If $side = 'L'$, $r = m$; if $side = 'R'$, $r = n$.

<i>vect</i>	CHARACTER*1. Must be 'Q' or 'P'. If <i>vect</i> = 'Q', then Q or Q^H is applied to C . If <i>vect</i> = 'P', then P or P^H is applied to C .
<i>side</i>	CHARACTER*1. Must be 'L' or 'R'. If <i>side</i> = 'L', multipliers are applied to C from the left. If <i>side</i> = 'R', they are applied to C from the right.
<i>trans</i>	CHARACTER*1. Must be 'N' or 'C'. If <i>trans</i> = 'N', then Q or P is applied to C . If <i>trans</i> = 'C', then Q^H or P^H is applied to C .

m INTEGER. The number of rows in C .
 n INTEGER. The number of columns in C .
 k INTEGER. One of the dimensions of A in ?gebrd:
 If $vect = 'Q'$, the number of columns in A ;
 If $vect = 'P'$, the number of rows in A .
 Constraints: $m \geq 0, n \geq 0, k \geq 0$.
 $a, c, work$ COMPLEX for cunmbr
 DOUBLE COMPLEX for zunmbr.
 Arrays:
 $a(lda,*)$ is the array a as returned by ?gebrd.
 Its second dimension must be at least $\max(1, \min(r, k))$ for
 $vect = 'Q'$, or $\max(1, r)$ for $vect = 'P'$.
 $c(ldc,*)$ holds the matrix C .
 Its second dimension must be at least $\max(1, n)$.
 $work$ is a workspace array, its dimension $\max(1,$
 $lwork)$.
 lda INTEGER. The first dimension of a . Constraints:
 $lda \geq \max(1, r)$ if $vect = 'Q'$;
 $lda \geq \max(1, \min(r, k))$ if $vect = 'P'$.
 ldc INTEGER. The first dimension of c ; $ldc \geq \max(1, m)$.
 tau COMPLEX for cunmbr
 DOUBLE COMPLEX for zunmbr.
 Array, DIMENSION at least $\max(1, \min(r, k))$.
 For $vect = 'Q'$, the array $tauq$ as returned by ?gebrd.
 For $vect = 'P'$, the array $taup$ as returned by ?gebrd.
 $lwork$ INTEGER. The size of the $work$ array.
 $lwork \geq \max(1, n)$ if $side = 'L'$;
 $lwork \geq \max(1, m)$ if $side = 'R'$.
 $lwork \geq 1$ if $n=0$ or $m=0$.
 For optimum performance $lwork \geq \max(1, n*nb)$ if $side$
 $= 'L'$, and $lwork \geq \max(1, m*nb)$ if $side = 'R'$, where
 nb is the optimal blocksize. ($nb = 0$ if $m = 0$ or $n =$
 0 .)

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#). See *Application Notes* for the suggested value of $lwork$.

Output Parameters

c	Overwritten by the product $Q^H C$, $Q^H C$, $C^H Q$, $C^H Q$, $P^H C$, $P^H C$, $C^H P$, or $C^H P$, as specified by $vect$, $side$, and $trans$.
$work(1)$	If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `unmbr` interface are the following:

a	Holds the matrix A of size $(r, \min(nq, k))$ where $r = nq$, if $vect = 'Q'$, $r = \min(nq, k)$, if $vect = 'P'$, $nq = m$, if $side = 'L'$, $nq = n$, if $side = 'R'$, $k = m$, if $vect = 'P'$, $k = n$, if $vect = 'Q'$.
τ	Holds the vector of length $\min(nq, k)$.
c	Holds the matrix C of size (m, n) .
$vect$	Must be 'Q' or 'P'. The default value is 'Q'.
$side$	Must be 'L' or 'R'. The default value is 'L'.
$trans$	Must be 'N' or 'C'. The default value is 'N'.

Application Notes

For better performance, use

$lwork = n * blocksize$ for $side = 'L'$, or

$lwork = m * blocksize$ for $side = 'R'$,

where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of $lwork$ for the first run, or set $lwork = -1$.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if $lwork$ is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed product differs from the exact product by a matrix E such that $\|E\|_2 = O(\epsilon) * \|C\|_2$.

The total number of floating-point operations is approximately

$8 * n * k (2 * m - k)$ if $side = 'L'$ and $m \geq k$;

$8 * m * k (2 * n - k)$ if $side = 'R'$ and $n \geq k$;

$8 * m^2 * n$ if $side = 'L'$ and $m < k$;

$8 * n^2 * m$ if $side = 'R'$ and $n < k$.

The real counterpart of this routine is [?ormbr](#).

?bdsqr

Computes the singular value decomposition of a general matrix that has been reduced to bidiagonal form.

Syntax

FORTRAN 77:

```
call sbdsqr(uplo, n, ncv, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc, work,
info)

call dbdsqr(uplo, n, ncv, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc, work,
info)

call cbdsqr(uplo, n, ncv, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc, work,
info)

call zbdsqr(uplo, n, ncv, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc, work,
info)
```

Fortran 95:

```
call rbdsqr(d, e [,vt] [,u] [,c] [,uplo] [,info])
call bdsqr(d, e [,vt] [,u] [,c] [,uplo] [,info])
```

Description

This routine computes the singular values and, optionally, the right and/or left singular vectors from the [Singular Value Decomposition](#) (SVD) of a real n -by- n (upper or lower) bidiagonal matrix B using the implicit zero-shift QR algorithm. The SVD of B has the form $B = Q * S * P^H$ where S is the diagonal matrix of singular values, Q is an orthogonal matrix of left singular vectors, and P is an orthogonal matrix of right singular vectors. If left singular vectors are requested, this subroutine actually returns $U * Q$ instead of Q , and, if right singular vectors are requested, this subroutine returns $P_H * VT$ instead of P_H , for given real/complex input matrices U and VT . When U and VT are the orthogonal/unitary matrices that reduce a general matrix A to bidiagonal form: $A = U * B * VT$, as computed by ?gebrd, then

$$A = (U * Q) * S * (P^H * VT)$$

is the SVD of A . Optionally, the subroutine may also compute $Q_H * C$ for a given real/complex input matrix C .

See also [?lasq1](#), [?lasq2](#), [?lasq3](#), [?lasq4](#), [?lasq5](#), [?lasq6](#) used by this routine.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>B</i> is an upper bidiagonal matrix. If <i>uplo</i> = 'L', <i>B</i> is a lower bidiagonal matrix.
<i>n</i>	INTEGER. The order of the matrix <i>B</i> ($n \geq 0$).
<i>ncvt</i>	INTEGER. The number of columns of the matrix <i>VT</i>, that is, the number of right singular vectors ($ncvt \geq 0$). Set <i>ncvt</i> = 0 if no right singular vectors are required.
<i>nru</i>	INTEGER. The number of rows in <i>U</i>, that is, the number of left singular vectors ($nru \geq 0$). Set <i>nru</i> = 0 if no left singular vectors are required.
<i>ncc</i>	INTEGER. The number of columns in the matrix <i>C</i> used for computing the product $Q^H * C$ ($ncc \geq 0$). Set <i>ncc</i> = 0 if no matrix <i>C</i> is supplied.
<i>d</i> , <i>e</i> , <i>work</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays: <i>d</i>(*) contains the diagonal elements of <i>B</i>. The dimension of <i>d</i> must be at least $\max(1, n)$. <i>e</i>(*) contains the $(n-1)$ off-diagonal elements of <i>B</i>. The dimension of <i>e</i> must be at least $\max(1, n)$. <i>e</i>(<i>n</i>) is used for workspace. <i>work</i>(*) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, 2*n)$ if $ncvt = nru = ncc = 0$; $\max(1, 4*(n-1))$ otherwise.
<i>vt</i> , <i>u</i> , <i>c</i>	REAL for sbdsqr DOUBLE PRECISION for dbdsqr COMPLEX for cbdsqr DOUBLE COMPLEX for zbdsqr. Arrays: <i>vt</i>(<i>ldvt</i>,*) contains an <i>n</i>-by-<i>ncvt</i> matrix <i>VT</i>. The second dimension of <i>vt</i> must be at least $\max(1, ncvt)$.

vt is not referenced if $ncvt = 0$.
 $u(ldu, *)$ contains an nru by n unit matrix U .
 The second dimension of u must be at least $\max(1, n)$.
 u is not referenced if $nru = 0$.
 $c ldc, *)$ contains the matrix C for computing the product $Q^H * C$.
 The second dimension of c must be at least $\max(1, ncc)$.
 The array is not referenced if $ncc = 0$.

$ldvt$ INTEGER. The first dimension of vt . Constraints:
 $ldvt \geq \max(1, n)$ if $ncvt > 0$;
 $ldvt \geq 1$ if $ncvt = 0$.

ldu INTEGER. The first dimension of u . Constraint:
 $ldu \geq \max(1, nru)$.

ldc INTEGER. The first dimension of c . Constraints:
 $ldc \geq \max(1, n)$ if $ncc > 0$; $ldc \geq 1$ otherwise.

Output Parameters

d On exit, if $info = 0$, overwritten by the singular values in decreasing order (see $info$).

e On exit, if $info = 0$, e is destroyed. See also $info$ below.

c Overwritten by the product $Q^H * C$.

vt On exit, this array is overwritten by $P^H * VT$.

u On exit, this array is overwritten by $U * Q$.

$info$ INTEGER.
 If $info = 0$, the execution is successful.
 If $info = -i$, the i -th parameter had an illegal value.
 If $info = i$, the algorithm failed to converge; i specifies how many off-diagonals did not converge.
 In this case, d and e contain on exit the diagonal and off-diagonal elements, respectively, of a bidiagonal matrix orthogonally equivalent to B .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `bdsqr` interface are the following:

<code>d</code>	Holds the vector of length (n) .
<code>e</code>	Holds the vector of length (n) .
<code>vt</code>	Holds the matrix VT of size (n, ncv) .
<code>u</code>	Holds the matrix U of size (nru, n) .
<code>c</code>	Holds the matrix C of size (n, ncc) .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.
<code>ncv</code>	If argument <code>vt</code> is present, then <code>ncv</code> is equal to the number of columns in matrix VT ; otherwise, <code>ncv</code> is set to zero.
<code>nru</code>	If argument <code>u</code> is present, then <code>nru</code> is equal to the number of rows in matrix U ; otherwise, <code>nru</code> is set to zero.
<code>ncc</code>	If argument <code>c</code> is present, then <code>ncc</code> is equal to the number of columns in matrix C ; otherwise, <code>ncc</code> is set to zero.

Note that two variants of Fortran 95 interface for `bdsqr` routine are needed because of an ambiguous choice between real and complex cases appear when `vt`, `u`, and `c` are omitted. Thus, the name `rbdsqr` is used in real cases (single or double precision), and the name `bdsqr` is used in complex cases (single or double precision).

Application Notes

Each singular value and singular vector is computed to high relative accuracy. However, the reduction to bidiagonal form (prior to calling the routine) may decrease the relative accuracy in the small singular values of the original matrix if its singular values vary widely in magnitude.

If s_i is an exact singular value of B , and s_i is the corresponding computed value, then

$$|s_i - \sigma_i| \leq p(m, n) * \epsilon * \sigma_i$$

where $p(m, n)$ is a modestly increasing function of m and n , and ϵ is the machine precision.

If only singular values are computed, they are computed more accurately than when some singular vectors are also computed (that is, the function $p(m, n)$ is smaller).

If u_i is the corresponding exact left singular vector of B , and w_i is the corresponding computed left singular vector, then the angle $\theta(u_i, w_i)$ between them is bounded as follows:

$$\theta(u_i, w_i) \leq p(m, n) * \epsilon / \min_{i \neq j} (|\sigma_i - \sigma_j| / |\sigma_i + \sigma_j|).$$

Here $\min_{i \neq j} (|\sigma_i - \sigma_j| / |\sigma_i + \sigma_j|)$ is the *relative gap* between σ_i and the other singular values. A similar error bound holds for the right singular vectors.

The total number of real floating-point operations is roughly proportional to n^2 if only the singular values are computed. About $6n^2 * nru$ additional operations ($12n^2 * nru$ for complex flavors) are required to compute the left singular vectors and about $6n^2 * ncvt$ operations ($12n^2 * ncvt$ for complex flavors) to compute the right singular vectors.

?bdsdc

Computes the singular value decomposition of a real bidiagonal matrix using a divide and conquer method.

Syntax

FORTRAN 77:

```
call sbdsdc(uplo, compq, n, d, e, u, ldu, vt, ldvt, q, iq, work, iwork, info)
call dbdsdc(uplo, compq, n, d, e, u, ldu, vt, ldvt, q, iq, work, iwork, info)
```

Fortran 95:

```
call bdsdc(d, e [,u] [,vt] [,q] [,iq] [,uplo] [,info])
```

Description

This routine computes the [Singular Value Decomposition](#) (SVD) of a real n -by- n (upper or lower) bidiagonal matrix B : $B = U * \Sigma * V^T$, using a divide and conquer method, where Σ is a diagonal matrix with non-negative diagonal elements (the singular values of B), and U and V are orthogonal matrices of left and right singular vectors, respectively. `?bdsdc` can be used to compute all singular values, and optionally, singular vectors or singular vectors in compact form.

This routine uses `?lasd0`, `?lasd1`, `?lasd2`, `?lasd3`, `?lasd4`, `?lasd5`, `?lasd6`, `?lasd7`, `?lasd8`, `?lasd9`, `?lasda`, `?lasdq`, `?lasdt`.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>B</i> is an upper bidiagonal matrix. If <i>uplo</i> = 'L', <i>B</i> is a lower bidiagonal matrix.
<i>compq</i>	CHARACTER*1. Must be 'N', 'P', or 'I'. If <i>compq</i> = 'N', compute singular values only. If <i>compq</i> = 'P', compute singular values and compute singular vectors in compact form. If <i>compq</i> = 'I', compute singular values and singular vectors.
<i>n</i>	INTEGER. The order of the matrix <i>B</i> ($n \geq 0$).
<i>d</i> , <i>e</i> , <i>work</i>	REAL for sbdsdc DOUBLE PRECISION for dbdsdc. Arrays: <i>d</i>(*) contains the <i>n</i> diagonal elements of the bidiagonal matrix <i>B</i>. The dimension of <i>d</i> must be at least $\max(1, n)$. <i>e</i>(*) contains the off-diagonal elements of the bidiagonal matrix <i>B</i>. The dimension of <i>e</i> must be at least $\max(1, n)$. <i>work</i>(*) is a workspace array. The dimension of <i>work</i> must be at least: $\max(1, 4*n)$, if <i>compq</i> = 'N'; $\max(1, 6*n)$, if <i>compq</i> = 'P'; $\max(1, 3*n^2+4*n)$, if <i>compq</i> = 'I'.
<i>ldu</i>	INTEGER. The first dimension of the output array <i>u</i>; $ldu \geq 1$. If singular vectors are desired, then $ldu \geq \max(1, n)$.
<i>ldvt</i>	INTEGER. The first dimension of the output array <i>vt</i>; $ldvt \geq 1$. If singular vectors are desired, then $ldvt \geq \max(1, n)$.
<i>iwork</i>	INTEGER. Workspace array, dimension at least $\max(1, 8*n)$.

Output Parameters

<i>d</i>	If <i>info</i> = 0, overwritten by the singular values of <i>B</i> .
<i>e</i>	On exit, <i>e</i> is overwritten.
<i>u</i> , <i>vt</i> , <i>q</i>	REAL for sbdsdc DOUBLE PRECISION for dbdsdc. Arrays: <i>u</i>(<i>ldu</i>,*), <i>vt</i>(<i>ldvt</i>,*), <i>q</i>(*). If <i>compq</i> = 'I', then on exit <i>u</i> contains the left singular vectors of the bidiagonal matrix <i>B</i>, unless <i>info</i> ≠ 0 (see <i>info</i>). For other values of <i>compq</i>, <i>u</i> is not referenced. The second dimension of <i>u</i> must be at least max(1,<i>n</i>). If <i>compq</i> = 'I', then on exit <i>vt</i> contains the right singular vectors of the bidiagonal matrix <i>B</i>, unless <i>info</i> ≠ 0 (see <i>info</i>). For other values of <i>compq</i>, <i>vt</i> is not referenced. The second dimension of <i>vt</i> must be at least max(1,<i>n</i>). If <i>compq</i> = 'P', then on exit, if <i>info</i> = 0, <i>q</i> and <i>iq</i> contain the left and right singular vectors in a compact form. Specifically, <i>q</i> contains all the REAL (for sbdsdc) or DOUBLE PRECISION (for dbdsdc) data for singular vectors. For other values of <i>compq</i>, <i>q</i> is not referenced. See <i>Application notes</i> for details.
<i>iq</i>	INTEGER. Array: <i>iq</i>(*). If <i>compq</i> = 'P', then on exit, if <i>info</i> = 0, <i>q</i> and <i>iq</i> contain the left and right singular vectors in a compact form. Specifically, <i>iq</i> contains all the INTEGER data for singular vectors. For other values of <i>compq</i>, <i>iq</i> is not referenced. See <i>Application notes</i> for details.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value. If <i>info</i> = <i>i</i>, the algorithm failed to compute a singular value. The update process of divide and conquer failed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `bdsdc` interface are the following:

<code>d</code>	Holds the vector of length (n) .
<code>e</code>	Holds the vector of length (n) .
<code>u</code>	Holds the matrix U of size (n, n) .
<code>vt</code>	Holds the matrix VT of size (n, n) .
<code>q</code>	Holds the vector of length (ldq) , where $ldq \geq n * (11 + 2 * smlsiz + 8 * \text{int}(\log_2(n / (smlsiz + 1))))$ and <code>smlsiz</code> is returned by <code>ilaenv</code> and is equal to the maximum size of the subproblems at the bottom of the computation tree (usually about 25).
<code>compq</code>	Restored based on the presence of arguments <code>u</code> , <code>vt</code> , <code>q</code> , and <code>iq</code> as follows: <code>compq</code> = 'N', if none of <code>u</code> , <code>vt</code> , <code>q</code> , and <code>iq</code> are present, <code>compq</code> = 'I', if both <code>u</code> and <code>vt</code> are present. Arguments <code>u</code> and <code>vt</code> must either be both present or both omitted, <code>compq</code> = 'P', if both <code>q</code> and <code>iq</code> are present. Arguments <code>q</code> and <code>iq</code> must either be both present or both omitted. Note that there will be an error condition if all of <code>u</code> , <code>vt</code> , <code>q</code> , and <code>iq</code> arguments are present simultaneously.

Symmetric Eigenvalue Problems

Symmetric eigenvalue problems are posed as follows: given an n -by- n real symmetric or complex Hermitian matrix A , find the *eigenvalues* λ and the corresponding *eigenvectors* z that satisfy the equation

$$Az = \lambda z \text{ (or, equivalently, } z^H A = \lambda z^H \text{)}.$$

In such eigenvalue problems, all n eigenvalues are real not only for real symmetric but also for complex Hermitian matrices A , and there exists an orthonormal system of n eigenvectors. If A is a symmetric or Hermitian positive-definite matrix, all eigenvalues are positive.

To solve a symmetric eigenvalue problem with LAPACK, you usually need to reduce the matrix to tridiagonal form and then solve the eigenvalue problem with the tridiagonal matrix obtained. LAPACK includes routines for reducing the matrix to a tridiagonal form by an orthogonal (or unitary) similarity transformation $A = QTQ^H$ as well as for solving tridiagonal symmetric eigenvalue problems. These routines (for Fortran 77 interface) are listed in [Table 4-3](#). Respective routine names in Fortran 95 interface are without the first symbol (see [Routine Naming Conventions](#)).

There are different routines for symmetric eigenvalue problems, depending on whether you need all eigenvectors or only some of them or eigenvalues only, whether the matrix A is positive-definite or not, and so on.

These routines are based on three primary algorithms for computing eigenvalues and eigenvectors of symmetric problems: the divide and conquer algorithm, the QR algorithm, and bisection followed by inverse iteration. The divide and conquer algorithm is generally more efficient and is recommended for computing all eigenvalues and eigenvectors. Furthermore, to solve an eigenvalue problem using the divide and conquer algorithm, you need to call only one routine. In general, more than one routine has to be called if the QR algorithm or bisection followed by inverse iteration is used.

Decision tree in [Figure 4-2](#) will help you choose the right routine or sequence of routines for eigenvalue problems with real symmetric matrices. A similar decision tree for complex Hermitian matrices is presented in [Figure 4-3](#).

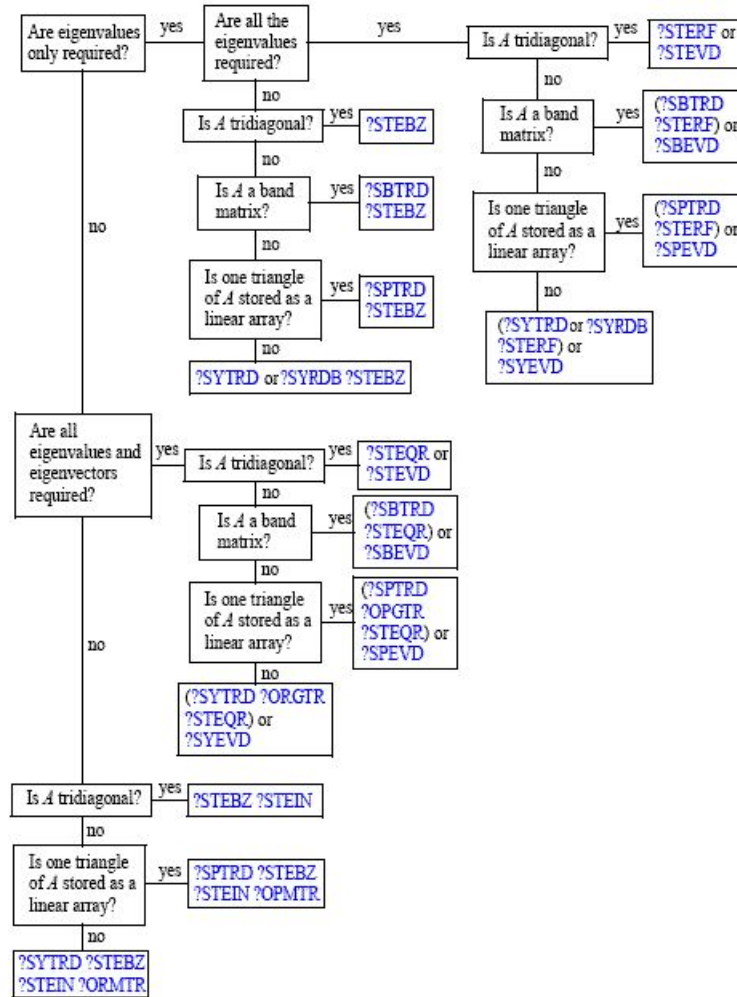
Figure 4-2 Decision Tree: Real Symmetric Eigenvalue Problems

Figure 4-3 Decision Tree: Complex Hermitian Eigenvalue Problems

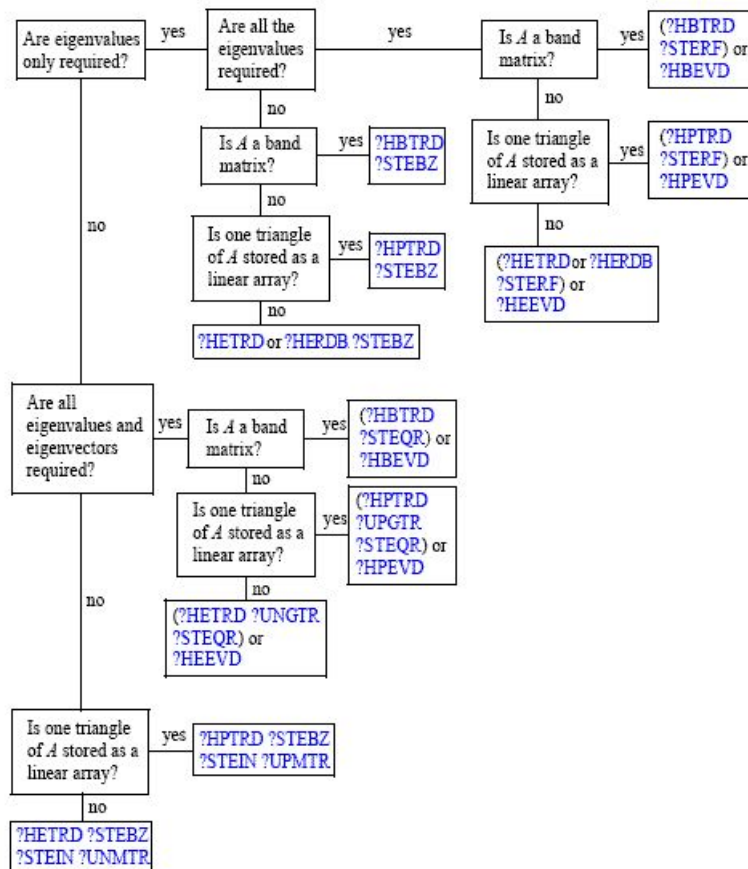


Table 4-3 Computational Routines for Solving Symmetric Eigenvalue Problems

Operation	Real symmetric matrices	Complex Hermitian matrices
Reduce to tridiagonal form $A = QTQ^H$ (full storage)	?sytrd ?syddb	?hetrd ?herdb
Reduce to tridiagonal form $A = QTQ^H$ (packed storage)	?sptrd	?hptrd

Operation	Real symmetric matrices	Complex Hermitian matrices
Reduce to tridiagonal form $A = QTQ^H$ (band storage).	?sbtrd	?hbtrd
Generate matrix Q (full storage)	?orgtr	?ungtr
Generate matrix Q (packed storage)	?opgtr	?upgtr
Apply matrix Q (full storage)	?ormtr	?unmtr
Apply matrix Q (packed storage)	?opmtr	?upmtr
Find all eigenvalues of a tridiagonal matrix T	?sterf	
Find all eigenvalues and eigenvectors of a tridiagonal matrix T	?stegr ?stedc	?stegr ?stedc
Find all eigenvalues and eigenvectors of a tridiagonal positive-definite matrix T .	?ptegr	?ptegr
Find selected eigenvalues of a tridiagonal matrix T	?stebz ?stegr	?stegr
Find selected eigenvectors of a tridiagonal matrix T	?stein ?stegr	?stein ?stegr
Find selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix T	?stemr	?stemr
Compute the reciprocal condition numbers for the eigenvectors	?disna	?disna

?sytrd

Reduces a real symmetric matrix to tridiagonal form.

Syntax

FORTRAN 77:

```
call ssytrd(uplo, n, a, lda, d, e, tau, work, lwork, info)
call dsytrd(uplo, n, a, lda, d, e, tau, work, lwork, info)
```

Fortran 95:

```
call sytrd(a, tau [,uplo] [,info])
```

Description

This routine reduces a real symmetric matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation: $A = Q^* T Q^T$. The orthogonal matrix Q is not formed explicitly but is represented as a product of $n-1$ elementary reflectors. Routines are provided for working with Q in this representation (see *Application Notes* below).

This routine calls [?latrd](#) to reduce a real symmetric matrix to tridiagonal form by an orthogonal similarity transformation.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).
<i>a, work</i>	REAL for ssytrd DOUBLE PRECISION for dsytrd. <i>a</i> (<i>lda</i> ,*) is an array containing either upper or lower triangular part of the matrix <i>A</i> , as specified by <i>uplo</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; at least $\max(1, n)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array ($lwork \geq n$).

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#). See *Application Notes* for the suggested value of *lwork*.

Output Parameters

<i>a</i>	Overwritten by the tridiagonal matrix <i>T</i> and details of the orthogonal matrix <i>Q</i> , as specified by <i>uplo</i> .
<i>d</i> , <i>e</i> , <i>tau</i>	REAL for ssytrd DOUBLE PRECISION for dsytrd. Arrays: <i>d</i> (*) contains the diagonal elements of the matrix <i>T</i> . The dimension of <i>d</i> must be at least $\max(1, n)$. <i>e</i> (*) contains the off-diagonal elements of <i>T</i> . The dimension of <i>e</i> must be at least $\max(1, n-1)$. <i>tau</i> (*) stores further details of the orthogonal matrix <i>Q</i> in the first $n-1$ elements. <i>tau</i> (<i>n</i>) is used as workspace. The dimension of <i>tau</i> must be at least $\max(1, n)$.
<i>work</i> (1)	If <i>info</i> =0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *sytrd* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (n, n) .
<i>tau</i>	Holds the vector of length $(n-1)$.
<i>d</i>	Holds the vector of length (n) .
<i>e</i>	Holds the vector of length $(n-1)$.

`uplo` Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For better performance, try using `lwork = n*blocksize`, where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of `lwork` for the first run, or set `lwork = -1`.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If `lwork = -1`, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if `lwork` is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix T is exactly similar to a matrix $A+E$, where $\|E\|_2 = c(n) * \epsilon * \|A\|_2$, $c(n)$ is a modestly increasing function of n , and ϵ is the machine precision.

The approximate number of floating-point operations is $(4/3)n^3$.

After calling this routine, you can call the following:

`?orgtr` to form the computed matrix Q explicitly
`?ormtr` to multiply a real matrix by Q .

The complex counterpart of this routine is `?hetrd`.

?syldb

Reduces a real symmetric matrix to tridiagonal form with Successive Bandwidth Reduction approach.

Syntax

FORTRAN 77:

```
call ssyldb(jobz, uplo, n, kd, a, lda, d, e, tau, z, ldz, work, lwork, info)
call dsyldb(jobz, uplo, n, kd, a, lda, d, e, tau, z, ldz, work, lwork, info)
```

Description

This routine reduces a real symmetric matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation: $A = Q^* T^* Q^T$ and optionally multiplies matrix Z by Q , or simply forms Q .

This routine reduces a full symmetric matrix to the banded symmetric form, and then to the tridiagonal symmetric form with a Successive Bandwidth Reduction approach after Prof. C.Bischof's works (see for instance, [Bischof92]). ?syldb is functionally close to ?sytrd routine but the tridiagonal form may differ from those obtained by ?sytrd. Unlike ?sytrd, the orthogonal matrix Q cannot be restored from the details of matrix A on exit.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only A is reduced to T . If <i>jobz</i> = 'V', then A is reduced to T and A contains Q on exit. If <i>jobz</i> = 'U', then A is reduced to T and Z contains Z^*Q on exit.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', a stores the upper triangular part of A . If <i>uplo</i> = 'L', a stores the lower triangular part of A .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>kd</i>	INTEGER. The bandwidth of the banded matrix B ($kd \geq 1$).
<i>a, z, work</i>	REAL for ssyldb.

DOUBLE PRECISION for dsyrdb.
 $a(lda,*)$ is an array containing either upper or lower triangular part of the matrix A , as specified by $uplo$.
The second dimension of a must be at least $\max(1, n)$.
 $z(ldz,*)$, the second dimension of z must be at least $\max(1, n)$.
If $jobz = 'U'$, then the matrix z is multiplied by Q .
If $jobz = 'N'$ or $'V'$, then z is not referenced.
 $work(lwork)$ is a workspace array.

lda INTEGER. The first dimension of a ; at least $\max(1, n)$.
ldz INTEGER. The first dimension of z ; at least $\max(1, n)$. Not referenced if $jobz = 'N'$.

lwork INTEGER. The size of the $work$ array ($lwork \geq (2kd+1)n+kd$).
If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by xerbla.
See *Application Notes* for the suggested value of $lwork$.

Output Parameters

a If $jobz = 'V'$, then overwritten by Q matrix.
If $jobz = 'N'$ or $'U'$, then overwritten by the banded matrix B and details of the orthogonal matrix Q_B to reduce A to B as specified by $uplo$.

 z On exit,
if $jobz = 'U'$, then the matrix z is overwritten by z^*Q .
If $jobz = 'N'$ or $'V'$, then z is not referenced.

 d, e, tau DOUBLE PRECISION.
Arrays:
 $d(*)$ contains the diagonal elements of the matrix T .
The dimension of d must be at least $\max(1, n)$.
 $e(*)$ contains the off-diagonal elements of T .
The dimension of e must be at least $\max(1, n-1)$.
 $tau(*)$ stores further details of the orthogonal matrix Q .
The dimension of tau must be at least $\max(1, n-kd-1)$.

work(1) If *info*=0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

For better performance, try using $lwork = n*(3*kd+3)$.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

For better performance, try using *kd* equal to 40 if $n \leq 2000$ and 64 otherwise.

Try using ?syrdB instead of ?sytrd on large matrices obtaining only eigenvalues - when no eigenvectors are needed, especially in multi-threaded environment. ?syrdB becomes faster beginning approximately with $n = 1000$, and much faster at larger matrices with a better scalability than ?sytrd.

Avoid applying ?syrdB for computing eigenvectors due to the two-step reduction, that is, the number of operations needed to apply orthogonal transformations to *z* is doubled compared to the traditional one-step reduction. In that case it is better to apply ?sytrd and ?ormtr/?orgtr to obtain tridiagonal form along with the orthogonal transformation matrix *Q*.

?herdb

Reduces a complex Hermitian matrix to tridiagonal form with Successive Bandwidth Reduction approach.

Syntax

FORTRAN 77:

```
call cherdb(jobz, uplo, n, kd, a, lda, d, e, tau, z, ldz, work, lwork, info)
call zherdb(jobz, uplo, n, kd, a, lda, d, e, tau, z, ldz, work, lwork, info)
```

Description

This routine reduces a complex Hermitian matrix A to symmetric tridiagonal form T by a unitary similarity transformation: $A = Q^* T Q^T$ and optionally multiplies matrix Z by Q , or simply forms Q .

This routine reduces a full Hermitian matrix to the banded Hermitian form, and then to the tridiagonal symmetric form with a Successive Bandwidth Reduction approach after Prof. C.Bischof's works (see for instance, [Bischof92]). ?herdb is functionally close to ?hetrd routine but the tridiagonal form may differ from those obtained by ?hetrd. Unlike ?hetrd, the orthogonal matrix Q cannot be restored from the details of matrix A on exit.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only A is reduced to T . If <i>jobz</i> = 'V', then A is reduced to T and A contains Q on exit. If <i>jobz</i> = 'U', then A is reduced to T and Z contains $Z^* Q$ on exit.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', a stores the upper triangular part of A . If <i>uplo</i> = 'L', a stores the lower triangular part of A .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>kd</i>	INTEGER. The bandwidth of the banded matrix B ($kd \geq 1$).
<i>a, z, work</i>	COMPLEX for cherdb.

DOUBLE COMPLEX for zherdb.
 $a(lda,*)$ is an array containing either upper or lower triangular part of the matrix A , as specified by $uplo$.
 The second dimension of a must be at least $\max(1, n)$.
 $z(ldz,*)$, the second dimension of z must be at least $\max(1, n)$.
 If $jobz = 'U'$, then the matrix z is multiplied by Q .
 If $jobz = 'N'$ or $'V'$, then z is not referenced.
 $work(lwork)$ is a workspace array.
 lda INTEGER. The first dimension of a ; at least $\max(1, n)$.
 ldz INTEGER. The first dimension of z ; at least $\max(1, n)$. Not referenced if $jobz = 'N'$.
 $lwork$ INTEGER. The size of the $work$ array ($lwork \geq (2kd+1)n+kd$).
 If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).
 See *Application Notes* for the suggested value of $lwork$.

Output Parameters

a If $jobz = 'V'$, then overwritten by Q matrix.
 If $jobz = 'N'$ or $'U'$, then overwritten by the banded matrix B and details of the unitary matrix Q_B to reduce A to B as specified by $uplo$.
 z On exit,
 if $jobz = 'U'$, then the matrix z is overwritten by Z^*Q .
 If $jobz = 'N'$ or $'V'$, then z is not referenced.
 d, e COMPLEX for cherdb.
 DOUBLE COMPLEX for zherdb.
 Arrays:
 $d(*)$ contains the diagonal elements of the matrix T .
 The dimension of d must be at least $\max(1, n)$.
 $e(*)$ contains the off-diagonal elements of T .
 The dimension of e must be at least $\max(1, n-1)$.
 $tau(*)$ stores further details of the orthogonal matrix Q .

	The dimension of <i>tau</i> must be at least $\max(1, n-kd-1)$.
<i>tau</i>	COMPLEX for <i>cherdb</i> . DOUBLE COMPLEX for <i>zherdb</i> . Array, DIMENSION at least $\max(1, n-1)$ Stores further details of the unitary matrix Q_B . The dimension of <i>tau</i> must be at least $\max(1, n-kd-1)$.
<i>work</i> (1)	If <i>info</i> =0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Application Notes

For better performance, try using $lwork = n*(3*kd+3)$.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

For better performance, try using *kd* equal to 40 if $n \leq 2000$ and 64 otherwise.

Try using *?herdb* instead of *?hetrd* on large matrices obtaining only eigenvalues - when no eigenvectors are needed, especially in multi-threaded environment. *?herdb* becomes faster beginning approximately with $n = 1000$, and much faster at larger matrices with a better scalability than *?hetrd*.

Avoid applying `?herdb` for computing eigenvectors due to the two-step reduction, that is, the number of operations needed to apply orthogonal transformations to z is doubled compared to the traditional one-step reduction. In that case it is better to apply `?hetrd` and `?unmtr/?ungtr` to obtain tridiagonal form along with the unitary transformation matrix Q .

?orgtr

Generates the real orthogonal matrix Q determined by ?sytrd.

Syntax

FORTRAN 77:

```
call sorgtr(uplo, n, a, lda, tau, work, lwork, info)
```

```
call dorgtr(uplo, n, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call orgtr(a, tau [,uplo] [,info])
```

Description

The routine explicitly generates the n -by- n orthogonal matrix Q formed by `?sytrd` when reducing a real symmetric matrix A to tridiagonal form: $A = Q^T Q$. Use this routine after a call to `?sytrd`.

Input Parameters

<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'. Use the same <code>uplo</code> as supplied to <code>?sytrd</code> .
<code>n</code>	INTEGER. The order of the matrix Q ($n \geq 0$).
<code>a, tau, work</code>	REAL for <code>sorgtr</code> DOUBLE PRECISION for <code>dorgtr</code> . Arrays: <code>a(lda,*)</code> is the array <code>a</code> as returned by <code>?sytrd</code> . The second dimension of <code>a</code> must be at least $\max(1, n)$. <code>tau(*)</code> is the array <code>tau</code> as returned by <code>?sytrd</code> . The dimension of <code>tau</code> must be at least $\max(1, n-1)$. <code>work</code> is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The first dimension of *a*; at least $\max(1, n)$.

lwork INTEGER. The size of the *work* array ($lwork \geq n$).
 If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#).
 See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a Overwritten by the orthogonal matrix *Q*.

work(1) If $info = 0$, on exit *work(1)* contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.
 If $info = 0$, the execution is successful.
 If $info = -i$, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `orgtr` interface are the following:

a Holds the matrix *A* of size (n, n) .

tau Holds the vector of length $(n-1)$.

uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For better performance, try using $lwork = (n-1) * blocksize$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible `lwork` sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix Q differs from an exactly orthogonal matrix by a matrix E such that $\|E\|_2 = O(\epsilon)$, where ϵ is the machine precision.

The approximate number of floating-point operations is $(4/3)n^3$.

The complex counterpart of this routine is `?ungtr`.

`?ormtr`

Multiplies a real matrix by the real orthogonal matrix Q determined by `?sytrd`.

Syntax

FORTRAN 77:

```
call sormtr(side, uplo, trans, m, n, a, lda, tau, c, ldc, work, lwork, info)
call dormtr(side, uplo, trans, m, n, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call ormtr(a, tau, c [,side] [,uplo] [,trans] [,info])
```

Description

The routine multiplies a real matrix C by Q or Q^T , where Q is the orthogonal matrix Q formed by `?sytrd` when reducing a real symmetric matrix A to tridiagonal form: $A = Q^T Q$. Use this routine after a call to `?sytrd`.

Depending on the parameters `side` and `trans`, the routine can form one of the matrix products $Q^T C$, $Q^T C$, $C^T Q$, or $C^T Q^T$ (overwriting the result on C).

Input Parameters

In the descriptions below, r denotes the order of Q :

If $side = 'L'$, $r = m$; if $side = 'R'$, $r = n$.

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If $side = 'L'$, Q or Q^T is applied to C from the left. If $side = 'R'$, Q or Q^T is applied to C from the right.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Use the same <i>uplo</i> as supplied to ?sytrd.
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'T'. If $trans = 'N'$, the routine multiplies C by Q. If $trans = 'T'$, the routine multiplies C by Q^T.
<i>m</i>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).
<i>a, c, tau, work</i>	REAL for sormtr DOUBLE PRECISION for dormtr $a(lda,*)$ and tau are the arrays returned by ?sytrd. The second dimension of a must be at least $\max(1, r)$. The dimension of tau must be at least $\max(1, r-1)$. $c(ldc,*)$ contains the matrix C. The second dimension of c must be at least $\max(1, n)$ $work$ is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The first dimension of a ; $lda \geq \max(1, r)$.
<i>ldc</i>	INTEGER. The first dimension of c ; $ldc \geq \max(1, n)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array. Constraints: $lwork \geq \max(1, n)$ if $side = 'L'$; $lwork \geq \max(1, m)$ if $side = 'R'$. If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla. See <i>Application Notes</i> for the suggested value of <i>lwork</i>.

Output Parameters

<i>c</i>	Overwritten by the product Q^*C , $Q^T C$, C^*Q , or C^*Q^T (as specified by <i>side</i> and <i>trans</i>).
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ormtr` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>r</i> , <i>r</i>). <i>r</i> = <i>m</i> if <i>side</i> = 'L'. <i>r</i> = <i>n</i> if <i>side</i> = 'R'.
<i>tau</i>	Holds the vector of length (<i>r</i> -1).
<i>c</i>	Holds the matrix <i>C</i> of size (<i>m</i> , <i>n</i>).
<i>side</i>	Must be 'L' or 'R'. The default value is 'L'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using *lwork* = *n*blocksize* for *side* = 'L', or *lwork* = *m*blocksize* for *side* = 'R', where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed product differs from the exact product by a matrix E such that $\|E\|^2 = O(\epsilon) * \|C\|^2$.

The total number of floating-point operations is approximately $2*m^2*n$, if `side = 'L'`, or $2*n^2*m$, if `side = 'R'`.

The complex counterpart of this routine is [?unmtr](#).

?hetrd

Reduces a complex Hermitian matrix to tridiagonal form.

Syntax

FORTRAN 77:

```
call chetrd(uplo, n, a, lda, d, e, tau, work, lwork, info)
call zhetrd(uplo, n, a, lda, d, e, tau, work, lwork, info)
```

Fortran 95:

```
call hetrd(a, tau [,uplo] [,info])
```

Description

This routine reduces a complex Hermitian matrix A to symmetric tridiagonal form T by a unitary similarity transformation: $A = Q * T * Q^H$. The unitary matrix Q is not formed explicitly but is represented as a product of $n-1$ elementary reflectors. Routines are provided to work with Q in this representation. (They are described later in this section.)

This routine calls [?latrd](#) to reduce a complex Hermitian matrix A to Hermitian tridiagonal form by a unitary similarity transformation.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).
<i>a, work</i>	COMPLEX for <code>chetrd</code> DOUBLE COMPLEX for <code>zhetr</code> d. <i>a</i> (<i>lda</i> ,*) is an array containing either upper or lower triangular part of the matrix <i>A</i> , as specified by <i>uplo</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; at least $\max(1, n)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array ($lwork \geq n$). If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <code>xerbla</code> . See <i>Application Notes</i> for the suggested value of <i>lwork</i> .

Output Parameters

<i>a</i>	Overwritten by the tridiagonal matrix <i>T</i> and details of the unitary matrix <i>Q</i> , as specified by <i>uplo</i> .
<i>d, e</i>	REAL for <code>chetrd</code> DOUBLE PRECISION for <code>zhetr</code> d. Arrays: <i>d</i> (*) contains the diagonal elements of the matrix <i>T</i> . The dimension of <i>d</i> must be at least $\max(1, n)$. <i>e</i> (*) contains the off-diagonal elements of <i>T</i> . The dimension of <i>e</i> must be at least $\max(1, n-1)$.
<i>tau</i>	COMPLEX for <code>chetrd</code> DOUBLE COMPLEX for <code>zhetr</code> d. Array, DIMENSION at least $\max(1, n-1)$. Stores further details of the unitary matrix <i>Q</i> .

<code>work(1)</code>	If <code>info = 0</code> , on exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance. Use this <code>lwork</code> for subsequent runs.
<code>info</code>	INTEGER. If <code>info = 0</code> , the execution is successful. If <code>info = -i</code> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hetrd` interface are the following:

<code>a</code>	Holds the matrix <i>A</i> of size (n, n) .
<code>tau</code>	Holds the vector of length $(n-1)$.
<code>d</code>	Holds the vector of length (n) .
<code>e</code>	Holds the vector of length $(n-1)$.
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For better performance, try using `lwork = n*blocksize`, where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible `lwork` sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix T is exactly similar to a matrix $A + E$, where $\|E\|_2 = c(n) * \epsilon * \|A\|_2$, $c(n)$ is a modestly increasing function of n , and ϵ is the machine precision.

The approximate number of floating-point operations is $(16/3)n^3$.

After calling this routine, you can call the following:

`?ungtr` to form the computed matrix Q explicitly
`?unmtr` to multiply a complex matrix by Q .

The real counterpart of this routine is `?sytrd`.

?ungtr

Generates the complex unitary matrix Q determined by ?hetrd.

Syntax

FORTRAN 77:

```
call cungr( uplo, n, a, lda, tau, work, lwork, info)
call zungtr( uplo, n, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call ungtr(a, tau [,uplo] [,info])
```

Description

The routine explicitly generates the n -by- n unitary matrix Q formed by `?hetrd` when reducing a complex Hermitian matrix A to tridiagonal form: $A = Q^* T^* Q^H$. Use this routine after a call to `?hetrd`.

Input Parameters

`uplo` CHARACTER*1. Must be 'U' or 'L'.
 Use the same `uplo` as supplied to `?hetrd`.

`n` INTEGER. The order of the matrix Q ($n \geq 0$).

`a, tau, work` COMPLEX for `cungr`
 DOUBLE COMPLEX for `zungtr`.

Arrays:

$a(lda,*)$ is the array a as returned by `?hetrd`.
The second dimension of a must be at least $\max(1, n)$.
 $tau(*)$ is the array tau as returned by `?hetrd`.
The dimension of tau must be at least $\max(1, n-1)$.
 $work$ is a workspace array, its dimension $\max(1, lwork)$.
 lda INTEGER. The first dimension of a ; at least $\max(1, n)$.
 $lwork$ INTEGER. The size of the $work$ array ($lwork \geq n$).
If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by `xerbla`.
See *Application Notes* for the suggested value of $lwork$.

Output Parameters

a Overwritten by the unitary matrix Q .
 $work(1)$ If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.
 $info$ INTEGER.
If $info = 0$, the execution is successful.
If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ungtr` interface are the following:

a Holds the matrix A of size (n, n) .
 tau Holds the vector of length $(n-1)$.
 $uplo$ Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For better performance, try using $lwork = (n-1)*blocksize$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set $lwork = -1$.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix *Q* differs from an exactly unitary matrix by a matrix *E* such that $\|E\|_2 = O(\epsilon)$, where ϵ is the machine precision.

The approximate number of floating-point operations is $(16/3)n^3$.

The real counterpart of this routine is [?orgtr](#).

?unmtr

Multiplies a complex matrix by the complex unitary matrix Q determined by ?hetrd.

Syntax

FORTRAN 77:

```
call cunmtr(side, uplo, trans, m, n, a, lda, tau, c, ldc, work, lwork, info)
call zunmtr(side, uplo, trans, m, n, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call unmtr(a, tau, c [,side] [,uplo] [,trans] [,info])
```

Description

The routine multiplies a complex matrix C by Q or Q^H , where Q is the unitary matrix Q formed by `?hetrd` when reducing a complex Hermitian matrix A to tridiagonal form: $A = Q^* T^* Q^H$. Use this routine after a call to `?hetrd`.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products Q^*C , Q^H^*C , C^*Q , or C^*Q^H (overwriting the result on C).

Input Parameters

In the descriptions below, r denotes the order of Q :

If *side* = 'L', $r = m$; if *side* = 'R', $r = n$.

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^H is applied to C from the left. If <i>side</i> = 'R', Q or Q^H is applied to C from the right.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Use the same <i>uplo</i> as supplied to <code>?hetrd</code> .
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'T'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'T', the routine multiplies C by Q^H .
<i>m</i>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).
<i>a</i> , <i>c</i> , <i>tau</i> , <i>work</i>	COMPLEX for <code>cunmtr</code> DOUBLE COMPLEX for <code>zunmtr</code> . <i>a</i> (<i>lda</i> ,*) and <i>tau</i> are the arrays returned by <code>?hetrd</code> . The second dimension of <i>a</i> must be at least $\max(1, r)$. The dimension of <i>tau</i> must be at least $\max(1, r-1)$. <i>c</i> (<i>ldc</i> ,*) contains the matrix C . The second dimension of <i>c</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; $lda \geq \max(1, r)$.
<i>ldc</i>	INTEGER. The first dimension of <i>c</i> ; $ldc \geq \max(1, n)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array. Constraints:

$lwork \geq \max(1, n)$ if $side = 'L'$;

$lwork \geq \max(1, m)$ if $side = 'R'$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#). See *Application Notes* for the suggested value of *lwork*.

Output Parameters

<i>c</i>	Overwritten by the product Q^*C , $Q^H * C$, C^*Q , or C^*Q^H (as specified by <i>side</i> and <i>trans</i>).
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `unmtr` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>r</i> , <i>r</i>). $r = m$ if $side = 'L'$. $r = n$ if $side = 'R'$.
<i>tau</i>	Holds the vector of length (<i>r</i> -1).
<i>c</i>	Holds the matrix <i>C</i> of size (<i>m</i> , <i>n</i>).
<i>side</i>	Must be 'L' or 'R'. The default value is 'L'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N' or 'C'. The default value is 'N'.

Application Notes

For better performance, try using $lwork = n * blocksize$ (for $side = 'L'$) or $lwork = m * blocksize$ (for $side = 'R'$) where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of $lwork$ for the first run, or set $lwork = -1$.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if $lwork$ is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed product differs from the exact product by a matrix E such that $\|E\|_2 = O(\epsilon) * \|C\|_2$, where ϵ is the machine precision.

The total number of floating-point operations is approximately $8 * m^2 * n$ if $side = 'L'$ or $8 * n^2 * m$ if $side = 'R'$.

The real counterpart of this routine is [?ormtr](#).

?sptd

Reduces a real symmetric matrix to tridiagonal form using packed storage.

Syntax

FORTRAN 77:

```
call ssptd(uplo, n, ap, d, e, tau, info)
call dsptd(uplo, n, ap, d, e, tau, info)
```

Fortran 95:

```
call sptd(a, tau [,uplo] [,info])
```

Description

This routine reduces a packed real symmetric matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation: $A = Q^* T^* Q^T$. The orthogonal matrix Q is not formed explicitly but is represented as a product of $n-1$ elementary reflectors. Routines are provided for working with Q in this representation (see *Application Notes* below).

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
 If *uplo* = 'U', *ap* stores the packed upper triangle of A .
 If *uplo* = 'L', *ap* stores the packed lower triangle of A .

n INTEGER. The order of the matrix A ($n \geq 0$).

ap REAL for ssptrd
 DOUBLE PRECISION for dsptrd.
 Array, DIMENSION at least $\max(1, n(n+1)/2)$. Contains either upper or lower triangle of A (as specified by *uplo*) in packed form.

Output Parameters

ap Overwritten by the tridiagonal matrix T and details of the orthogonal matrix Q , as specified by *uplo*.

d, *e*, *tau* REAL for ssptrd
 DOUBLE PRECISION for dsptrd.
 Arrays:
d(*) contains the diagonal elements of the matrix T .
 The dimension of *d* must be at least $\max(1, n)$.
e(*) contains the off-diagonal elements of T .
 The dimension of *e* must be at least $\max(1, n-1)$.
tau(*) stores further details of the matrix Q .
 The dimension of *tau* must be at least $\max(1, n-1)$.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sptprd` interface are the following:

<code>a</code>	Stands for argument <code>ap</code> in Fortran 77 interface. Holds the array <code>A</code> of size $(n * (n+1) / 2)$.
<code>tau</code>	Holds the vector of length $(n-1)$.
<code>d</code>	Holds the vector of length (n) .
<code>e</code>	Holds the vector of length $(n-1)$.
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed matrix T is exactly similar to a matrix $A+E$, where $\|E\|_2 = c(n) * \epsilon * \|A\|_2$, $c(n)$ is a modestly increasing function of n , and ϵ is the machine precision. The approximate number of floating-point operations is $(4/3) n^3$.

After calling this routine, you can call the following:

<code>?opgtr</code>	to form the computed matrix Q explicitly
<code>?opmtr</code>	to multiply a real matrix by Q .

The complex counterpart of this routine is [?hptprd](#).

?opgtr

Generates the real orthogonal matrix Q determined by ?sptprd.

Syntax

FORTRAN 77:

```
call sopgtr(uplo, n, ap, tau, q, ldq, work, info)
call dopgtr(uplo, n, ap, tau, q, ldq, work, info)
```

Fortran 95:

```
call opgtr(a, tau, q [,uplo] [,info])
```

Description

The routine explicitly generates the n -by- n orthogonal matrix Q formed by ?sptdr when reducing a packed real symmetric matrix A to tridiagonal form: $A = Q^*T^*Q^T$. Use this routine after a call to ?sptdr.

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'. Use the same *uplo* as supplied to ?sptdr.

n INTEGER. The order of the matrix Q ($n \geq 0$).

ap, tau REAL for sopgtr
DOUBLE PRECISION for dopgtr.
Arrays *ap* and *tau*, as returned by ?sptdr.
The dimension of *ap* must be at least $\max(1, n(n+1)/2)$.
The dimension of *tau* must be at least $\max(1, n-1)$.

ldq INTEGER. The first dimension of the output array q_i at least $\max(1, n)$.

work REAL for sopgtr
DOUBLE PRECISION for dopgtr.
Workspace array, DIMENSION at least $\max(1, n-1)$.

Output Parameters

q REAL for sopgtr
DOUBLE PRECISION for dopgtr.
Array, DIMENSION (*ldq*,*).
Contains the computed matrix Q .
The second dimension of *q* must be at least $\max(1, n)$.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `opgtr` interface are the following:

<code>a</code>	Stands for argument <code>ap</code> in Fortran 77 interface. Holds the array <code>A</code> of size $(n * (n + 1) / 2)$.
<code>tau</code>	Holds the vector of length $(n - 1)$.
<code>q</code>	Holds the matrix <code>Q</code> of size (n, n) .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed matrix `Q` differs from an exactly orthogonal matrix by a matrix `E` such that $\|E\|_2 = O(\epsilon)$, where ϵ is the machine precision.

The approximate number of floating-point operations is $(4/3)n^3$.

The complex counterpart of this routine is [?upgtr](#).

?opmtr

Multiplies a real matrix by the real orthogonal matrix `Q` determined by ?sptrd.

Syntax

FORTRAN 77:

```
call sopmtr(side, uplo, trans, m, n, ap, tau, c, ldc, work, info)
call dopmtr(side, uplo, trans, m, n, ap, tau, c, ldc, work, info)
```

Fortran 95:

```
call opmtr(a, tau, c [,side] [,uplo] [,trans] [,info])
```

Description

The routine multiplies a real matrix C by Q or Q^T , where Q is the orthogonal matrix Q formed by `?sptdr` when reducing a packed real symmetric matrix A to tridiagonal form: $A = Q^T T Q$. Use this routine after a call to `?sptdr`.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products $Q^T C$, $Q^T C$, $C^T Q$, or $C^T Q^T$ (overwriting the result on C).

Input Parameters

In the descriptions below, r denotes the order of Q :

If *side* = 'L', $r = m$; if *side* = 'R', $r = n$.

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^T is applied to C from the left. If <i>side</i> = 'R', Q or Q^T is applied to C from the right.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Use the same <i>uplo</i> as supplied to <code>?sptdr</code> .
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'T'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'T', the routine multiplies C by Q^T .
<i>m</i>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).
<i>ap</i> , <i>tau</i> , <i>c</i> , <i>work</i>	REAL for <code>sopmtr</code> DOUBLE PRECISION for <code>dopmtr</code> . <i>ap</i> and <i>tau</i> are the arrays returned by <code>?sptdr</code> . The dimension of <i>ap</i> must be at least $\max(1, r(r+1)/2)$. The dimension of <i>tau</i> must be at least $\max(1, r-1)$. <i>c(ldc,*)</i> contains the matrix C . The second dimension of <i>c</i> must be at least $\max(1, n)$. <i>work(*)</i> is a workspace array. The dimension of <i>work</i> must be at least $\max(1, n)$ if <i>side</i> = 'L'; $\max(1, m)$ if <i>side</i> = 'R'.
<i>ldc</i>	INTEGER. The first dimension of <i>c</i> ; $ldc \geq \max(1, n)$.

Output Parameters

<i>c</i>	Overwritten by the product Q^*C , Q^T*C , $C*Q$, or $C*Q^T$ (as specified by <i>side</i> and <i>trans</i>).
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `opmtr` interface are the following:

<i>a</i>	Stands for argument <i>ap</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(r*(r+1)/2)$, where $r = m$ if <i>side</i> = 'L'. $r = n$ if <i>side</i> = 'R'.
<i>tau</i>	Holds the vector of length $(r-1)$.
<i>c</i>	Holds the matrix <i>C</i> of size (m,n) .
<i>side</i>	Must be 'L' or 'R'. The default value is 'L'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.

Application Notes

The computed product differs from the exact product by a matrix *E* such that $\|E\|_2 = O(\epsilon)$ $\|C\|_2$, where ϵ is the machine precision.

The total number of floating-point operations is approximately $2*m^2*n$ if *side* = 'L', or $2*n^2*m$ if *side* = 'R'.

The complex counterpart of this routine is [?upmtr](#).

?hptrd

Reduces a complex Hermitian matrix to tridiagonal form using packed storage.

Syntax

FORTRAN 77:

```
call chptrd(uplo, n, ap, d, e, tau, info)
call zhptrd(uplo, n, ap, d, e, tau, info)
```

Fortran 95:

```
call hptrd(a, tau [,uplo] [,info])
```

Description

This routine reduces a packed complex Hermitian matrix A to symmetric tridiagonal form T by a unitary similarity transformation: $A = Q^* T^* Q^H$. The unitary matrix Q is not formed explicitly but is represented as a product of $n-1$ elementary reflectors. Routines are provided for working with Q in this representation (see *Application Notes* below).

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ap</i> stores the packed upper triangle of A . If <i>uplo</i> = 'L', <i>ap</i> stores the packed lower triangle of A .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>ap</i>	COMPLEX for <i>chptrd</i> DOUBLE COMPLEX for <i>zhptrd</i> . Array, DIMENSION at least $\max(1, n(n+1)/2)$. Contains either upper or lower triangle of A (as specified by <i>uplo</i>) in packed form.

Output Parameters

<i>ap</i>	Overwritten by the tridiagonal matrix T and details of the orthogonal matrix Q , as specified by <i>uplo</i> .
<i>d</i> , <i>e</i>	REAL for <i>chptrd</i> DOUBLE PRECISION for <i>zhptrd</i> .

Arrays:
 $d(*)$ contains the diagonal elements of the matrix T .
 The dimension of d must be at least $\max(1, n)$.
 $e(*)$ contains the off-diagonal elements of T .
 The dimension of e must be at least $\max(1, n-1)$.

tau COMPLEX for `chptrd`
 DOUBLE COMPLEX for `zhptrd`.
 Arrays, DIMENSION at least $\max(1, n-1)$. Contains further
 details of the orthogonal matrix Q .

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hptrd` interface are the following:

<i>a</i>	Stands for argument <i>ap</i> in Fortran 77 interface. Holds the array A of size $(n * (n+1) / 2)$.
<i>tau</i>	Holds the vector of length $(n-1)$.
<i>d</i>	Holds the vector of length (n) .
<i>e</i>	Holds the vector of length $(n-1)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed matrix T is exactly similar to a matrix $A + E$, where $\|E\|_2 = c(n) * \epsilon * \|A\|_2$, $c(n)$ is a modestly increasing function of n , and ϵ is the machine precision.

The approximate number of floating-point operations is $(16/3) n^3$.

After calling this routine, you can call the following:

<code>?upgtr</code>	to form the computed matrix Q explicitly
<code>?upmtr</code>	to multiply a complex matrix by Q .

The real counterpart of this routine is `?sptrd`.

?upgtr

Generates the complex unitary matrix Q determined by ?hptrd.

Syntax

FORTRAN 77:

```
call cupgtr(uplo, n, ap, tau, q, ldq, work, info)
call zupgtr(uplo, n, ap, tau, q, ldq, work, info)
```

Fortran 95:

```
call upgtr(a, tau, q [,uplo] [,info])
```

Description

The routine explicitly generates the n -by- n unitary matrix Q formed by ?hptrd when reducing a packed complex Hermitian matrix A to tridiagonal form: $A = Q^* T^* Q^H$. Use this routine after a call to ?hptrd.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Use the same <i>uplo</i> as supplied to ?hptrd.
<i>n</i>	INTEGER. The order of the matrix Q ($n \geq 0$).
<i>ap, tau</i>	COMPLEX for cupgtr DOUBLE COMPLEX for zupgtr. Arrays <i>ap</i> and <i>tau</i> , as returned by ?hptrd. The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$. The dimension of <i>tau</i> must be at least $\max(1, n-1)$.
<i>ldq</i>	INTEGER. The first dimension of the output array <i>q</i> ; at least $\max(1, n)$.
<i>work</i>	COMPLEX for cupgtr DOUBLE COMPLEX for zupgtr. Workspace array, DIMENSION at least $\max(1, n-1)$.

Output Parameters

<i>q</i>	COMPLEX for <code>cupgtr</code> DOUBLE COMPLEX for <code>zupgtr</code> . Array, DIMENSION (<i>ldq</i> ,*). Contains the computed matrix <i>Q</i> . The second dimension of <i>q</i> must be at least $\max(1, n)$.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `upgtr` interface are the following:

<i>a</i>	Stands for argument <i>ap</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(n * (n + 1) / 2)$.
<i>tau</i>	Holds the vector of length $(n - 1)$.
<i>q</i>	Holds the matrix <i>Q</i> of size (n, n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed matrix *Q* differs from an exactly orthogonal matrix by a matrix *E* such that $\|E\|_2 = O(\varepsilon)$, where ε is the machine precision.

The approximate number of floating-point operations is $(16/3) n^3$.

The real counterpart of this routine is `?opgtr`.

?upmtr

Multiplies a complex matrix by the unitary matrix Q determined by ?hptrd.

Syntax

FORTRAN 77:

```
call cupmtr(side, uplo, trans, m, n, ap, tau, c, ldc, work, info)
call zupmtr(side, uplo, trans, m, n, ap, tau, c, ldc, work, info)
```

Fortran 95:

```
call upmtr(a, tau, c [,side] [,uplo] [,trans] [,info])
```

Description

The routine multiplies a complex matrix C by Q or Q^H , where Q is the unitary matrix formed by ?hptrd when reducing a packed complex Hermitian matrix A to tridiagonal form: $A = Q^* T Q^H$. Use this routine after a call to ?hptrd.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products $Q^* C$, $Q^H C$, $C Q$, or $C Q^H$ (overwriting the result on C).

Input Parameters

In the descriptions below, r denotes the order of Q :

If *side* = 'L', $r = m$; if *side* = 'R', $r = n$.

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^H is applied to C from the left. If <i>side</i> = 'R', Q or Q^H is applied to C from the right.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Use the same <i>uplo</i> as supplied to ?hptrd.
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'T'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'T', the routine multiplies C by Q^H .
<i>m</i>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).

ap, *tau*, *c*,
 COMPLEX for `cupmtr`
 DOUBLE COMPLEX for `zupmtr`.
ap and *tau* are the arrays returned by `?hptrd`.
 The dimension of *ap* must be at least $\max(1, r(r+1)/2)$.
 The dimension of *tau* must be at least $\max(1, r-1)$.
c(ldc,)* contains the matrix *C*.
 The second dimension of *c* must be at least $\max(1, n)$
work()* is a workspace array.
 The dimension of *work* must be at least
 $\max(1, n)$ if *side* = 'L';
 $\max(1, m)$ if *side* = 'R'.
ldc
 INTEGER. The first dimension of *c*; $ldc \geq \max(1, n)$.

Output Parameters

c
 Overwritten by the product Q^*C , Q^H*C , $C*Q$, or $C*Q^H$ (as specified by *side* and *trans*).
info
 INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `upmtr` interface are the following:

a
 Stands for argument *ap* in Fortran 77 interface. Holds the array *A* of size $(r*(r+1)/2)$, where
 $r = m$ if *side* = 'L'.
 $r = n$ if *side* = 'R'.
tau
 Holds the vector of length $(r-1)$.
c
 Holds the matrix *C* of size (m, n) .
side
 Must be 'L' or 'R'. The default value is 'L'.
uplo
 Must be 'U' or 'L'. The default value is 'U'.
trans
 Must be 'N' or 'C'. The default value is 'N'.

Application Notes

The computed product differs from the exact product by a matrix E such that $\|E\|_2 = O(\epsilon) * \|C\|_2$, where ϵ is the machine precision.

The total number of floating-point operations is approximately $8*m^2*n$ if $side = 'L'$ or $8*n^2*m$ if $side = 'R'$.

The real counterpart of this routine is `?opmtr`.

?sbtrd

Reduces a real symmetric band matrix to tridiagonal form.

Syntax

FORTRAN 77:

```
call ssbtrd(vect, uplo, n, kd, ab, ldab, d, e, q, ldq, work, info)
call dsbtrd(vect, uplo, n, kd, ab, ldab, d, e, q, ldq, work, info)
```

Fortran 95:

```
call sbtrd(a [, q] [, vect] [, uplo] [, info])
```

Description

This routine reduces a real symmetric band matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation: $A = Q*T*Q^T$. The orthogonal matrix Q is determined as a product of Givens rotations.

If required, the routine can also form the matrix Q explicitly.

Input Parameters

<code>vect</code>	CHARACTER*1. Must be 'V' or 'N'. If <code>vect = 'V'</code> , the routine returns the explicit matrix Q . If <code>vect = 'N'</code> , the routine does not return Q .
<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'. If <code>uplo = 'U'</code> , <code>ab</code> stores the upper triangular part of A . If <code>uplo = 'L'</code> , <code>ab</code> stores the lower triangular part of A .

<i>n</i>	INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).
<i>kd</i>	INTEGER. The number of super- or sub-diagonals in <i>A</i> ($kd \geq 0$).
<i>ab, work</i>	REAL for ssbtrd DOUBLE PRECISION for dsbtrd. <i>ab</i> (<i>ldab</i> ,*) is an array containing either upper or lower triangular part of the matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format. The second dimension of <i>ab</i> must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, n)$.
<i>ldab</i>	INTEGER. The first dimension of <i>ab</i> ; at least $kd+1$.
<i>ldq</i>	INTEGER. The first dimension of <i>q</i> . Constraints: $ldq \geq \max(1, n)$ if <i>vect</i> = 'V'; $ldq \geq 1$ if <i>vect</i> = 'N'.

Output Parameters

<i>ab</i>	On exit, the array <i>ab</i> is overwritten.
<i>d, e, q</i>	REAL for ssbtrd DOUBLE PRECISION for dsbtrd. Arrays: <i>d</i> (*) contains the diagonal elements of the matrix <i>T</i> . The dimension of <i>d</i> must be at least $\max(1, n)$. <i>e</i> (*) contains the off-diagonal elements of <i>T</i> . The dimension of <i>e</i> must be at least $\max(1, n-1)$. <i>q</i> (<i>ldq</i> ,*) is not referenced if <i>vect</i> = 'N'. If <i>vect</i> = 'V', <i>q</i> contains the <i>n</i> -by- <i>n</i> matrix <i>Q</i> . The second dimension of <i>q</i> must be: at least $\max(1, n)$ if <i>vect</i> = 'V'; at least 1 if <i>vect</i> = 'N'.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sbtrd` interface are the following:

<code>a</code>	Stands for argument <code>ab</code> in Fortran 77 interface. Holds the array <code>A</code> of size $(kd+1, n)$.
<code>q</code>	Holds the matrix <code>Q</code> of size (n, n) .
<code>d</code>	Holds the vector of length (n) .
<code>e</code>	Holds the vector of length $(n-1)$.
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.
<code>vect</code>	If omitted, this argument is restored based on the presence of argument <code>q</code> as follows: <code>vect</code> = 'V', if <code>q</code> is present, <code>vect</code> = 'N', if <code>q</code> is omitted. If present, <code>vect</code> must be equal to 'V' or 'U' and the argument <code>q</code> must also be present. Note that there will be an error condition if <code>vect</code> is present and <code>q</code> omitted.

Application Notes

The computed matrix `T` is exactly similar to a matrix $A+E$, where $\|E\|_2 = c(n) * \epsilon * \|A\|_2$, $c(n)$ is a modestly increasing function of n , and ϵ is the machine precision. The computed matrix `Q` differs from an exactly orthogonal matrix by a matrix `E` such that $\|E\|_2 = O(\epsilon)$.

The total number of floating-point operations is approximately $6n^2 * kd$ if `vect` = 'N', with $3n^3 * (kd-1) / kd$ additional operations if `vect` = 'V'.

The complex counterpart of this routine is [?hbtrd](#).

?hbtrd

Reduces a complex Hermitian band matrix to tridiagonal form.

Syntax

FORTRAN 77:

```
call chbtrd(vect, uplo, n, kd, ab, ldab, d, e, q, ldq, work, info)
call zhbtrd(vect, uplo, n, kd, ab, ldab, d, e, q, ldq, work, info)
```

Fortran 95:

```
call hbtrd(a [, q] [,vect] [,uplo] [,info])
```

Description

This routine reduces a complex Hermitian band matrix A to symmetric tridiagonal form T by a unitary similarity transformation: $A = Q^* T Q^H$. The unitary matrix Q is determined as a product of Givens rotations.

If required, the routine can also form the matrix Q explicitly.

Input Parameters

<i>vect</i>	CHARACTER*1. Must be 'V' or 'N'. If <i>vect</i> = 'V', the routine returns the explicit matrix Q . If <i>vect</i> = 'N', the routine does not return Q .
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ab</i> stores the upper triangular part of A . If <i>uplo</i> = 'L', <i>ab</i> stores the lower triangular part of A .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>kd</i>	INTEGER. The number of super- or sub-diagonals in A ($kd \geq 0$).
<i>ab, work</i>	COMPLEX for chbtrd DOUBLE COMPLEX for zhbtrd. <i>ab</i> (<i>ldab</i> ,*) is an array containing either upper or lower triangular part of the matrix A (as specified by <i>uplo</i>) in band storage format.

The second dimension of *ab* must be at least $\max(1, n)$.
work(*) is a workspace array.
 The dimension of *work* must be at least $\max(1, n)$.
ldab INTEGER. The first dimension of *ab*; at least $kd+1$.
ldq INTEGER. The first dimension of *q*. Constraints:
 $ldq \geq \max(1, n)$ if *vect* = 'V';
 $ldq \geq 1$ if *vect* = 'N'.

Output Parameters

ab On exit, the array *ab* is overwritten.
d, e REAL for *chbtrd*
 DOUBLE PRECISION for *zhbtrd*.
 Arrays:
d(*) contains the diagonal elements of the matrix *T*.
 The dimension of *d* must be at least $\max(1, n)$.
e(*) contains the off-diagonal elements of *T*.
 The dimension of *e* must be at least $\max(1, n-1)$.
q COMPLEX for *chbtrd*
 DOUBLE COMPLEX for *zhbtrd*.
 Array, DIMENSION (*ldq*,*).
 If *vect* = 'N', *q* is not referenced.
 If *vect* = 'V', *q* contains the *n*-by-*n* matrix *Q*.
 The second dimension of *q* must be:
 at least $\max(1, n)$ if *vect* = 'V';
 at least 1 if *vect* = 'N'.
info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *hbtrd* interface are the following:

<i>a</i>	Stands for argument <i>ab</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(kd+1, n)$.
<i>q</i>	Holds the matrix <i>Q</i> of size (n, n) .
<i>d</i>	Holds the vector of length (n) .
<i>e</i>	Holds the vector of length $(n-1)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>vect</i>	If omitted, this argument is restored based on the presence of argument <i>q</i> as follows: <i>vect</i> = 'V', if <i>q</i> is present, <i>vect</i> = 'N', if <i>q</i> is omitted. If present, <i>vect</i> must be equal to 'V' or 'U' and the argument <i>q</i> must also be present. Note that there will be an error condition if <i>vect</i> is present and <i>q</i> omitted.

Application Notes

The computed matrix *T* is exactly similar to a matrix $A + E$, where $\|E\|_2 = c(n) * \epsilon * \|A\|_2$, $c(n)$ is a modestly increasing function of n , and ϵ is the machine precision. The computed matrix *Q* differs from an exactly unitary matrix by a matrix *E* such that $\|E\|_2 = O(\epsilon)$.

The total number of floating-point operations is approximately $20n^2 * kd$ if *vect* = 'N', with $10n^3 * (kd-1) / kd$ additional operations if *vect* = 'V'.

The real counterpart of this routine is [?sbtrd](#).

?sterf

Computes all eigenvalues of a real symmetric tridiagonal matrix using QR algorithm.

Syntax

FORTRAN 77:

```
call ssterf(n, d, e, info)
call dsterf(n, d, e, info)
```

Fortran 95:

```
call sterf(d, e [,info])
```

Description

This routine computes all the eigenvalues of a real symmetric tridiagonal matrix T (which can be obtained by reducing a symmetric or Hermitian matrix to tridiagonal form). The routine uses a square-root-free variant of the QR algorithm.

If you need not only the eigenvalues but also the eigenvectors, call [?stegr](#).

Input Parameters

n INTEGER. The order of the matrix T ($n \geq 0$).

d, e REAL for `ssterf`
DOUBLE PRECISION for `dsterf`.
Arrays:
 $d(*)$ contains the diagonal elements of T .
The dimension of d must be at least $\max(1, n)$.
 $e(*)$ contains the off-diagonal elements of T .
The dimension of e must be at least $\max(1, n-1)$.

Output Parameters

d The n eigenvalues in ascending order, unless $info > 0$.
See also $info$.

e On exit, the array is overwritten; see $info$.

$info$ INTEGER.
If $info = 0$, the execution is successful.
If $info = i$, the algorithm failed to find all the eigenvalues after $30n$ iterations:
 i off-diagonal elements have not converged to zero. On exit, d and e contain, respectively, the diagonal and off-diagonal elements of a tridiagonal matrix orthogonally similar to T .
If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sterf` interface are the following:

d Holds the vector of length (*n*).

e Holds the vector of length (*n*-1).

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T+E$ such that $\|E\|_2 = O(\epsilon) * \|T\|_2$, where ϵ is the machine precision.

If λ_i is an exact eigenvalue, and m_i is the corresponding computed value, then

$$|\mu_i - \lambda_i| \leq c(n) * \epsilon * \|T\|_2$$

where $c(n)$ is a modestly increasing function of n .

The total number of floating-point operations depends on how rapidly the algorithm converges. Typically, it is about $14n^2$.

?steqr

Computes all eigenvalues and eigenvectors of a symmetric or Hermitian matrix reduced to tridiagonal form (QR algorithm).

Syntax

FORTRAN 77:

```
call ssteqr(compz, n, d, e, z, ldz, work, info)
call dsteqr(compz, n, d, e, z, ldz, work, info)
call csteqr(compz, n, d, e, z, ldz, work, info)
call zsteqr(compz, n, d, e, z, ldz, work, info)
```

Fortran 95:

```
call rsteqr(d, e [,z] [,compz] [,info])
call steqr(d, e [,z] [,compz] [,info])
```

Description

This routine computes all the eigenvalues and (optionally) all the eigenvectors of a real symmetric tridiagonal matrix T . In other words, the routine can compute the spectral factorization: $T = Z^* \Lambda Z^T$. Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i ; Z is an orthogonal matrix whose columns are eigenvectors. Thus,

$$T^* Z_i = \lambda_i^* Z_i \text{ for } i = 1, 2, \dots, n.$$

The routine normalizes the eigenvectors so that $\|Z_i\|_2 = 1$.

You can also use the routine for computing the eigenvalues and eigenvectors of an arbitrary real symmetric (or complex Hermitian) matrix A reduced to tridiagonal form T : $A = Q^* T Q^H$.

In this case, the spectral factorization is as follows: $A = Q^* T Q^H = (Q^* Z)^* \Lambda^* (Q^* Z)^H$. Before calling `?steqr`, you must reduce A to tridiagonal form and generate the explicit matrix Q by calling the following routines:

	for real matrices:	for complex matrices:
full storage	<code>?sytrd</code> , <code>?orgtr</code>	<code>?hetrd</code> , <code>?ungtr</code>
packed storage	<code>?sptrd</code> , <code>?opgtr</code>	<code>?hptrd</code> , <code>?upgtr</code>
band storage	<code>?sbtrd</code> (<code>vect='V'</code>)	<code>?hbtrd</code> (<code>vect='V'</code>)

If you need eigenvalues only, it's more efficient to call `?sterf`. If T is positive-definite, `?pteqr` can compute small eigenvalues more accurately than `?steqr`.

To solve the problem by a single call, use one of the divide and conquer routines `?stevd`, `?syevd`, `?spevd`, or `?sbevd` for real symmetric matrices or `?heevd`, `?hpevd`, or `?hbevd` for complex Hermitian matrices.

Input Parameters

compz

CHARACTER*1. Must be 'N' or 'I' or 'V'.

If *compz* = 'N', the routine computes eigenvalues only.

If *compz* = 'I', the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix T .

If *compz* = 'V', the routine computes the eigenvalues and eigenvectors of A (and the array z must contain the matrix Q on entry).

n	INTEGER. The order of the matrix T ($n \geq 0$).
$d, e, work$	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays: $d(*)$ contains the diagonal elements of T . The dimension of d must be at least $\max(1, n)$. $e(*)$ contains the off-diagonal elements of T . The dimension of e must be at least $\max(1, n-1)$. $work(*)$ is a workspace array. The dimension of $work$ must be: at least 1 if $compz = 'N'$; at least $\max(1, 2*n-2)$ if $compz = 'V'$ or $'I'$.
z	REAL for ssteqr DOUBLE PRECISION for dsteqr COMPLEX for csteqr DOUBLE COMPLEX for zsteqr. Array, DIMENSION ($ldz, *$) If $compz = 'N'$ or $'I'$, z need not be set. If $vect = 'V'$, z must contain the n -by- n matrix Q . The second dimension of z must be: at least 1 if $compz = 'N'$; at least $\max(1, n)$ if $compz = 'V'$ or $'I'$. $work(lwork)$ is a workspace array.
ldz	INTEGER. The first dimension of z . Constraints: $ldz \geq 1$ if $compz = 'N'$; $ldz \geq \max(1, n)$ if $compz = 'V'$ or $'I'$.

Output Parameters

d	The n eigenvalues in ascending order, unless $info > 0$. See also $info$.
e	On exit, the array is overwritten; see $info$.
z	If $info = 0$, contains the n orthonormal eigenvectors, stored by columns. (The i -th column corresponds to the i th eigenvalue.)
$info$	INTEGER.

If $info = 0$, the execution is successful.
 If $info = i$, the algorithm failed to find all the eigenvalues after $30n$ iterations: i off-diagonal elements have not converged to zero. On exit, d and e contain, respectively, the diagonal and off-diagonal elements of a tridiagonal matrix orthogonally similar to T .
 If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `steqr` interface are the following:

d	Holds the vector of length (n) .
e	Holds the vector of length $(n-1)$.
z	Holds the matrix z of size (n, n) .
$compz$	If omitted, this argument is restored based on the presence of argument z as follows: $compz = 'I'$, if z is present, $compz = 'N'$, if z is omitted. If present, $compz$ must be equal to 'I' or 'V' and the argument z must also be present. Note that there will be an error condition if $compz$ is present and z omitted. Note that two variants of Fortran 95 interface for <code>steqr</code> routine are needed because of an ambiguous choice between real and complex cases appear when z is omitted. Thus, the name <code>rsteqr</code> is used in real cases (single or double precision), and the name <code>steqr</code> is used in complex cases (single or double precision).

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T+E$ such that $\|E\|_2 = O(\epsilon) * \|T\|_2$, where ϵ is the machine precision.

If λ_i is an exact eigenvalue, and μ_i is the corresponding computed value, then

$$|\mu_i - \lambda_i| \leq c(n) * \epsilon * \|T\|_2$$

where $c(n)$ is a modestly increasing function of n .

If z_i is the corresponding exact eigenvector, and w_i is the corresponding computed vector, then the angle $\theta(z_i, w_i)$ between them is bounded as follows:

$$\theta(z_i, w_i) \leq c(n) * \epsilon * \|T\|_2 / \min_{i \neq j} |\lambda_i - \lambda_j|.$$

The total number of floating-point operations depends on how rapidly the algorithm converges. Typically, it is about

$24n^2$ if `compz = 'N'`;

$7n^3$ (for complex flavors, $14n^3$) if `compz = 'V' or 'I'`.

stemr

Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

Syntax

FORTRAN 77:

```
call sstemr(jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, isuppz,
tryrac, work, lwork, iwork, liwork, info)

call dstemr(jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, isuppz,
tryrac, work, lwork, iwork, liwork, info)

call cstemr(jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, isuppz,
tryrac, work, lwork, iwork, liwork, info)

call zstemr(jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, isuppz,
tryrac, work, lwork, iwork, liwork, info)
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix T . Any such unreduced matrix has a well defined set of pairwise different real eigenvalues, the corresponding real eigenvectors are pairwise orthogonal.

The spectrum may be computed either completely or partially by specifying either an interval $(vl, vu]$ or a range of indices `il:iu` for the desired eigenvalues.

Depending on the number of desired eigenvalues, these are computed either by bisection or the *dqds* algorithm. Numerically orthogonal eigenvectors are computed by the use of various suitable $L^*D^*L^T$ factorizations near clusters of close eigenvalues (referred to as RRRs, Relatively Robust Representations). An informal sketch of the algorithm follows.

For each unreduced block (submatrix) of T ,

- a.** Compute $T - \sigma I = L^*D^*L^T$, so that L and D define all the wanted eigenvalues to high relative accuracy. This means that small relative changes in the entries of L and D cause only small relative changes in the eigenvalues and eigenvectors. The standard (unfactored) representation of the tridiagonal matrix T does not have this property in general.
- b.** Compute the eigenvalues to suitable accuracy. If the eigenvectors are desired, the algorithm attains full accuracy of the computed eigenvalues only right before the corresponding vectors have to be computed, see steps c and d.
- c.** For each cluster of close eigenvalues, select a new shift close to the cluster, find a new factorization, and refine the shifted eigenvalues to suitable accuracy.
- d.** For each eigenvalue with a large enough relative separation compute the corresponding eigenvector by forming a rank revealing twisted factorization. Go back to step c for any clusters that remain.

For more details, see: [Dhillon04], [Dhillon04-02], [Dhillon97]

The routine works only on machines which follow IEEE-754 floating-point standard in their handling of infinities and NaNs (NaN stands for "not a number"). This permits the use of efficient inner loops avoiding a check for zero divisors.

LAPACK routines can be used to reduce a complex Hermitean matrix to real symmetric tridiagonal form.

(Any complex Hermitean tridiagonal matrix has real values on its diagonal and potentially complex numbers on its off-diagonals. By applying a similarity transform with an appropriate diagonal matrix $\text{diag}(1, e^{i\phi_1}, \dots, e^{i\phi_{n-1}})$, the complex Hermitean matrix can be transformed into a real symmetric matrix and complex arithmetic can be entirely avoided.) While the eigenvectors of the real symmetric tridiagonal matrix are real, the eigenvectors of original complex Hermitean matrix have complex entries in general. Since LAPACK drivers overwrite the matrix data with the eigenvectors, *zstemr* accepts complex workspace to facilitate interoperability with *zunmtr* or *zupmtr*.

Input Parameters

jobz CHARACTER*1. Must be 'N' or 'V'.
If *jobz* = 'N', then only eigenvalues are computed.

	If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes all eigenvalues in the half-open interval: (<i>vl</i>, <i>vu</i>]. If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i>.
<i>n</i>	INTEGER. The order of the matrix <i>T</i> (<i>n</i>≥0).
<i>d</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION (<i>n</i>). Contains <i>n</i> diagonal elements of the tridiagonal matrix <i>T</i>.
<i>e</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION (<i>n</i>-1). Contains (<i>n</i>-1) off-diagonal elements of the tridiagonal matrix <i>T</i> in elements 1 to <i>n</i>-1 of <i>e</i>. <i>e</i>(<i>n</i>) need not be set on input, but is used internally as workspace.
<i>vl, vu</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: <i>vl</i><<i>vu</i>. If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il, iu</i>	INTEGER. If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: 1≤<i>il</i>≤<i>iu</i>≤<i>n</i>, if <i>n</i>>0. If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i>. if <i>jobz</i> = 'V', then <i>ldz</i> ≥ max(1, <i>n</i>); <i>ldz</i> ≥ 1 otherwise.
<i>nzc</i>	INTEGER. The number of eigenvectors to be held in the array <i>z</i>. If <i>range</i> = 'A', then <i>nzc</i>≥max(1, <i>n</i>);

If *range* = 'V', then *nzc* is greater than or equal to the number of eigenvalues in the half-open interval: (*vl*, *vu*].

If *range* = 'I', then $nzc \geq il + iu + 1$.

This value is returned as the first entry of the array *z*, and no error message related to *nzc* is issued by the routine *xerbla*.

tryrac

LOGICAL.

If *tryrac* = .TRUE., it indicates that the code should check whether the tridiagonal matrix defines its eigenvalues to high relative accuracy. If so, the code uses relative-accuracy preserving algorithms that might be (a bit) slower depending on the matrix. If the matrix does not define its eigenvalues to high relative accuracy, the code can use possibly faster algorithms.

If *tryrac* = .FALSE., the code is not required to guarantee relatively accurate eigenvalues and can use the fastest possible techniques.

work

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Workspace array, DIMENSION (*lwork*).

lwork

INTEGER.

The dimension of the array *work*,

$lwork \geq \max(1, 18 * n)$.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by *xerbla*.

iwork

INTEGER.

Workspace array, DIMENSION (*liwork*).

liwork

INTEGER.

The dimension of the array *iwork*.

$lwork \geq \max(1, 10 * n)$ if the eigenvectors are desired, and

$lwork \geq \max(1, 8 * n)$ if only the eigenvalues are to be computed.

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *iwork* array, returns this value as the first entry of the *iwork* array, and no error message related to *liwork* is issued by xerbla.

Output Parameters

<i>e</i>	On exit, the array <i>e</i> is overwritten.
<i>m</i>	INTEGER. The total number of eigenvalues found, $0 \leq m \leq n$. If <i>range</i> = 'A', then $m=n$, and if <i>range</i> = 'I', then $m=iu-il+1$.
<i>w</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION (<i>n</i>). The first <i>m</i> elements contain the selected eigenvalues in ascending order.
<i>z</i>	REAL for sstemr DOUBLE PRECISION for dstemr COMPLEX for cstemr DOUBLE COMPLEX for zstemr. Array <i>z</i>(<i>ldz</i>, *), the second dimension of <i>z</i> must be at least $\max(1, m)$. If <i>jobz</i> = 'V', and <i>info</i> = 0, then the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix <i>T</i> corresponding to the selected eigenvalues, with the <i>i</i>-th column of <i>z</i> holding the eigenvector associated with <i>w</i>(<i>i</i>). If <i>jobz</i> = 'N', then <i>z</i> is not referenced. Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array <i>z</i>; if <i>range</i> = 'V', the exact value of <i>m</i> is not known in advance and can be computed with a workspace query by setting <i>nzc</i> = -1, see description of the parameter <i>nzc</i>.
<i>isuppz</i>	INTEGER. Array, DIMENSION ($2 * \max(1, m)$).

	The support of the eigenvectors in <i>z</i> , that is the indices indicating the nonzero elements in <i>z</i> . The <i>i</i> -th computed eigenvector is nonzero only in elements <i>isuppz</i> (2*i-1) through <i>isuppz</i> (2*i). This is relevant in the case when the matrix is split. <i>isuppz</i> is only accessed when <i>jobz</i> = 'V' and <i>n</i> >0.
<i>tryrac</i>	On exit, TRUE. <i>tryrac</i> is set to .FALSE. if the matrix does not define its eigenvalues to high relative accuracy.
<i>work</i> (1)	On exit, if <i>info</i> = 0, then <i>work</i> (1) returns the optimal (and minimal) size of <i>lwork</i> .
<i>iwork</i> (1)	On exit, if <i>info</i> = 0, then <i>iwork</i> (1) returns the optimal size of <i>liwork</i> .
<i>info</i>	INTEGER. If = 0, the execution is successful. If <i>info</i> = -i, the <i>i</i> -th parameter had an illegal value. If <i>info</i> = 1, internal error in ?larre occurred, if <i>info</i> = 2, internal error in ?larrv occurred.

?stedc

Computes all eigenvalues and eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method.

Syntax

FORTRAN 77:

```
call sstedc(compz, n, d, e, z, ldz, work, lwork, iwork, liwork, info)
call dstedc(compz, n, d, e, z, ldz, work, lwork, iwork, liwork, info)
call cstedc(compz, n, d, e, z, ldz, work, lwork, rwork, lrwork, iwork, liwork, info)
call zstedc(compz, n, d, e, z, ldz, work, lwork, rwork, lrwork, iwork, liwork, info)
```

Fortran 95:

```
call rstedc(d, e [,z] [,compz] [,info])
```

```
call stedc(d, e [,z] [,compz] [,info])
```

Description

This routine computes all the eigenvalues and (optionally) all the eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method. The eigenvectors of a full or band real symmetric or complex Hermitian matrix can also be found if [?sytrd/?hetrd](#) or [?sptrd/?hptrd](#) or [?sbtrd/?hbtrd](#) has been used to reduce this matrix to tridiagonal form.

See also [?laed0](#), [?laed1](#), [?laed2](#), [?laed3](#), [?laed4](#), [?laed5](#), [?laed6](#), [?laed7](#), [?laed8](#), [?laed9](#), and [?laeda](#) used by this function.

Input Parameters

<i>compz</i>	CHARACTER*1. Must be 'N' or 'I' or 'V'. If <i>compz</i> = 'N', the routine computes eigenvalues only. If <i>compz</i> = 'I', the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix. If <i>compz</i> = 'V', the routine computes the eigenvalues and eigenvectors of original symmetric/Hermitian matrix. On entry, the array <i>z</i> must contain the orthogonal/unitary matrix used to reduce the original matrix to tridiagonal form.
<i>n</i>	INTEGER. The order of the symmetric tridiagonal matrix ($n \geq 0$).
<i>d, e, rwork</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays: <i>d</i>(*) contains the diagonal elements of the tridiagonal matrix. The dimension of <i>d</i> must be at least $\max(1, n)$. <i>e</i>(*) contains the subdiagonal elements of the tridiagonal matrix. The dimension of <i>e</i> must be at least $\max(1, n-1)$. <i>rwork</i> is a workspace array, its dimension $\max(1, lrwork)$.
<i>z, work</i>	REAL for sstedc DOUBLE PRECISION for dstedc

COMPLEX for `cstedc`
 DOUBLE COMPLEX for `zstedc`.
 Arrays: `z(ldz, *)`, `work(*)`.
 If `compz = 'V'`, then, on entry, `z` must contain the orthogonal/unitary matrix used to reduce the original matrix to tridiagonal form.
 The second dimension of `z` must be at least $\max(1, n)$.
`work` is a workspace array, its dimension $\max(1, lwork)$.

`ldz` INTEGER. The first dimension of `z`. Constraints:
 $ldz \geq 1$ if `compz = 'N'`;
 $ldz \geq \max(1, n)$ if `compz = 'V' or 'I'`.

`lwork` INTEGER. The dimension of the array `work`.
 If `compz = 'N' or 'I'`, or $n \leq 1$, `lwork` must be at least 1.
 If `compz = 'V'` and $n > 1$, `lwork` must be at least $n*n$.
 Note that for `compz = 'V'`, and if n is less than or equal to the minimum divide size, usually 25, then `lwork` need only be 1.
 If `lwork = -1`, then a workspace query is assumed; the routine only calculates the optimal size of the `work`, `rwork` and `iwork` arrays, returns these values as the first entries of the `work`, `rwork` and `iwork` arrays, and no error message related to `lwork` or `lrwork` or `liwork` is issued by [xerbla](#). See *Application Notes* for the required value of `lwork`.

`lrwork` INTEGER. The dimension of the array `rwork` (used for complex flavors only).
 If `compz = 'N'`, or $n \leq 1$, `lrwork` must be at least 1.
 If `compz = 'V'` and $n > 1$, `lrwork` must be at least $(1+3*n+2*n*\lg(n)+3*n*n)$, where $\lg(n)$ is the smallest integer k such that $2^{**k} \geq n$.
 If `compz = 'I'` and $n > 1$, `lrwork` must be at least $(1+4*n+2*n*n)$.
 Note that for `compz = 'V' or 'I'`, and if n is less than or equal to the minimum divide size, usually 25, then `lrwork` need only be $\max(1, 2*(n-1))$.

If $lrwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for the required value of *lrwork*.

iwork INTEGER. Workspace array, its dimension $\max(1, liwork)$.

liwork INTEGER. The dimension of the array *iwork*.

If $compz = 'N'$, or $n \leq 1$, *liwork* must be at least 1.

If $compz = 'V'$ and $n > 1$, *liwork* must be at least $(6+6*n+5*n*\lg(n))$, where $\lg(n)$ is the smallest integer k such that $2^{**k} \geq n$.

If $compz = 'I'$ and $n > 1$, *liwork* must be at least $(3+5*n)$.

Note that for $compz = 'V'$ or $'I'$, and if n is less than or equal to the minimum divide size, usually 25, then *liwork* need only be 1.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for the required value of *liwork*.

Output Parameters

d The n eigenvalues in ascending order, unless *info* $\neq 0$. See also *info*.

e On exit, the array is overwritten; see *info*.

z If *info* = 0, then if $compz = 'V'$, *z* contains the orthonormal eigenvectors of the original symmetric/Hermitian matrix, and if $compz = 'I'$, *z* contains the orthonormal eigenvectors of the symmetric tridiagonal matrix. If $compz = 'N'$, *z* is not referenced.

work(1) On exit, if *info* = 0, then *work*(1) returns the optimal *lwork*.

<i>rwork(1)</i>	On exit, if <i>info</i> = 0, then <i>rwork(1)</i> returns the optimal <i>lrwork</i> (for complex flavors only).
<i>iwork(1)</i>	On exit, if <i>info</i> = 0, then <i>iwork(1)</i> returns the optimal <i>liwork</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns <i>i</i> /(<i>n</i> +1) through mod(<i>i</i> , <i>n</i> +1).

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `stedc` interface are the following:

<i>d</i>	Holds the vector of length (<i>n</i>).
<i>e</i>	Holds the vector of length (<i>n</i> -1).
<i>z</i>	Holds the matrix <i>z</i> of size (<i>n</i> , <i>n</i>).
<i>compz</i>	If omitted, this argument is restored based on the presence of argument <i>z</i> as follows: <i>compz</i> = 'I', if <i>z</i> is present, <i>compz</i> = 'N', if <i>z</i> is omitted. If present, <i>compz</i> must be equal to 'I' or 'V' and the argument <i>z</i> must also be present. Note that there will be an error condition if <i>compz</i> is present and <i>z</i> omitted.

Note that two variants of Fortran 95 interface for `stedc` routine are needed because of an ambiguous choice between real and complex cases appear when *z* and *work* are omitted. Thus, the name `rstedc` is used in real cases (single or double precision), and the name `stedc` is used in complex cases (single or double precision).

Application Notes

The required size of workspace arrays must be as follows.

For `sstedc/dstedc`:

If *compz* = 'N' or *n* ≤ 1 then *lwork* must be at least 1.

If `compz = 'V'` and $n > 1$ then `lwork` must be at least $(1 + 3n + 2n \cdot \lg n + 3n^2)$, where $\lg(n)$ = smallest integer k such that $2^k \geq n$.

If `compz = 'I'` and $n > 1$ then `lwork` must be at least $(1 + 4n + n^2)$.

If `compz = 'N'` or $n \leq 1$ then `liwork` must be at least 1.

If `compz = 'V'` and $n > 1$ then `liwork` must be at least $(6 + 6n + 5n \cdot \lg n)$.

If `compz = 'I'` and $n > 1$ then `liwork` must be at least $(3 + 5n)$.

For `cstedc/zstedc`:

If `compz = 'N'` or `'I'`, or $n \leq 1$, `lwork` must be at least 1.

If `compz = 'V'` and $n > 1$, `lwork` must be at least n^2 .

If `compz = 'N'` or $n \leq 1$, `lrwork` must be at least 1.

If `compz = 'V'` and $n > 1$, `lrwork` must be at least $(1 + 3n + 2n \cdot \lg n + 3n^2)$, where $\lg(n)$ = smallest integer k such that $2^k \geq n$.

If `compz = 'I'` and $n > 1$, `lrwork` must be at least $(1 + 4n + 2n^2)$.

The required value of `liwork` for complex flavors is the same as for real flavors.

If `lwork` (or `liwork` or `lrwork`, if supplied) is equal to -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`, `iwork`, `rwork`). This operation is called a workspace query.

Note that if `lwork` (`liwork`, `lrwork`) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?stegr

Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

Syntax

FORTRAN 77:

```
call sstegr(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz,
work, lwork, iwork, liwork, info)

call dstegr(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz,
work, lwork, iwork, liwork, info)

call cstegr(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz,
work, lwork, iwork, liwork, info)

call zstegr(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz,
work, lwork, iwork, liwork, info)
```

Fortran 95:

```
call rstegr(d, e, w [,z] [,vl] [,vu] [,il] [,iu] [,m] [,isuppz] [,abstol]
[,info])

call stegr(d, e, w [,z] [,vl] [,vu] [,il] [,iu] [,m] [,isuppz] [,abstol]
[,info])
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix T . Any such unreduced matrix has a well defined set of pairwise different real eigenvalues, the corresponding real eigenvectors are pairwise orthogonal.

The spectrum may be computed either completely or partially by specifying either an interval $[vl, vu]$ or a range of indices $il:iu$ for the desired eigenvalues.

?sregr is a compatibility wrapper around the improved ?stemr routine. See its description for further details.

Note that the *abstol* parameter no longer provides any benefit and hence is no longer used.

See also auxiliary ?lasq2 ?lasq5, ?lasq6 , used by this routine.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues $\lambda(i)$ in the half-open interval: $vl < \lambda(i) \leq vu.$ If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i>.
<i>n</i>	INTEGER. The order of the matrix T ($n \geq 0$).
<i>d, e, work</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays: <i>d</i>(*) contains the diagonal elements of T. The dimension of <i>d</i> must be at least $\max(1, n)$. <i>e</i>(*) contains the subdiagonal elements of T in elements 1 to $n-1$; <i>e</i>(<i>n</i>) need not be set on input, but it is used as a workspace. The dimension of <i>e</i> must be at least $\max(1, n)$. <i>work</i>(<i>lwork</i>) is a workspace array.
<i>vl, vu</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: $vl < vu$. If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il, iu</i>	INTEGER. If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$. If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.
<i>abstol</i>	REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.
 Unused. Was the absolute error tolerance for the eigenvalues/eigenvectors in previous versions.

ldz INTEGER. The leading dimension of the output array *z*.
 Constraints:
ldz < 1 if *jobz* = 'N';
ldz < max(1, *n*) if *jobz* = 'V', an.

lwork INTEGER.
 The dimension of the array *work*,
lwork ≥ max(1, 18**n*) if *jobz* = 'V', and
lwork ≥ max(1, 12**n*) if *jobz* = 'N'.
 If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#). See *Application Notes* below for details.

iwork INTEGER.
 Workspace array, DIMENSION (*liwork*).

liwork INTEGER.
 The dimension of the array *iwork*, *lwork* ≥ max(1, 10**n*) if the eigenvectors are desired, and *lwork* ≥ max(1, 8**n*) if only the eigenvalues are to be computed..
 If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *iwork* array, returns this value as the first entry of the *iwork* array, and no error message related to *liwork* is issued by [xerbla](#). See *Application Notes* below for details.

Output Parameters

d, *e* On exit, *d* and *e* are overwritten.

m INTEGER. The total number of eigenvalues found,
 $0 \leq m \leq n$.
 If *range* = 'A', *m* = *n*, and if *range* = 'I', *m* = *iu-il*+1.

w REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.

	Array, DIMENSION at least $\max(1, n)$. The selected eigenvalues in ascending order, stored in $w(1)$ to $w(m)$.
z	REAL for <code>sstegr</code> DOUBLE PRECISION for <code>dstegr</code> COMPLEX for <code>cstegr</code> DOUBLE COMPLEX for <code>zstegr</code>. Array $z(ldz, *)$, the second dimension of z must be at least $\max(1, m)$. If $jobz = 'V'$, and if $info = 0$, the first m columns of z contain the orthonormal eigenvectors of the matrix T corresponding to the selected eigenvalues, with the i-th column of z holding the eigenvector associated with $w(i)$. If $jobz = 'N'$, then z is not referenced. Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array z; if $range = 'V'$, the exact value of m is not known in advance and an upper bound must be used. Supplying n columns is always safe.
$isuppz$	INTEGER. Array, DIMENSION at least $(2 * \max(1, m))$. The support of the eigenvectors in z, that is the indices indicating the nonzero elements in z. The i-th computed eigenvector is nonzero only in elements $isuppz(2*i-1)$ through $isuppz(2*i)$. This is relevant in the case when the matrix is split. $isuppz$ is only accessed when $jobz = 'V'$, and $n > 0$.
$work(1)$	On exit, if $info = 0$, then $work(1)$ returns the required minimal size of $lwork$.
$iwork(1)$	On exit, if $info = 0$, then $iwork(1)$ returns the required minimal size of $liwork$.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i-th parameter had an illegal value. If $info = 1x$, internal error in <code>?larre</code> occurred, If $info = 2x$, internal error in <code>?larrv</code> occurred. Here the digit $x = \text{abs}(iinfo) < 10$, where $iinfo$ is the non-zero error code returned by <code>?larre</code> or <code>?larrv</code>, respectively.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `stegr` interface are the following:

<code>d</code>	Holds the vector of length (n) .
<code>e</code>	Holds the vector of length (n) .
<code>w</code>	Holds the vector of length (n) .
<code>z</code>	Holds the matrix z of size (n, m) .
<code>isuppz</code>	Holds the vector of length $(2*m)$.
<code>vl</code>	Default value for this argument is <code>vl = - HUGE (vl)</code> where <code>HUGE(a)</code> means the largest machine number of the same precision as argument <code>a</code> .
<code>vu</code>	Default value for this argument is <code>vu = HUGE (vl)</code> .
<code>il</code>	Default value for this argument is <code>il = 1</code> .
<code>iu</code>	Default value for this argument is <code>iu = n</code> .
<code>abstol</code>	Default value for this argument is <code>abstol = 0.0_WP</code> .
<code>jobz</code>	Restored based on the presence of the argument <code>z</code> as follows: <code>jobz = 'V'</code> , if <code>z</code> is present, <code>jobz = 'N'</code> , if <code>z</code> is omitted.
<code>range</code>	Restored based on the presence of arguments <code>vl</code> , <code>vu</code> , <code>il</code> , <code>iu</code> as follows: <code>range = 'V'</code> , if one of or both <code>vl</code> and <code>vu</code> are present, <code>range = 'I'</code> , if one of or both <code>il</code> and <code>iu</code> are present, <code>range = 'A'</code> , if none of <code>vl</code> , <code>vu</code> , <code>il</code> , <code>iu</code> is present, Note that there will be an error condition if one of or both <code>vl</code> and <code>vu</code> are present and at the same time one of or both <code>il</code> and <code>iu</code> are present.

Note that two variants of Fortran 95 interface for `stegr` routine are needed because of an ambiguous choice between real and complex cases appear when `z` is omitted. Thus, the name `rstegr` is used in real cases (single or double precision), and the name `stegr` is used in complex cases (single or double precision).

Application Notes

Currently `?stegr` is only set up to find *all* the n eigenvalues and eigenvectors of T in $O(n^2)$ time, that is, only `range = 'A'` is supported.

?stegr works only on machines which follow IEEE-754 floating-point standard in their handling of infinities and NaNs. Normal execution of ?stegr may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not conform to the IEEE-754 standard.

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run, or set *lwork* = -1 (*liwork* = -1).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If *lwork* = -1 (*liwork* = -1), then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *lwork* (*liwork*) is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?pteqr

Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric positive-definite tridiagonal matrix.

Syntax

FORTRAN 77:

```
call spteqr(compz, n, d, e, z, ldz, work, info)
call dpteqr(compz, n, d, e, z, ldz, work, info)
call cpteqr(compz, n, d, e, z, ldz, work, info)
call zpteqr(compz, n, d, e, z, ldz, work, info)
```

Fortran 95:

```
call rpteqr(d, e [,z] [,compz] [,info])
call pteqr(d, e [,z] [,compz] [,info])
```

Description

This routine computes all the eigenvalues and (optionally) all the eigenvectors of a real symmetric positive-definite tridiagonal matrix T . In other words, the routine can compute the spectral factorization: $T = Z^* \Lambda^* Z^T$.

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i ; Z is an orthogonal matrix whose columns are eigenvectors. Thus,

$$T^* z_i = \lambda_i^* z_i \text{ for } i = 1, 2, \dots, n.$$

(The routine normalizes the eigenvectors so that $\|z_i\|_2 = 1$.)

You can also use the routine for computing the eigenvalues and eigenvectors of real symmetric (or complex Hermitian) positive-definite matrices A reduced to tridiagonal form T : $A = Q^* T^* Q^H$.

In this case, the spectral factorization is as follows: $A = Q^* T^* Q^H = (QZ)^* \Lambda^* (QZ)^H$. Before calling `?pteqr`, you must reduce A to tridiagonal form and generate the explicit matrix Q by calling the following routines:

	for real matrices:	for complex matrices:
full storage	<code>?sytrd, ?orgtr</code>	<code>?hetrd, ?ungtr</code>
packed storage	<code>?sptrd, ?opgtr</code>	<code>?hptrd, ?upgtr</code>
band storage	<code>?sbtrd (vect='V')</code>	<code>?hbtrd (vect='V')</code>

The routine first factorizes T as $L^* D^* L^H$ where L is a unit lower bidiagonal matrix, and D is a diagonal matrix. Then it forms the bidiagonal matrix $B = L^* D^{1/2}$ and calls `?bdsqr` to compute the singular values of B , which are the same as the eigenvalues of T .

Input Parameters

compz CHARACTER*1. Must be 'N' or 'I' or 'V'.
 If *compz* = 'N', the routine computes eigenvalues only.
 If *compz* = 'I', the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix T .
 If *compz* = 'V', the routine computes the eigenvalues and eigenvectors of A (and the array z must contain the matrix Q on entry).

n	INTEGER. The order of the matrix T ($n \geq 0$).
$d, e, work$	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays: $d(*)$ contains the diagonal elements of T . The dimension of d must be at least $\max(1, n)$. $e(*)$ contains the off-diagonal elements of T . The dimension of e must be at least $\max(1, n-1)$. $work(*)$ is a workspace array. The dimension of $work$ must be: at least 1 if $compz = 'N'$; at least $\max(1, 4*n-4)$ if $compz = 'V'$ or $'I'$.
z	REAL for <code>spteqr</code> DOUBLE PRECISION for <code>dpteqr</code> COMPLEX for <code>cpteqr</code> DOUBLE COMPLEX for <code>zpteqr</code> . Array, DIMENSION ($ldz, *$) If $compz = 'N'$ or $'I'$, z need not be set. If $vect = 'V'$, z must contain the n -by- n matrix Q . The second dimension of z must be: at least 1 if $compz = 'N'$; at least $\max(1, n)$ if $compz = 'V'$ or $'I'$.
ldz	INTEGER. The first dimension of z . Constraints: $ldz \geq 1$ if $compz = 'N'$; $ldz \geq \max(1, n)$ if $compz = 'V'$ or $'I'$.

Output Parameters

d	The n eigenvalues in descending order, unless $info > 0$. See also $info$.
e	On exit, the array is overwritten.
z	If $info = 0$, contains the n orthonormal eigenvectors, stored by columns. (The i -th column corresponds to the i -th eigenvalue.)
$info$	INTEGER. If $info = 0$, the execution is successful.

If $info = i$, the leading minor of order i (and hence T itself) is not positive-definite.

If $info = n + i$, the algorithm for computing singular values failed to converge; i off-diagonal elements have not converged to zero.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `pteqr` interface are the following:

d	Holds the vector of length (n) .
e	Holds the vector of length $(n-1)$.
z	Holds the matrix Z of size (n, n) .
$compz$	If omitted, this argument is restored based on the presence of argument z as follows: $compz = 'I'$, if z is present, $compz = 'N'$, if z is omitted. If present, $compz$ must be equal to 'I' or 'V' and the argument z must also be present. Note that there will be an error condition if $compz$ is present and z omitted.

Note that two variants of Fortran 95 interface for `pteqr` routine are needed because of an ambiguous choice between real and complex cases appear when z is omitted. Thus, the name `rpteqr` is used in real cases (single or double precision), and the name `pteqr` is used in complex cases (single or double precision).

Application Notes

If λ_i is an exact eigenvalue, and μ_i is the corresponding computed value, then

$$|\mu_i - \lambda_i| \leq c(n) * \epsilon * K * \lambda_i$$

where $c(n)$ is a modestly increasing function of n , ϵ is the machine precision, and $K = ||DTD||_2 * ||(DTD)^{-1}||_2$, D is diagonal with $d_{ii} = t_{ii}^{-1/2}$.

If z_i is the corresponding exact eigenvector, and w_i is the corresponding computed vector, then the angle $\theta(z_i, w_i)$ between them is bounded as follows:

$$\theta(u_i, w_i) \leq c(n) \varepsilon_K / \min_{i \neq j} (|\lambda_i - \lambda_j| / |\lambda_i + \lambda_j|).$$

Here $\min_{i \neq j} (|\lambda_i - \lambda_j| / |\lambda_i + \lambda_j|)$ is the *relative gap* between λ_i and the other eigenvalues.

The total number of floating-point operations depends on how rapidly the algorithm converges.

Typically, it is about

$30n^2$ if `compz = 'N'`;

$6n^3$ (for complex flavors, $12n^3$) if `compz = 'V' or 'I'`.

?stebz

Computes selected eigenvalues of a real symmetric tridiagonal matrix by bisection.

Syntax

FORTRAN 77:

```
call sstebz (range, order, n, vl, vu, il, iu, abstol, d, e, m, nsplit, w,
            iblock, isplit, work, iwork, info)
```

```
call dstebz (range, order, n, vl, vu, il, iu, abstol, d, e, m, nsplit, w,
            iblock, isplit, work, iwork, info)
```

Fortran 95:

```
call stebz(d, e, m, nsplit, w, iblock, isplit [, order] [,vl] [,vu] [,il]
[,iu] [,abstol] [,info])
```

Description

This routine computes some (or all) of the eigenvalues of a real symmetric tridiagonal matrix T by bisection. The routine searches for zero or negligible off-diagonal elements to see if T splits into block-diagonal form $T = \text{diag}(T_1, T_2, \dots)$. Then it performs bisection on each of the blocks T_i and returns the block index of each computed eigenvalue, so that a subsequent call to [?stein](#) can also take advantage of the block structure.

See also [?laebz](#).

Input Parameters

<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues $\lambda(i)$ in the half-open interval: $vl < \lambda(i) \leq vu$. If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i>.
<i>order</i>	CHARACTER*1. Must be 'B' or 'E'. If <i>order</i> = 'B', the eigenvalues are to be ordered from smallest to largest within each split-off block. If <i>order</i> = 'E', the eigenvalues for the entire matrix are to be ordered from smallest to largest.
<i>n</i>	INTEGER. The order of the matrix T ($n \geq 0$).
<i>vl, vu</i>	REAL for sstebz DOUBLE PRECISION for dstebz. If <i>range</i> = 'V', the routine computes eigenvalues $\lambda(i)$ in the half-open interval: $vl < \lambda(i) \leq vu$. If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il, iu</i>	INTEGER. Constraint: $1 \leq il \leq iu \leq n$. If <i>range</i> = 'I', the routine computes eigenvalues $\lambda(i)$ such that $il \leq i \leq iu$ (assuming that the eigenvalues $\lambda(i)$ are in ascending order). If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.
<i>abstol</i>	REAL for sstebz DOUBLE PRECISION for dstebz. The absolute tolerance to which each eigenvalue is required. An eigenvalue (or cluster) is considered to have converged if it lies in an interval of width <i>abstol</i>. If <i>abstol</i> ≤ 0.0, then the tolerance is taken as $\text{eps} * T$, where <i>eps</i> is the machine precision, and T is the 1-norm of the matrix T.
<i>d, e, work</i>	REAL for sstebz

DOUBLE PRECISION for `dstebz`.

Arrays:

`d(*)` contains the diagonal elements of T .

The dimension of d must be at least $\max(1, n)$.

`e(*)` contains the off-diagonal elements of T .

The dimension of e must be at least $\max(1, n-1)$.

`work(*)` is a workspace array.

The dimension of `work` must be at least $\max(1, 4n)$.

`iwork`

INTEGER. Workspace.

Array, DIMENSION at least $\max(1, 3n)$.

Output Parameters

`m`

INTEGER. The actual number of eigenvalues found.

`nsplit`

INTEGER. The number of diagonal blocks detected in T .

`w`

REAL for `sstebz`

DOUBLE PRECISION for `dstebz`.

Array, DIMENSION at least $\max(1, n)$. The computed eigenvalues, stored in $w(1)$ to $w(m)$.

`iblock, isplit`

INTEGER.

Arrays, DIMENSION at least $\max(1, n)$.

A positive value `iblock(i)` is the block number of the eigenvalue stored in $w(i)$ (see also `info`).

The leading `nsplit` elements of `isplit` contain points at which T splits into blocks T_i as follows: the block T_1 contains rows/columns 1 to `isplit(1)`; the block T_2 contains rows/columns `isplit(1)+1` to `isplit(2)`, and so on.

`info`

INTEGER.

If `info = 0`, the execution is successful.

If `info = 1`, for `range = 'A'` or `'V'`, the algorithm failed to compute some of the required eigenvalues to the desired accuracy; `iblock(i)<0` indicates that the eigenvalue stored in $w(i)$ failed to converge.

If `info = 2`, for `range = 'I'`, the algorithm failed to compute some of the required eigenvalues. Try calling the routine again with `range = 'A'`.

If `info = 3`:

for *range* = 'A' or 'V', same as *info* = 1;
 for *range* = 'I', same as *info* = 2.
 If *info* = 4, no eigenvalues have been computed. The
 floating-point arithmetic on the computer is not behaving
 as expected.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *stebz* interface are the following:

<i>d</i>	Holds the vector of length (<i>n</i>).
<i>e</i>	Holds the vector of length (<i>n</i> -1).
<i>w</i>	Holds the vector of length (<i>n</i>).
<i>iblock</i>	Holds the vector of length (<i>n</i>).
<i>isplit</i>	Holds the vector of length (<i>n</i>).
<i>order</i>	Must be 'B' or 'E'. The default value is 'B'.
<i>vl</i>	Default value for this argument is <i>vl</i> = - HUGE (<i>vl</i>) where HUGE(<i>a</i>) means the largest machine number of the same precision as argument <i>a</i> .
<i>vu</i>	Default value for this argument is <i>vu</i> = HUGE (<i>vl</i>).
<i>il</i>	Default value for this argument is <i>il</i> = 1.
<i>iu</i>	Default value for this argument is <i>iu</i> = <i>n</i> .
<i>abstol</i>	Default value for this argument is <i>abstol</i> = 0.0_WP.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: <i>range</i> = 'V', if one of or both <i>vl</i> and <i>vu</i> are present, <i>range</i> = 'I', if one of or both <i>il</i> and <i>iu</i> are present, <i>range</i> = 'A', if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present, Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

The eigenvalues of T are computed to high relative accuracy which means that if they vary widely in magnitude, then any small eigenvalues will be computed more accurately than, for example, with the standard QR method. However, the reduction to tridiagonal form (prior to calling the routine) may exclude the possibility of obtaining high relative accuracy in the small eigenvalues of the original matrix if its eigenvalues vary widely in magnitude.

?stein

Computes the eigenvectors corresponding to specified eigenvalues of a real symmetric tridiagonal matrix.

Syntax

FORTRAN 77:

```
call sstein(n, d, e, m, w, iblock, isplit, z, ldz, work, iwork, ifailv, info)
call dstein(n, d, e, m, w, iblock, isplit, z, ldz, work, iwork, ifailv, info)
call cstein(n, d, e, m, w, iblock, isplit, z, ldz, work, iwork, ifailv, info)
call zstein(n, d, e, m, w, iblock, isplit, z, ldz, work, iwork, ifailv, info)
```

Fortran 95:

```
call stein(d, e, w, iblock, isplit, z [,ifailv] [,info])
```

Description

This routine computes the eigenvectors of a real symmetric tridiagonal matrix T corresponding to specified eigenvalues, by inverse iteration. It is designed to be used in particular after the specified eigenvalues have been computed by ?stebz with `order = 'B'`, but may also be used when the eigenvalues have been computed by other routines.

If you use this routine after ?stebz, it can take advantage of the block structure by performing inverse iteration on each block T_i separately, which is more efficient than using the whole matrix T .

If T has been formed by reduction of a full symmetric or Hermitian matrix A to tridiagonal form, you can transform eigenvectors of T to eigenvectors of A by calling ?ormtr or ?opmtr (for real flavors) or by calling ?unmtr or ?upmtr (for complex flavors).

Input Parameters

<i>n</i>	INTEGER. The order of the matrix T ($n \geq 0$).
<i>m</i>	INTEGER. The number of eigenvectors to be returned.
<i>d</i> , <i>e</i> , <i>w</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays: <i>d</i> (*) contains the diagonal elements of T . The dimension of <i>d</i> must be at least $\max(1, n)$. <i>e</i> (*) contains the sub-diagonal elements of T stored in elements 1 to $n-1$. The dimension of <i>e</i> must be at least $\max(1, n-1)$. <i>w</i> (*) contains the eigenvalues of T , stored in $w(1)$ to $w(m)$ (as returned by ?stebz). Eigenvalues of T_1 must be supplied first, in non-decreasing order; then those of T_2 , again in non-decreasing order, and so on. Constraint: if $iblock(i) = iblock(i+1)$, $w(i) \leq w(i+1)$. The dimension of <i>w</i> must be at least $\max(1, n)$.
<i>iblock</i> , <i>isplit</i>	INTEGER. Arrays, DIMENSION at least $\max(1, n)$. The arrays <i>iblock</i> and <i>isplit</i> , as returned by ?stebz with <i>order</i> = 'B'. If you did not call ?stebz with <i>order</i> = 'B', set all elements of <i>iblock</i> to 1, and <i>isplit</i> (1) to <i>n</i> .)
<i>ldz</i>	INTEGER. The first dimension of the output array <i>z</i> ; $ldz \geq \max(1, n)$.
<i>work</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Workspace array, DIMENSION at least $\max(1, 5n)$.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>z</i>	REAL for sstein DOUBLE PRECISION for dstein COMPLEX for cstein

DOUBLE COMPLEX for `zstein`.
 Array, DIMENSION (`ldz`, *).
 If `info = 0`, `z` contains the m orthonormal eigenvectors, stored by columns. (The i th column corresponds to the i -th specified eigenvalue.)

`ifailv` INTEGER.
 Array, DIMENSION at least $\max(1, m)$.
 If `info = i > 0`, the first i elements of `ifailv` contain the indices of any eigenvectors that failed to converge.

`info` INTEGER.
 If `info = 0`, the execution is successful.
 If `info = i`, then i eigenvectors (as indicated by the parameter `ifailv`) each failed to converge in 5 iterations. The current iterates are stored in the corresponding columns of the array `z`.
 If `info = -i`, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `stein` interface are the following:

<code>d</code>	Holds the vector of length (n).
<code>e</code>	Holds the vector of length (n).
<code>w</code>	Holds the vector of length (n).
<code>iblock</code>	Holds the vector of length (n).
<code>isplit</code>	Holds the vector of length (n).
<code>z</code>	Holds the matrix Z of size (n, m).
<code>ifailv</code>	Holds the vector of length (m).

Application Notes

Each computed eigenvector z_i is an exact eigenvector of a matrix $T + E_i$, where $\|E_i\|_2 = O(\epsilon) * \|T\|_2$. However, a set of eigenvectors computed by this routine may not be orthogonal to so high a degree of accuracy as those computed by `?steqr`.

?disna

Computes the reciprocal condition numbers for the eigenvectors of a symmetric/ Hermitian matrix or for the left or right singular vectors of a general matrix.

Syntax

FORTTRAN 77:

```
call sdisna(job, m, n, d, sep, info)
call ddisna(job, m, n, d, sep, info)
```

Fortran 95:

```
call disna(d, sep [,job] [,minmn] [,info])
```

Description

This routine computes the reciprocal condition numbers for the eigenvectors of a real symmetric or complex Hermitian matrix or for the left or right singular vectors of a general m -by- n matrix.

The reciprocal condition number is the 'gap' between the corresponding eigenvalue or singular value and the nearest other one.

The bound on the error, measured by angle in radians, in the i -th computed vector is given by

```
slamch('E')*(anorm/sep(i))
```

where $anorm = ||A||_2 = \max(|d(j)|)$. $sep(i)$ is not allowed to be smaller than $slamch('E')*anorm$ in order to limit the size of the error bound.

?disna may also be used to compute error bounds for eigenvectors of the generalized symmetric definite eigenproblem.

Input Parameters

<i>job</i>	CHARACTER*1. Must be 'E', 'L', or 'R'. Specifies for which problem the reciprocal condition numbers should be computed: <i>job</i> = 'E': for the eigenvectors of a symmetric/Hermitian matrix; <i>job</i> = 'L': for the left singular vectors of a general matrix;

	<i>job</i> = 'R': for the right singular vectors of a general matrix.
<i>m</i>	INTEGER. The number of rows of the matrix ($m \geq 0$).
<i>n</i>	INTEGER. If <i>job</i> = 'L', or 'R', the number of columns of the matrix ($n \geq 0$). Ignored if <i>job</i> = 'E'.
<i>d</i>	REAL for sdisna DOUBLE PRECISION for ddisna. Array, dimension at least $\max(1, m)$ if <i>job</i> = 'E', and at least $\max(1, \min(m, n))$ if <i>job</i> = 'L' or 'R'. This array must contain the eigenvalues (if <i>job</i> = 'E') or singular values (if <i>job</i> = 'L' or 'R') of the matrix, in either increasing or decreasing order. If singular values, they must be non-negative.

Output Parameters

<i>sep</i>	REAL for sdisna DOUBLE PRECISION for ddisna. Array, dimension at least $\max(1, m)$ if <i>job</i> = 'E', and at least $\max(1, \min(m, n))$ if <i>job</i> = 'L' or 'R'. The reciprocal condition numbers of the vectors.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `disna` interface are the following:

<i>d</i>	Holds the vector of length $\min(m, n)$.
<i>sep</i>	Holds the vector of length $\min(m, n)$.
<i>job</i>	Must be 'E', 'L', or 'R'. The default value is 'E'.
<i>minmn</i>	Indicates which of the values <i>m</i> or <i>n</i> is smaller. Must be either 'M' or 'N', the default is 'M'.

If $job = 'E'$, this argument is superfluous, If $job = 'L'$ or $'R'$, this argument is used by the routine.

Generalized Symmetric-Definite Eigenvalue Problems

Generalized symmetric-definite eigenvalue problems are as follows: find the eigenvalues λ and the corresponding eigenvectors z that satisfy one of these equations:

$$Az = \lambda Bz, ABz = \lambda z, \text{ or } BAz = \lambda z,$$

where A is an n -by- n symmetric or Hermitian matrix, and B is an n -by- n symmetric positive-definite or Hermitian positive-definite matrix.

In these problems, there exist n real eigenvectors corresponding to real eigenvalues (even for complex Hermitian matrices A and B).

Routines described in this section allow you to reduce the above generalized problems to standard symmetric eigenvalue problem $Cy = \lambda y$, which you can solve by calling LAPACK routines described earlier in this chapter (see [Symmetric Eigenvalue Problems](#)).

Different routines allow the matrices to be stored either conventionally or in packed storage. Prior to reduction, the positive-definite matrix B must first be factorized using either [?potrf](#) or [?pptrf](#).

The reduction routine for the banded matrices A and B uses a split Cholesky factorization for which a specific routine [?pbstf](#) is provided. This refinement halves the amount of work required to form matrix C .

[Table 4-4](#) lists LAPACK routines (Fortran 77 interface) that can be used to solve generalized symmetric-definite eigenvalue problems. Respective routine names in Fortran 95 interface are without the first symbol (see [Routine Naming Conventions](#)).

Table 4-4 Computational Routines for Reducing Generalized Eigenproblems to Standard Problems

Matrix type	Reduce to standard problems (full storage)	Reduce to standard problems (packed storage)	Reduce to standard problems (band matrices)	Factorize band matrix
real symmetric matrices	?sygst	?spgst	?sbgst	?pbstf
complex Hermitian matrices	?hegst	?hpgst	?hbgst	?pbstf

?sygst

Reduces a real symmetric-definite generalized eigenvalue problem to the standard form.

Syntax

FORTRAN 77:

```
call ssygst(itype, uplo, n, a, lda, b, ldb, info)
call dsygst(itype, uplo, n, a, lda, b, ldb, info)
```

Fortran 95:

```
call sygst(a, b [,itype] [,uplo] [,info])
```

Description

This routine reduces real symmetric-definite generalized eigenproblems

$$A^*z = \lambda^*B^*z, \quad A^*B^*z = \lambda^*z, \quad \text{or} \quad B^*A^*z = \lambda^*z$$

to the standard form $C^*y = \lambda^*y$. Here A is a real symmetric matrix, and B is a real symmetric positive-definite matrix. Before calling this routine, call ?potrf to compute the Cholesky factorization: $B = U^T * U$ or $B = L^* L^T$.

Input Parameters

itype INTEGER. Must be 1 or 2 or 3.
 If *itype* = 1, the generalized eigenproblem is $A^*z = \lambda^*B^*z$
 for *uplo* = 'U': $C = \text{inv}(U^T) * A * \text{inv}(U)$, $z = \text{inv}(U) * y$;
 for *uplo* = 'L': $C = \text{inv}(L) * A * \text{inv}(L^T)$, $z = \text{inv}(L^T) * y$.
 If *itype* = 2, the generalized eigenproblem is $A^*B^*z = \lambda^*z$
 for *uplo* = 'U': $C = U^* A^* U^T$, $z = \text{inv}(U) * y$;
 for *uplo* = 'L': $C = L^T * A^* L$, $z = \text{inv}(L^T) * y$.
 If *itype* = 3, the generalized eigenproblem is $B^*A^*z = \lambda^*z$
 for *uplo* = 'U': $C = U^* A^* U^T$, $z = U^T * y$;
 for *uplo* = 'L': $C = L^T * A^* L$, $z = L^* y$.

uplo CHARACTER*1. Must be 'U' or 'L'.
 If *uplo* = 'U', the array *a* stores the upper triangle of *A*;
 you must supply *B* in the factored form $B = U^T * U$.
 If *uplo* = 'L', the array *a* stores the lower triangle of *A*;
 you must supply *B* in the factored form $B = L * L^T$.

n INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).

a, *b* REAL for ssygst
 DOUBLE PRECISION for dsygst.
 Arrays:
a(*lda*,*) contains the upper or lower triangle of *A*.
 The second dimension of *a* must be at least $\max(1, n)$.
b(*ldb*,*) contains the Cholesky-factored matrix *B*:
 $B = U^T * U$ or $B = L * L^T$ (as returned by ?potrf).
 The second dimension of *b* must be at least $\max(1, n)$.

lda INTEGER. The first dimension of *a*; at least $\max(1, n)$.

ldb INTEGER. The first dimension of *b*; at least $\max(1, n)$.

Output Parameters

a The upper or lower triangle of *A* is overwritten by the upper or lower triangle of *C*, as specified by the arguments *itype* and *uplo*.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *sygst* interface are the following:

a Holds the matrix *A* of size (*n*,*n*).

b Holds the matrix *B* of size (*n*,*n*).

itype Must be 1, 2, or 3. The default value is 1.

uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

Forming the reduced matrix C is a stable procedure. However, it involves implicit multiplication by $\text{inv}(B)$ (if $itype = 1$) or B (if $itype = 2$ or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if B is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is n^3 .

?hegst

Reduces a complex Hermitian-definite generalized eigenvalue problem to the standard form.

Syntax

FORTRAN 77:

```
call chegst(itype, uplo, n, a, lda, b, ldb, info)
call zhegst(itype, uplo, n, a, lda, b, ldb, info)
```

Fortran 95:

```
call hegst(a, b [,itype] [,uplo] [,info])
```

Description

This routine reduces complex Hermitian-definite generalized eigenvalue problems

$$A^*x = \lambda^*B^*x, \quad A^*B^*x = \lambda^*x, \quad \text{or} \quad B^*A^*x = \lambda^*x.$$

to the standard form $Cy = \lambda y$. Here the matrix A is complex Hermitian, and B is complex Hermitian positive-definite. Before calling this routine, you must call ?potrf to compute the Cholesky factorization: $B = U^H * U$ or $B = L * L^H$.

Input Parameters

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. If <i>itype</i> = 1, the generalized eigenproblem is $A^*z = \lambda^*B^*z$ for <i>uplo</i> = 'U': $C = \text{inv}(U^H) * A * \text{inv}(U)$, $z = \text{inv}(U) * y$; for <i>uplo</i> = 'L': $C = \text{inv}(L) * A * \text{inv}(L^H)$, $z = \text{inv}(L^H) * y$.

If $itype = 2$, the generalized eigenproblem is $A^*B^*z = \lambda z$
 for $uplo = 'U'$: $C = U^*A^*U^H$, $z = \text{inv}(U)^*y$;
 for $uplo = 'L'$: $C = L^H^*A^*L$, $z = \text{inv}(L^H)^*y$.
 If $itype = 3$, the generalized eigenproblem is $B^*A^*z = \lambda z$
 for $uplo = 'U'$: $C = U^*A^*U^H$, $z = U^H^*y$;
 for $uplo = 'L'$: $C = L^H^*A^*L$, $z = L^*y$.
 CHARACTER*1. Must be 'U' or 'L'.
 If $uplo = 'U'$, the array a stores the upper triangle of A ;
 you must supply B in the factored form $B = U^H^*U$.
 If $uplo = 'L'$, the array a stores the lower triangle of A ;
 you must supply B in the factored form $B = L^*L^H$.
 INTEGER. The order of the matrices A and B ($n \geq 0$).
 COMPLEX for chegstDOUBLE COMPLEX for zhegst.
 Arrays:
 $a(lda,*)$ contains the upper or lower triangle of A .
 The second dimension of a must be at least $\max(1, n)$.
 $b(l db,*)$ contains the Cholesky-factored matrix B :
 $B = U^H^*U$ or $B = L^*L^H$ (as returned by ?potrf).
 The second dimension of b must be at least $\max(1, n)$.
 INTEGER. The first dimension of a ; at least $\max(1, n)$.
 INTEGER. The first dimension of b ; at least $\max(1, n)$.

Output Parameters

a The upper or lower triangle of A is overwritten by the upper or lower triangle of C , as specified by the arguments $itype$ and $uplo$.
 INTEGER.
 If $info = 0$, the execution is successful.
 If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hegst` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (n, n) .
<i>b</i>	Holds the matrix <i>B</i> of size (n, n) .
<i>itype</i>	Must be 1, 2, or 3. The default value is 1.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

Forming the reduced matrix *C* is a stable procedure. However, it involves implicit multiplication by $\text{inv}(B)$ (if *itype* = 1) or *B* (if *itype* = 2 or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if *B* is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is n^3 .

?spgst

Reduces a real symmetric-definite generalized eigenvalue problem to the standard form using packed storage.

Syntax

FORTRAN 77:

```
call sspgst(itype, uplo, n, ap, bp, info)
call dspgst(itype, uplo, n, ap, bp, info)
```

Fortran 95:

```
call spgst(a, b [,itype] [,uplo] [,info])
```

Description

This routine reduces real symmetric-definite generalized eigenproblems

$$A^*x = \lambda^*B^*x, A^*B^*x = \lambda^*x, \text{ or } B^*A^*x = \lambda^*x$$

to the standard form $C^*Y = \lambda^*Y$, using packed matrix storage. Here A is a real symmetric matrix, and B is a real symmetric positive-definite matrix. Before calling this routine, call `?pptrf` to compute the Cholesky factorization: $B = U^T * U$ or $B = L * L^T$.

Input Parameters

itype INTEGER. Must be 1 or 2 or 3.
 If *itype* = 1, the generalized eigenproblem is $A^*z = \lambda^*B^*z$
 for *uplo* = 'U': $C = \text{inv}(U^T) * A * \text{inv}(U)$, $z = \text{inv}(U) * y$;
 for *uplo* = 'L': $C = \text{inv}(L) * A * \text{inv}(L^T)$, $z = \text{inv}(L^T) * y$.
 If *itype* = 2, the generalized eigenproblem is $A^*B^*z = \lambda^*z$
 for *uplo* = 'U': $C = U^T * A * U$, $z = \text{inv}(U) * y$;
 for *uplo* = 'L': $C = L^T * A * L$, $z = \text{inv}(L^T) * y$.
 If *itype* = 3, the generalized eigenproblem is $B^*A^*z = \lambda^*z$
 for *uplo* = 'U': $C = U^T * A * U$, $z = U^T * y$;
 for *uplo* = 'L': $C = L^T * A * L$, $z = L^T * y$.

uplo CHARACTER*1. Must be 'U' or 'L'.
 If *uplo* = 'U', *ap* stores the packed upper triangle of A ;
 you must supply B in the factored form $B = U^T * U$.
 If *uplo* = 'L', *ap* stores the packed lower triangle of A ;
 you must supply B in the factored form $B = L * L^T$.

n INTEGER. The order of the matrices A and B ($n \geq 0$).

ap, *bp* REAL for `sspgst`
 DOUBLE PRECISION for `dspgst`.
 Arrays:
ap(*) contains the packed upper or lower triangle of A .
 The dimension of *ap* must be at least $\max(1, n*(n+1)/2)$.

b *p*(*) contains the packed Cholesky factor of B (as returned
 by `?pptrf` with the same *uplo* value).
 The dimension of *bp* must be at least $\max(1, n*(n+1)/2)$.

Output Parameters

<i>ap</i>	The upper or lower triangle of <i>A</i> is overwritten by the upper or lower triangle of <i>C</i> , as specified by the arguments <i>itype</i> and <i>uplo</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `spgst` interface are the following:

<i>a</i>	Stands for argument <i>ap</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(n * (n + 1) / 2)$.
<i>b</i>	Stands for argument <i>bp</i> in Fortran 77 interface. Holds the array <i>B</i> of size $(n * (n + 1) / 2)$.
<i>itype</i>	Must be 1, 2, or 3. The default value is 1.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

Forming the reduced matrix *C* is a stable procedure. However, it involves implicit multiplication by $\text{inv}(B)$ (if *itype* = 1) or *B* (if *itype* = 2 or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if *B* is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is n^3 .

?hpgst

Reduces a complex Hermitian-definite generalized eigenvalue problem to the standard form using packed storage.

Syntax

FORTRAN 77:

```
call chpgst(itype, uplo, n, ap, bp, info)
call zhpgst(itype, uplo, n, ap, bp, info)
```

Fortran 95:

```
call hpgst(a, b [,itype] [,uplo] [,info])
```

Description

This routine reduces real symmetric-definite generalized eigenproblems

$A^*z = \lambda^*B^*z$, $A^*B^*z = \lambda^*z$, or $B^*A^*z = \lambda^*z$.

to the standard form $C^*y = \lambda^*y$, using packed matrix storage. Here A is a real symmetric matrix, and B is a real symmetric positive-definite matrix. Before calling this routine, you must call ?pptrf to compute the Cholesky factorization: $B = U^H * U$ or $B = L * L^H$.

Input Parameters

itype INTEGER. Must be 1 or 2 or 3.
 If *itype* = 1, the generalized eigenproblem is $A^*z = \lambda^*B^*z$
 for *uplo* = 'U': $C = \text{inv}(U^H) * A * \text{inv}(U)$, $z = \text{inv}(U) * y$;
 for *uplo* = 'L': $C = \text{inv}(L) * A * \text{inv}(L^H)$, $z = \text{inv}(L^H) * y$.
 If *itype* = 2, the generalized eigenproblem is $A^*B^*z = \lambda^*z$
 for *uplo* = 'U': $C = U^*A^*U^H$, $z = \text{inv}(U) * y$;
 for *uplo* = 'L': $C = L^H * A^*L$, $z = \text{inv}(L^H) * y$.
 If *itype* = 3, the generalized eigenproblem is $B^*A^*z = \lambda^*z$
 for *uplo* = 'U': $C = U^*A^*U^H$, $z = U^H * y$;
 for *uplo* = 'L': $C = L^H * A^*L$, $z = L * y$.

uplo CHARACTER*1. Must be 'U' or 'L'.
 If *uplo* = 'U', *ap* stores the packed upper triangle of *A*;
 you must supply *B* in the factored form $B = U^H * U$.
 If *uplo* = 'L', *ap* stores the packed lower triangle of *A*;
 you must supply *B* in the factored form $B = L * L^H$.

n INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).

ap, *bp* COMPLEX for chpgstDOUBLE COMPLEX for zhpgst.
Arrays:
ap(*) contains the packed upper or lower triangle of *A*.
 The dimension of *a* must be at least $\max(1, n*(n+1)/2)$.
bp(*) contains the packed Cholesky factor of *B* (as returned
 by *zpptrf* with the same *uplo* value).
 The dimension of *b* must be at least $\max(1, n*(n+1)/2)$.

Output Parameters

ap The upper or lower triangle of *A* is overwritten by the upper
 or lower triangle of *C*, as specified by the arguments *itype*
 and *uplo*.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *hpgst* interface are the following:

a Stands for argument *ap* in Fortran 77 interface. Holds the array *A* of size
 $(n*(n+1)/2)$.

b Stands for argument *bp* in Fortran 77 interface. Holds the array *B* of size
 $(n*(n+1)/2)$.

itype Must be 1, 2, or 3. The default value is 1.

uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

Forming the reduced matrix C is a stable procedure. However, it involves implicit multiplication by $\text{inv}(B)$ (if $\text{itype} = 1$) or B (if $\text{itype} = 2$ or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if B is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is n^3 .

?sbgst

Reduces a real symmetric-definite generalized eigenproblem for banded matrices to the standard form using the factorization performed by ?pbstf.

Syntax

FORTRAN 77:

```
call ssgbst(vect, uplo, n, ka, kb, ab, ldab, bb, ldbb, x, ldx, work, info)
call dsgbst(vect, uplo, n, ka, kb, ab, ldab, bb, ldbb, x, ldx, work, info)
```

Fortran 95:

```
call sbgst(a, b [,x] [,uplo] [,info])
```

Description

To reduce the real symmetric-definite generalized eigenproblem $A^*z = \lambda^*B^*z$ to the standard form $C^*y = \lambda^*y$, where A , B and C are banded, this routine must be preceded by a call to [spbstf/dpbstf](#), which computes the split Cholesky factorization of the positive-definite matrix B : $B = S^T * S$. The split Cholesky factorization, compared with the ordinary Cholesky factorization, allows the work to be approximately halved.

This routine overwrites A with $C = X^T * A * X$, where $X = \text{inv}(S) * Q$ and Q is an orthogonal matrix chosen (implicitly) to preserve the bandwidth of A . The routine also has an option to allow the accumulation of X , and then, if z is an eigenvector of C , X^*z is an eigenvector of the original system.

Input Parameters

vect CHARACTER*1. Must be 'N' or 'V'.

	If <code>vect = 'N'</code>, then matrix X is not returned; If <code>vect = 'V'</code>, then matrix X is returned.
<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'. If <code>uplo = 'U'</code>, ab stores the upper triangular part of A. If <code>uplo = 'L'</code>, ab stores the lower triangular part of A.
<code>n</code>	INTEGER. The order of the matrices A and B ($n \geq 0$).
<code>ka</code>	INTEGER. The number of super- or sub-diagonals in A ($ka \geq 0$).
<code>kb</code>	INTEGER. The number of super- or sub-diagonals in B ($ka \geq kb \geq 0$).
<code>ab, bb, work</code>	REAL for <code>ssbgst</code> DOUBLE PRECISION for <code>dsbgst</code> $ab(l_{dab},*)$ is an array containing either upper or lower triangular part of the symmetric matrix A (as specified by <code>uplo</code>) in band storage format. The second dimension of the array ab must be at least $\max(1, n)$. $bb(l_{dbb},*)$ is an array containing the banded split Cholesky factor of B as specified by <code>uplo</code>, n and <code>kb</code> and returned by spbstf/dpbstf. The second dimension of the array bb must be at least $\max(1, n)$. $work(*)$ is a workspace array, dimension at least $\max(1, 2*n)$
<code>ldab</code>	INTEGER. The first dimension of the array ab; must be at least $ka+1$.
<code>lddb</code>	INTEGER. The first dimension of the array bb; must be at least $kb+1$.
<code>ldx</code>	The first dimension of the output array x. Constraints: if <code>vect = 'N'</code>, then $ldx \geq 1$; if <code>vect = 'V'</code>, then $ldx \geq \max(1, n)$.

Output Parameters

<i>ab</i>	On exit, this array is overwritten by the upper or lower triangle of <i>C</i> as specified by <i>uplo</i> .
<i>x</i>	REAL for <i>ssbgst</i> DOUBLE PRECISION for <i>dsbgst</i> Array. If <i>vect</i> = 'V', then <i>x</i> (<i>ldx</i> ,*) contains the <i>n</i> -by- <i>n</i> matrix $X = \text{inv}(S) * Q$. If <i>vect</i> = 'N', then <i>x</i> is not referenced. The second dimension of <i>x</i> must be: at least $\max(1, n)$, if <i>vect</i> = 'V'; at least 1, if <i>vect</i> = 'N'.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *sbgst* interface are the following:

<i>a</i>	Stands for argument <i>ab</i> in Fortran 77 interface. Holds the array <i>A</i> of size (<i>ka</i> +1, <i>n</i>).
<i>b</i>	Stands for argument <i>bb</i> in Fortran 77 interface. Holds the array <i>B</i> of size (<i>kb</i> +1, <i>n</i>).
<i>x</i>	Holds the matrix <i>X</i> of size (<i>n</i> , <i>n</i>).
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>vect</i>	Restored based on the presence of the argument <i>x</i> as follows: <i>vect</i> = 'V', if <i>x</i> is present, <i>vect</i> = 'N', if <i>x</i> is omitted.

Application Notes

Forming the reduced matrix *C* involves implicit multiplication by $\text{inv}(B)$. When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if *B* is ill-conditioned with respect to inversion.

If ka and kb are much less than n then the total number of floating-point operations is approximately $6n^2*kb$, when $vect = 'N'$. Additional $(3/2)n^3*(kb/ka)$ operations are required when $vect = 'V'$.

?hbgst

Reduces a complex Hermitian-definite generalized eigenproblem for banded matrices to the standard form using the factorization performed by ?pbstf.

Syntax

FORTRAN 77:

```
call chbgst(vect, uplo, n, ka, kb, ab, ldab, bb, ldbb, x, ldx, work, rwork,
info)

call zhbgst(vect, uplo, n, ka, kb, ab, ldab, bb, ldbb, x, ldx, work, rwork,
info)
```

Fortran 95:

```
call hbgst(a, b [,x] [,uplo] [,info])
```

Description

To reduce the complex Hermitian-definite generalized eigenproblem $A^*z = \lambda^*B^*z$ to the standard form $C^*x = \lambda^*y$, where A , B and C are banded, this routine must be preceded by a call to [cpbstf/zpbstf](#), which computes the split Cholesky factorization of the positive-definite matrix B : $B = S^H*S$. The split Cholesky factorization, compared with the ordinary Cholesky factorization, allows the work to be approximately halved.

This routine overwrites A with $C = X^H*A*X$, where $X = \text{inv}(S)*Q$, and Q is a unitary matrix chosen (implicitly) to preserve the bandwidth of A . The routine also has an option to allow the accumulation of X , and then, if z is an eigenvector of C , X^*z is an eigenvector of the original system.

Input Parameters

<i>vect</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>vect</i> = 'N', then matrix <i>x</i> is not returned; If <i>vect</i> = 'V', then matrix <i>x</i> is returned.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ab</i> stores the upper triangular part of <i>A</i>. If <i>uplo</i> = 'L', <i>ab</i> stores the lower triangular part of <i>A</i>.
<i>n</i>	INTEGER. The order of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>ka</i>	INTEGER. The number of super- or sub-diagonals in <i>A</i> ($ka \geq 0$).
<i>kb</i>	INTEGER. The number of super- or sub-diagonals in <i>B</i> ($ka \geq kb \geq 0$).
<i>ab, bb, work</i>	COMPLEX for <i>chbgst</i> DOUBLE COMPLEX for <i>zhbgst</i> <i>ab</i> (<i>ldab</i>,*) is an array containing either upper or lower triangular part of the Hermitian matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format. The second dimension of the array <i>ab</i> must be at least $\max(1, n)$. <i>bb</i> (<i>ldbb</i>,*) is an array containing the banded split Cholesky factor of <i>B</i> as specified by <i>uplo</i>, <i>n</i> and <i>kb</i> and returned by cpbstf/zpbstf. The second dimension of the array <i>bb</i> must be at least $\max(1, n)$. <i>work</i>(*) is a workspace array, dimension at least $\max(1, n)$
<i>ldab</i>	INTEGER. The first dimension of the array <i>ab</i> ; must be at least $ka+1$.
<i>ldbb</i>	INTEGER. The first dimension of the array <i>bb</i> ; must be at least $kb+1$.
<i>ldx</i>	The first dimension of the output array <i>x</i>. Constraints: if <i>vect</i> = 'N', then $ldx \geq 1$; if <i>vect</i> = 'V', then $ldx \geq \max(1, n)$.
<i>rwork</i>	REAL for <i>chbgst</i> DOUBLE PRECISION for <i>zhbgst</i> Workspace array, dimension at least $\max(1, n)$

Output Parameters

<i>ab</i>	On exit, this array is overwritten by the upper or lower triangle of <i>C</i> as specified by <i>uplo</i> .
<i>x</i>	COMPLEX for <i>chbgst</i> DOUBLE COMPLEX for <i>zhbgst</i> Array. If <i>vect</i> = 'V', then <i>x</i> (<i>ldx</i> ,*) contains the <i>n</i> -by- <i>n</i> matrix $X = \text{inv}(S) * Q$. If <i>vect</i> = 'N', then <i>x</i> is not referenced. The second dimension of <i>x</i> must be: at least $\max(1, n)$, if <i>vect</i> = 'V'; at least 1, if <i>vect</i> = 'N'.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *hbgst* interface are the following:

<i>a</i>	Stands for argument <i>ab</i> in Fortran 77 interface. Holds the array <i>A</i> of size (<i>ka</i> +1, <i>n</i>).
<i>b</i>	Stands for argument <i>bb</i> in Fortran 77 interface. Holds the array <i>B</i> of size (<i>kb</i> +1, <i>n</i>).
<i>x</i>	Holds the matrix <i>X</i> of size (<i>n</i> , <i>n</i>).
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>vect</i>	Restored based on the presence of the argument <i>x</i> as follows: <i>vect</i> = 'V', if <i>x</i> is present, <i>vect</i> = 'N', if <i>x</i> is omitted.

Application Notes

Forming the reduced matrix *C* involves implicit multiplication by $\text{inv}(B)$. When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if *B* is ill-conditioned with respect to inversion. The

total number of floating-point operations is approximately $20n^2*kb$, when $vect = 'N'$. Additional $5n^3*(kb/ka)$ operations are required when $vect = 'V'$. All these estimates assume that both ka and kb are much less than n .

?pbstf

Computes a split Cholesky factorization of a real symmetric or complex Hermitian positive-definite banded matrix used in ?sbgst/?hbgst.

Syntax

FORTRAN 77:

```
call spbstf(uplo, n, kb, bb, ldbb, info)
call dpbstf(uplo, n, kb, bb, ldbb, info)
call cpbstf(uplo, n, kb, bb, ldbb, info)
call zpbstf(uplo, n, kb, bb, ldbb, info)
```

Fortran 95:

```
call pbstf(b [, uplo] [,info])
```

Description

This routine computes a split Cholesky factorization of a real symmetric or complex Hermitian positive-definite band matrix B . It is to be used in conjunction with [?sbgst/?hbgst](#).

The factorization has the form $B = S^T * S$ (or $B = S^H * S$ for complex flavors), where S is a band matrix of the same bandwidth as B and the following structure: S is upper triangular in the first $(n+kb)/2$ rows and lower triangular in the remaining rows.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>bb</i> stores the upper triangular part of B . If <i>uplo</i> = 'L', <i>bb</i> stores the lower triangular part of B .
<i>n</i>	INTEGER. The order of the matrix B ($n \geq 0$).
<i>kb</i>	INTEGER. The number of super- or sub-diagonals in B ($kb \geq 0$).

bb REAL for `spbstf`
 DOUBLE PRECISION for `dpbstf`
 COMPLEX for `cpbstf`
 DOUBLE COMPLEX for `zpbstf`.
bb (*ldb*,*) is an array containing either upper or lower triangular part of the matrix *B* (as specified by *uplo*) in band storage format.
 The second dimension of the array *bb* must be at least $\max(1, n)$.

ldb INTEGER. The first dimension of *bb*; must be at least $kb+1$.

Output Parameters

bb On exit, this array is overwritten by the elements of the split Cholesky factor *S*.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = *i*, then the factorization could not be completed, because the updated element b_{ii} would be the square root of a negative number; hence the matrix *B* is not positive-definite.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `pbstf` interface are the following:

b Stands for argument *bb* in Fortran 77 interface. Holds the array *B* of size $(kb+1, n)$.

uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed factor *S* is the exact factor of a perturbed matrix $B + E$, where

$$|E| \leq c(kb + 1)\epsilon \|S^H\| |S|, \quad |e_{ij}| \leq c(kb + 1)\epsilon \sqrt{b_{ii}b_{jj}}$$

$c(n)$ is a modest linear function of n , and ϵ is the machine precision.

The total number of floating-point operations for real flavors is approximately $n(kb+1)^2$. The number of operations for complex flavors is 4 times greater. All these estimates assume that kb is much less than n .

After calling this routine, you can call `?sbgst/?hbgst` to solve the generalized eigenproblem $Az = \lambda Bz$, where A and B are banded and B is positive-definite.

Nonsymmetric Eigenvalue Problems

This section describes LAPACK routines for solving nonsymmetric eigenvalue problems, computing the Schur factorization of general matrices, as well as performing a number of related computational tasks.

A *nonsymmetric eigenvalue problem* is as follows: given a nonsymmetric (or non-Hermitian) matrix A , find the *eigenvalues* λ and the corresponding *eigenvectors* z that satisfy the equation

$$Az = \lambda z \text{ (right eigenvectors } z)$$

or the equation

$$z^H A = \lambda z^H \text{ (left eigenvectors } z).$$

Nonsymmetric eigenvalue problems have the following properties:

- The number of eigenvectors may be less than the matrix order (but is not less than the number of *distinct eigenvalues* of A).
- Eigenvalues may be complex even for a real matrix A .
- If a real nonsymmetric matrix has a complex eigenvalue $a+bi$ corresponding to an eigenvector z , then $a-bi$ is also an eigenvalue. The eigenvalue $a-bi$ corresponds to the eigenvector whose elements are complex conjugate to the elements of z .

To solve a nonsymmetric eigenvalue problem with LAPACK, you usually need to reduce the matrix to the upper Hessenberg form and then solve the eigenvalue problem with the Hessenberg matrix obtained. Table 4-5 lists LAPACK routines (Fortran 77 interface) for reducing the matrix to the upper Hessenberg form by an orthogonal (or unitary) similarity transformation $A = QHQ^H$ as well as routines for solving eigenvalue problems with Hessenberg matrices, forming the

Schur factorization of such matrices and computing the corresponding condition numbers. Respective routine names in Fortran 95 interface are without the first symbol (see [Routine Naming Conventions](#)).

Decision tree in [Figure 4-4](#) helps you choose the right routine or sequence of routines for an eigenvalue problem with a real nonsymmetric matrix. If you need to solve an eigenvalue problem with a complex non-Hermitian matrix, use the decision tree shown in [Figure 4-5](#).

Table 4-5 Computational Routines for Solving Nonsymmetric Eigenvalue Problems

Operation performed	Routines for real matrices	Routines for complex matrices
Reduce to Hessenberg form $A = QHQ^H$	?gehrd ,	?gehrd
Generate the matrix Q	?orghr	?unghr
Apply the matrix Q	?ormhr	?unmhr
Balance matrix	?gebal	?gebal
Transform eigenvectors of balanced matrix to those of the original matrix	?gebak	?gebak
Find eigenvalues and Schur factorization (QR algorithm)	?hseqr	?hseqr
Find eigenvectors from Hessenberg form (inverse iteration)	?hsein	?hsein
Find eigenvectors from Schur factorization	?trevc	?trevc
Estimate sensitivities of eigenvalues and eigenvectors	?trsna	?trsna
Reorder Schur factorization	?trexc	?trexc
Reorder Schur factorization, find the invariant subspace and estimate sensitivities	?trsen	?trsen

Operation performed	Routines for real matrices	Routines for complex matrices
Solves Sylvester's equation.	<code>?trsyl</code>	<code>?trsyl</code>

Figure 4-4 Decision Tree: Real Nonsymmetric Eigenvalue Problems

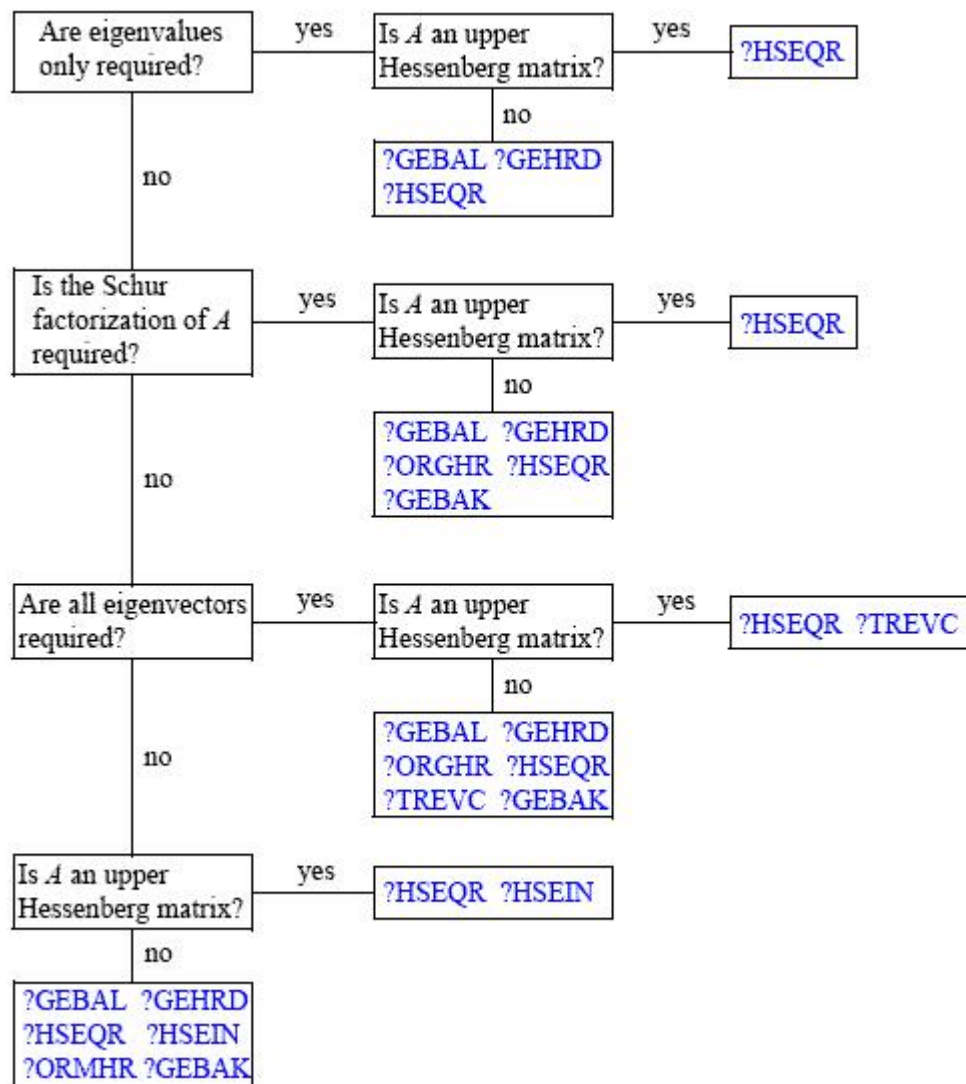
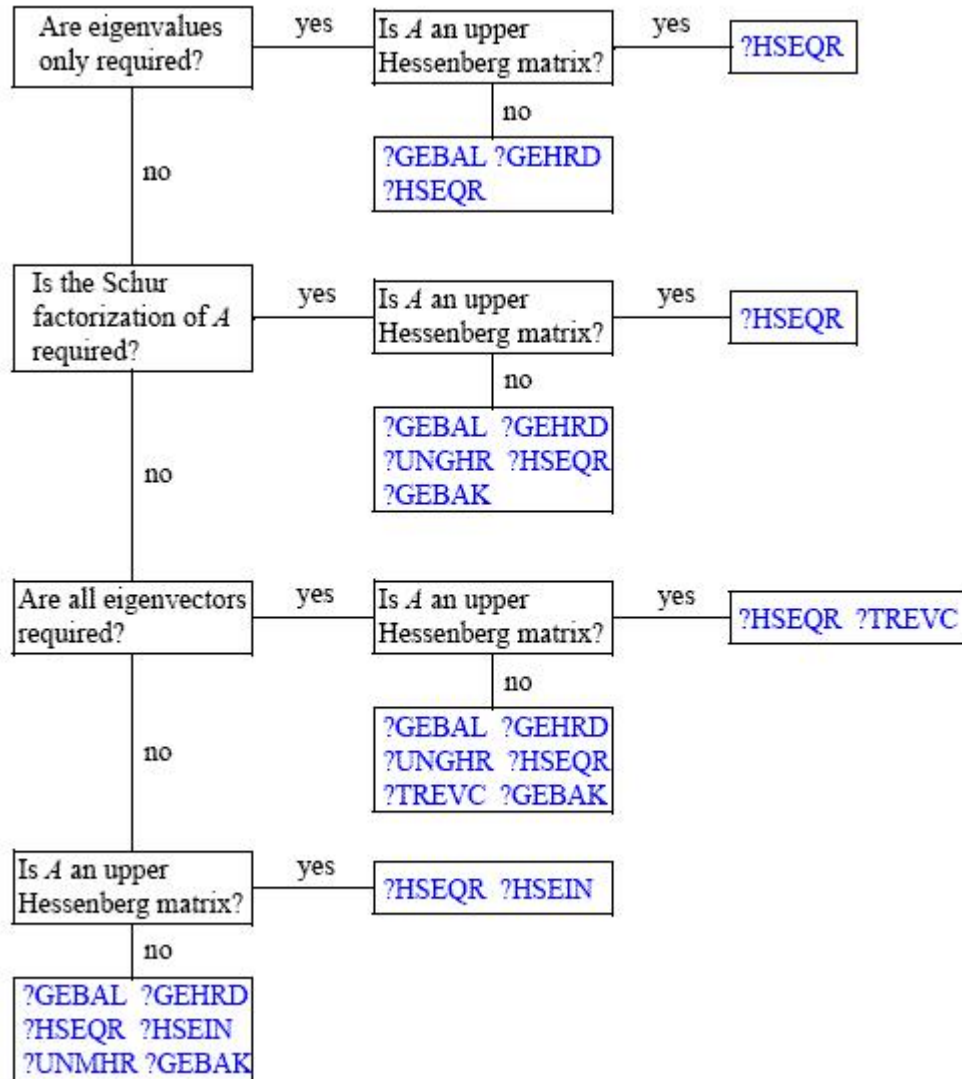


Figure 4-5 Decision Tree: Complex Non-Hermitian Eigenvalue Problems

?gehrd

Reduces a general matrix to upper Hessenberg form.

Syntax

FORTRAN 77:

```
call sgehrd(n, ilo, ihi, a, lda, tau, work, lwork, info)
call dgehrd(n, ilo, ihi, a, lda, tau, work, lwork, info)
call cgehrd(n, ilo, ihi, a, lda, tau, work, lwork, info)
call zgehrd(n, ilo, ihi, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call gehrd(a [, tau] [,ilo] [,ihi] [,info])
```

Description

The routine reduces a general matrix A to upper Hessenberg form H by an orthogonal or unitary similarity transformation $A = Q^*H^*Q^H$. Here H has real subdiagonal elements.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of *elementary reflectors*. Routines are provided to work with Q in this representation.

Input Parameters

n	INTEGER. The order of the matrix A ($n \geq 0$).
ilo, ihi	INTEGER. If A is an output by ?gebal, then ilo and ihi must contain the values returned by that routine. Otherwise $ilo = 1$ and $ihi = n$. (If $n > 0$, then $1 \leq ilo \leq ihi \leq n$; if $n = 0$, $ilo = 1$ and $ihi = 0$.)
$a, work$	REAL for sgehrd DOUBLE PRECISION for dgehrd COMPLEX for cgehrd DOUBLE COMPLEX for zgehrd. Arrays: $a(lda,*)$ contains the matrix A. The second dimension of a must be at least $\max(1, n)$.

work (*lwork*) is a workspace array.

lda INTEGER. The first dimension of *a*; at least $\max(1, n)$.

lwork INTEGER. The size of the *work* array; at least $\max(1, n)$.
 If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#).
 See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a Overwritten by the upper Hessenberg matrix *H* and details of the matrix *Q*. The subdiagonal elements of *H* are real.

tau REAL for *sgehrd*
 DOUBLE PRECISION for *dgehrd*
 COMPLEX for *cgehrd*
 DOUBLE COMPLEX for *zgehrd*.
 Array, DIMENSION at least $\max(1, n-1)$.
 Contains additional information on the matrix *Q*.

work(1) If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *gehrd* interface are the following:

a Holds the matrix *A* of size (n, n) .

tau Holds the vector of length $(n-1)$.

ilo Default value for this argument is *ilo* = 1.

ihi Default value for this argument is *ihi* = *n*.

Application Notes

For better performance, try using $lwork = n * blocksize$, where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed Hessenberg matrix H is exactly similar to a nearby matrix $A + E$, where $\|E\|_2 < c(n)\epsilon\|A\|_2$, $c(n)$ is a modestly increasing function of n , and ϵ is the machine precision.

The approximate number of floating-point operations for real flavors is $(2/3) * (ihi - ilo)^2 (2ihi + 2ilo + 3n)$; for complex flavors it is 4 times greater.

?orghr

Generates the real orthogonal matrix Q determined by ?gehrd.

Syntax

FORTRAN 77:

```
call sorghr(n, ilo, ihi, a, lda, tau, work, lwork, info)
call dorghr(n, ilo, ihi, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call orghr(a, tau [,ilo] [,ihi] [,info])
```


Description

This routine explicitly generates the orthogonal matrix Q that has been determined by a preceding call to `sgehrd/dgehrd`. (The routine `?gehrd` reduces a real general matrix A to upper Hessenberg form H by an orthogonal similarity transformation, $A = Q^* H^* Q^T$, and represents the matrix Q as a product of $ihi-ilo$ elementary reflectors. Here ilo and ihi are values determined by `sgebal/dgebal` when balancing the matrix; if the matrix has not been balanced, $ilo = 1$ and $ihi = n$.)

The matrix Q generated by `?orghr` has the structure:

$$Q = \begin{bmatrix} I & 0 & 0 \\ 0 & Q_{22} & 0 \\ 0 & 0 & I \end{bmatrix}$$

where Q_{22} occupies rows and columns ilo to ihi .

Input Parameters

n	INTEGER. The order of the matrix Q ($n \geq 0$).
ilo, ihi	INTEGER. These must be the same parameters ilo and ihi , respectively, as supplied to <code>?gehrd</code> . (If $n > 0$, then $1 \leq ilo \leq ihi \leq n$; if $n = 0$, $ilo = 1$ and $ihi = 0$.)
$a, tau, work$	REAL for <code>sorghr</code> DOUBLE PRECISION for <code>dorghr</code> Arrays: $a(lda,*)$ contains details of the vectors which define the elementary reflectors, as returned by <code>?gehrd</code> . The second dimension of a must be at least $\max(1, n)$. $tau(*)$ contains further details of the elementary reflectors, as returned by <code>?gehrd</code> . The dimension of tau must be at least $\max(1, n-1)$. $work$ is a workspace array, its dimension $\max(1, lwork)$.
lda	INTEGER. The first dimension of a ; at least $\max(1, n)$.
$lwork$	INTEGER. The size of the $work$ array;

$lwork \geq \max(1, ihi - ilo)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by *xerbla*. See *Application Notes* for the suggested value of *lwork*.

Output Parameters

<i>a</i>	Overwritten by the <i>n</i> -by- <i>n</i> orthogonal matrix <i>Q</i> .
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *orghr* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>tau</i>	Holds the vector of length (<i>n</i> -1).
<i>ilo</i>	Default value for this argument is <i>ilo</i> = 1.
<i>ihi</i>	Default value for this argument is <i>ihi</i> = <i>n</i> .

Application Notes

For better performance, try using $lwork = (ihi - ilo) * blocksize$ where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix $\hat{Q}$ differs from the exact result by a matrix E such that $\|E\|_2 = O(\epsilon)$, where ϵ is the machine precision.

The approximate number of floating-point operations is $(4/3)(ihi-ilo)^3$.

The complex counterpart of this routine is [?unghr](#).

[?ormhr](#)

Multiplies an arbitrary real matrix C by the real orthogonal matrix Q determined by [?gehrd](#).

Syntax

FORTRAN 77:

```
call sormhr(side, trans, m, n, ilo, ihi, a, lda, tau, c, ldc, work, lwork,
info)
```

```
call dormhr(side, trans, m, n, ilo, ihi, a, lda, tau, c, ldc, work, lwork,
info)
```

Fortran 95:

```
call ormhr(a, tau, c [,ilo] [,ihi] [,side] [,trans] [,info])
```

Description

This routine multiplies a matrix C by the orthogonal matrix Q that has been determined by a preceding call to [sgehrd](#)/[dgehrd](#). (The routine [?gehrd](#) reduces a real general matrix A to upper Hessenberg form H by an orthogonal similarity transformation, $A = Q^*H^*Q^T$, and represents

the matrix Q as a product of *ihi-ilo elementary reflectors*. Here *ilo* and *ihi* are values determined by *sgebal/dgebal* when balancing the matrix; if the matrix has not been balanced, *ilo* = 1 and *ihi* = *n*.)

With *?ormhr*, you can form one of the matrix products Q^*C , $Q^T * C$, C^*Q , or C^*Q^T , overwriting the result on *c* (which may be any real rectangular matrix).

A common application of *?ormhr* is to transform a matrix *v* of eigenvectors of *H* to the matrix QV of eigenvectors of *A*.

Input Parameters

<i>side</i>	CHARACTER*1. Must be 'L' or 'R'. If <i>side</i> = 'L', then the routine forms Q^*C or $Q^T * C$. If <i>side</i> = 'R', then the routine forms C^*Q or C^*Q^T .
<i>trans</i>	CHARACTER*1. Must be 'N' or 'T'. If <i>trans</i> = 'N', then Q is applied to <i>c</i> . If <i>trans</i> = 'T', then Q^T is applied to <i>c</i> .
<i>m</i>	INTEGER. The number of rows in <i>c</i> ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in <i>c</i> ($n \geq 0$).
<i>ilo, ihi</i>	INTEGER. These must be the same parameters <i>ilo</i> and <i>ihi</i> , respectively, as supplied to <i>?gehrd</i> . If $m > 0$ and <i>side</i> = 'L', then $1 \leq ilo \leq ihi \leq m$. If $m = 0$ and <i>side</i> = 'L', then <i>ilo</i> = 1 and <i>ihi</i> = 0. If $n > 0$ and <i>side</i> = 'R', then $1 \leq ilo \leq ihi \leq n$. If $n = 0$ and <i>side</i> = 'R', then <i>ilo</i> = 1 and <i>ihi</i> = 0.
<i>a, tau, c, work</i>	REAL for <i>sormhr</i> DOUBLE PRECISION for <i>dormhr</i> Arrays: <i>a(lda,*)</i> contains details of the vectors which define the <i>elementary reflectors</i> , as returned by <i>?gehrd</i> . The second dimension of <i>a</i> must be at least $\max(1, m)$ if <i>side</i> = 'L' and at least $\max(1, n)$ if <i>side</i> = 'R'. <i>tau(*)</i> contains further details of the <i>elementary reflectors</i> , as returned by <i>?gehrd</i> . The dimension of <i>tau</i> must be at least $\max(1, m-1)$ if <i>side</i> = 'L' and at least $\max(1, n-1)$ if <i>side</i> = 'R'.

$c(ldc,*)$ contains the m by n matrix C . The second dimension of c must be at least $\max(1, n)$.
 $work$ is a workspace array, its dimension $\max(1, lwork)$.
lda INTEGER. The first dimension of a ; at least $\max(1, m)$ if $side = 'L'$ and at least $\max(1, n)$ if $side = 'R'$.
ldc INTEGER. The first dimension of c ; at least $\max(1, m)$.
lwork INTEGER. The size of the $work$ array.
 If $side = 'L'$, $lwork \geq \max(1, n)$.
 If $side = 'R'$, $lwork \geq \max(1, m)$.
 If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).
 See *Application Notes* for the suggested value of $lwork$.

Output Parameters

c C is overwritten by product Q^*C , $Q^T C$, C^*Q , or $C^T Q$ as specified by $side$ and $trans$.
 $work(1)$ If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.
info INTEGER.
 If $info = 0$, the execution is successful.
 If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ormhr` interface are the following:

a Holds the matrix A of size (r, r) .
 $r = m$ if $side = 'L'$.
 $r = n$ if $side = 'R'$.
 τ Holds the vector of length $(r-1)$.

<i>c</i>	Holds the matrix <i>c</i> of size (m,n) .
<i>ilo</i>	Default value for this argument is <i>ilo</i> = 1.
<i>ihi</i>	Default value for this argument is <i>ihi</i> = <i>n</i> .
<i>side</i>	Must be 'L' or 'R'. The default value is 'L'.
<i>trans</i>	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, *lwork* should be at least $n \times \text{blocksize}$ if *side* = 'L' and at least $m \times \text{blocksize}$ if *side* = 'R', where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix *Q* differs from the exact result by a matrix *E* such that $\|E\|_2 = O(\epsilon) \|C\|_2$, where ϵ is the machine precision.

The approximate number of floating-point operations is

$2n(ihi-ilo)^2$ if *side* = 'L';

$2m(ihi-ilo)^2$ if *side* = 'R'.

The complex counterpart of this routine is [?unmhr](#).

?unghr

Generates the complex unitary matrix Q determined by ?gehrd.

Syntax

FORTRAN 77:

```
call cunghr(n, ilo, ihi, a, lda, tau, work, lwork, info)
call zunghr(n, ilo, ihi, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call unghr(a, tau [,ilo] [,ihi] [,info])
```

Description

This routine is intended to be used following a call to cgehrd/zgehrd, which reduces a complex matrix A to upper Hessenberg form H by a unitary similarity transformation: $A = Q^* H Q$.

?gehrd represents the matrix Q as a product of $ihi-ilo$ *elementary reflectors*. Here ilo and ihi are values determined by cgebal/zgebal when balancing the matrix; if the matrix has not been balanced, $ilo = 1$ and $ihi = n$.

Use the routine ?unghr to generate Q explicitly as a square matrix. The matrix Q has the structure:

$$Q = \begin{bmatrix} I & 0 & 0 \\ 0 & Q_{22} & 0 \\ 0 & 0 & I \end{bmatrix}$$

where Q_{22} occupies rows and columns ilo to ihi .

Input Parameters

n INTEGER. The order of the matrix Q ($n \geq 0$).

<i>ilo, ihi</i>	INTEGER. These must be the same parameters <i>ilo</i> and <i>ihi</i> , respectively, as supplied to <code>?gehrd</code> . (If $n > 0$, then $1 \leq ilo \leq ihi \leq n$. If $n = 0$, then $ilo = 1$ and $ihi = 0$.)
<i>a, tau, work</i>	COMPLEX for <code>cunghr</code> DOUBLE COMPLEX for <code>zunghr</code> . Arrays: <i>a</i> (<i>lda</i> ,*) contains details of the vectors which define the <i>elementary reflectors</i> , as returned by <code>?gehrd</code> . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>tau</i> (*) contains further details of the <i>elementary reflectors</i> , as returned by <code>?gehrd</code> . The dimension of <i>tau</i> must be at least $\max(1, n-1)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; at least $\max(1, n)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array; $lwork \geq \max(1, ihi-ilo)$. If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <code>xerbla</code> . See <i>Application Notes</i> for the suggested value of <i>lwork</i> .

Output Parameters

<i>a</i>	Overwritten by the n -by- n unitary matrix Q .
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `unghr` interface are the following:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>tau</code>	Holds the vector of length $(n-1)$.
<code>ilo</code>	Default value for this argument is $ilo = 1$.
<code>ihi</code>	Default value for this argument is $ihi = n$.

Application Notes

For better performance, try using $lwork = (ihi-ilo)*blocksize$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set $lwork = -1$.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix Q differs from the exact result by a matrix E such that $\|E\|_2 = O(\epsilon)$, where ϵ is the machine precision.

The approximate number of real floating-point operations is $(16/3)(ihi-ilo)^3$.

The real counterpart of this routine is `?orghr`.

?unmhr

Multiplies an arbitrary complex matrix C by the complex unitary matrix Q determined by ?gehrd.

Syntax

FORTRAN 77:

```
call cunmhr(side, trans, m, n, ilo, ihi, a, lda, tau, c, ldc, work, lwork,
info)

call zunmhr(side, trans, m, n, ilo, ihi, a, lda, tau, c, ldc, work, lwork,
info)
```

Fortran 95:

```
call unmhr(a, tau, c [,ilo] [,ihi] [,side] [,trans] [,info])
```

Description

This routine multiplies a matrix C by the unitary matrix Q that has been determined by a preceding call to cgehrd/zgehrd. (The routine ?gehrd reduces a real general matrix A to upper Hessenberg form H by an orthogonal similarity transformation, $A = Q^H H Q$, and represents the matrix Q as a product of $ihi-ilo$ elementary reflectors. Here ilo and ihi are values determined by cgebal/zgebal when balancing the matrix; if the matrix has not been balanced, $ilo = 1$ and $ihi = n$.)

With ?unmhr, you can form one of the matrix products $Q^H C$, $Q^H C Q$, $C Q$, or $C Q^H$, overwriting the result on C (which may be any complex rectangular matrix). A common application of this routine is to transform a matrix V of eigenvectors of H to the matrix QV of eigenvectors of A .

Input Parameters

<i>side</i>	CHARACTER*1. Must be 'L' or 'R'. If <i>side</i> = 'L', then the routine forms $Q^H C$ or $Q^H C Q$. If <i>side</i> = 'R', then the routine forms $C Q$ or $C Q^H$.
<i>trans</i>	CHARACTER*1. Must be 'N' or 'C'. If <i>trans</i> = 'N', then Q is applied to C . If <i>trans</i> = 'T', then Q^H is applied to C .
<i>m</i>	INTEGER. The number of rows in C ($m \geq 0$).

n INTEGER. The number of columns in *c* ($n \geq 0$).

ilo, ihi INTEGER. These must be the same parameters *ilo* and *ihi*, respectively, as supplied to `?gehrd`.

If $m > 0$ and *side* = 'L', then $1 \leq ilo \leq ihi \leq m$.
 If $m = 0$ and *side* = 'L', then *ilo* = 1 and *ihi* = 0.
 If $n > 0$ and *side* = 'R', then $1 \leq ilo \leq ihi \leq n$.
 If $n = 0$ and *side* = 'R', then *ilo* = 1 and *ihi* = 0.

a, tau, c, work COMPLEX for `cunmhr`
 DOUBLE COMPLEX for `zunmhr`.
 Arrays:
a(*lda*,*) contains details of the vectors which define the elementary reflectors, as returned by `?gehrd`.
 The second dimension of *a* must be at least $\max(1, m)$ if *side* = 'L' and at least $\max(1, n)$ if *side* = 'R'.
tau(*) contains further details of the elementary reflectors, as returned by `?gehrd`.
 The dimension of *tau* must be at least $\max(1, m-1)$ if *side* = 'L' and at least $\max(1, n-1)$ if *side* = 'R'.
c(*ldc*,*) contains the *m*-by-*n* matrix *C*. The second dimension of *c* must be at least $\max(1, n)$.
work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The first dimension of *a*; at least $\max(1, m)$ if *side* = 'L' and at least $\max(1, n)$ if *side* = 'R'.

ldc INTEGER. The first dimension of *c*; at least $\max(1, m)$.

lwork INTEGER. The size of the *work* array.
 If *side* = 'L', $lwork \geq \max(1, n)$.
 If *side* = 'R', $lwork \geq \max(1, m)$.
 If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by `xerbla`.
 See *Application Notes* for the suggested value of *lwork*.

Output Parameters

<i>c</i>	<i>c</i> is overwritten by $Q^H C$, or $Q^H C$, or $C^H Q$, or $C^H Q$ as specified by <i>side</i> and <i>trans</i> .
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `unmhr` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>r</i> , <i>r</i>). <i>r</i> = <i>m</i> if <i>side</i> = 'L'. <i>r</i> = <i>n</i> if <i>side</i> = 'R'.
<i>tau</i>	Holds the vector of length (<i>r</i> -1).
<i>c</i>	Holds the matrix <i>C</i> of size (<i>m</i> , <i>n</i>).
<i>ilo</i>	Default value for this argument is <i>ilo</i> = 1.
<i>ihi</i>	Default value for this argument is <i>ihi</i> = <i>n</i> .
<i>side</i>	Must be 'L' or 'R'. The default value is 'L'.
<i>trans</i>	Must be 'N' or 'C'. The default value is 'N'.

Application Notes

For better performance, *lwork* should be at least $n \times \text{blocksize}$ if *side* = 'L' and at least $m \times \text{blocksize}$ if *side* = 'R', where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If `lwork = -1`, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if `lwork` is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix Q differs from the exact result by a matrix E such that $\|E\|_2 = O(\epsilon) * \|C\|_2$, where ϵ is the machine precision.

The approximate number of floating-point operations is

$8n(ihi-ilo)^2$ if `side = 'L'`;

$8m(ihi-ilo)^2$ if `side = 'R'`.

The real counterpart of this routine is [?ormhr](#).

[?gebal](#)

Balances a general matrix to improve the accuracy of computed eigenvalues and eigenvectors.

Syntax

FORTRAN 77:

```
call sgebal(job, n, a, lda, ilo, ihi, scale, info)
call dgebal(job, n, a, lda, ilo, ihi, scale, info)
call cgebal(job, n, a, lda, ilo, ihi, scale, info)
call zgebal(job, n, a, lda, ilo, ihi, scale, info)
```

Fortran 95:

```
call gebal(a [, scale] [,ilo] [,ihi] [,job] [,info])
```

Description

This routine *balances* a matrix *A* by performing either or both of the following two similarity transformations:

- (1) The routine first attempts to permute *A* to block upper triangular form:

$$PAP^T = A' = \begin{bmatrix} A'_{11} & A'_{12} & A'_{13} \\ 0 & A'_{22} & A'_{23} \\ 0 & 0 & A'_{33} \end{bmatrix}$$

where *P* is a permutation matrix, and *A'*₁₁ and *A'*₃₃ are upper triangular. The diagonal elements of *A'*₁₁ and *A'*₃₃ are eigenvalues of *A*. The rest of the eigenvalues of *A* are the eigenvalues of the central diagonal block *A'*₂₂, in rows and columns *ilo* to *ihi*. Subsequent operations to compute the eigenvalues of *A* (or its Schur factorization) need only be applied to these rows and columns; this can save a significant amount of work if *ilo* > 1 and *ihi* < *n*.

If no suitable permutation exists (as is often the case), the routine sets *ilo* = 1 and *ihi* = *n*, and *A'*₂₂ is the whole of *A*.

- (2) The routine applies a diagonal similarity transformation to *A'*, to make the rows and columns of *A'*₂₂ as close in norm as possible:

$$A'' = DA'D^{-1} = \begin{bmatrix} I & 0 & 0 \\ 0 & D_{22} & 0 \\ 0 & 0 & I \end{bmatrix} \times \begin{bmatrix} A'_{11} & A'_{12} & A'_{13} \\ 0 & A'_{22} & A'_{23} \\ 0 & 0 & A'_{33} \end{bmatrix} \times \begin{bmatrix} I & 0 & 0 \\ 0 & D_{22}^{-1} & 0 \\ 0 & 0 & I \end{bmatrix}$$

This scaling can reduce the norm of the matrix (that is, $\|A'_{22}\| < \|A'_{22}\|$), and hence reduce the effect of rounding errors on the accuracy of computed eigenvalues and eigenvectors.

Input Parameters

job CHARACTER*1. Must be 'N' or 'P' or 'S' or 'B'.

If $job = 'N'$, then A is neither permuted nor scaled (but ilo , ihi , and $scale$ get their values).
 If $job = 'P'$, then A is permuted but not scaled.
 If $job = 'S'$, then A is scaled but not permuted.
 If $job = 'B'$, then A is both scaled and permuted.

n INTEGER. The order of the matrix A ($n \geq 0$).

a REAL for sgebal
 DOUBLE PRECISION for dgebal
 COMPLEX for cgebal
 DOUBLE COMPLEX for zgebal.
 Arrays:
 $a(lda,*)$ contains the matrix A .
 The second dimension of a must be at least $\max(1, n)$. a is not referenced if $job = 'N'$.

lda INTEGER. The first dimension of a ; at least $\max(1, n)$.

Output Parameters

a Overwritten by the balanced matrix (a is not referenced if $job = 'N'$).

ilo, ihi INTEGER. The values ilo and ihi such that on exit $a(i, j)$ is zero if $i > j$ and $1 \leq j < ilo$ or $ihi < i \leq n$.
 If $job = 'N'$ or $'S'$, then $ilo = 1$ and $ihi = n$.

$scale$ REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors
 Array, DIMENSION at least $\max(1, n)$.
 Contains details of the permutations and scaling factors.
 More precisely, if p_j is the index of the row and column interchanged with row and column j , and d_j is the scaling factor used to balance row and column j , then
 $scale(j) = p_j$ for $j = 1, 2, \dots, ilo-1, ihi+1, \dots, n$;
 $scale(j) = d_j$ for $j = ilo, ilo + 1, \dots, ihi$.
 The order in which the interchanges are made is n to $ihi+1$, then 1 to $ilo-1$.

$info$ INTEGER.

If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gebal` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>scale</i>	Holds the vector of length (<i>n</i>).
<i>ilo</i>	Default value for this argument is <i>ilo</i> = 1.
<i>ihi</i>	Default value for this argument is <i>ihi</i> = <i>n</i> .
<i>job</i>	Must be 'B', 'S', 'P', or 'N'. The default value is 'B'.

Application Notes

The errors are negligible, compared with those in subsequent computations.

If the matrix *A* is balanced by this routine, then any eigenvectors computed subsequently are eigenvectors of the matrix *A*'', and hence you must call `?gebak` to transform them back to eigenvectors of *A*.

If the Schur vectors of *A* are required, do not call this routine with *job* = 'S' or 'B', because then the balancing transformation is not orthogonal (not unitary for complex flavors).

If you call this routine with *job* = 'P', then any Schur vectors computed subsequently are Schur vectors of the matrix *A*'', and you need to call `?gebak` (with *side* = 'R') to transform them back to Schur vectors of *A*.

The total number of floating-point operations is proportional to n^2 .

?gebak

Transforms eigenvectors of a balanced matrix to those of the original nonsymmetric matrix.

Syntax

FORTRAN 77:

```
call sgebak(job, side, n, ilo, ihi, scale, m, v, ldv, info)
call dgebak(job, side, n, ilo, ihi, scale, m, v, ldv, info)
call cgebak(job, side, n, ilo, ihi, scale, m, v, ldv, info)
call zgebak(job, side, n, ilo, ihi, scale, m, v, ldv, info)
```

Fortran 95:

```
call gebak(v, scale [,ilo] [,ihi] [,job] [,side] [,info])
```

Description

This routine is intended to be used after a matrix A has been balanced by a call to ?gebal, and eigenvectors of the balanced matrix A''_{22} have subsequently been computed. For a description of balancing, see ?gebal. The balanced matrix A'' is obtained as $A'' = D * P * A * P^T * \text{inv}(D)$, where P is a permutation matrix and D is a diagonal scaling matrix. This routine transforms the eigenvectors as follows:

if x is a right eigenvector of A'' , then $P^T * \text{inv}(D) * x$ is a right eigenvector of A ; if x is a left eigenvector of A'' , then $P^T * D * y$ is a left eigenvector of A .

Input Parameters

<i>job</i>	CHARACTER*1. Must be 'N' or 'P' or 'S' or 'B'. The same parameter <i>job</i> as supplied to ?gebal.
<i>side</i>	CHARACTER*1. Must be 'L' or 'R'. If <i>side</i> = 'L', then left eigenvectors are transformed. If <i>side</i> = 'R', then right eigenvectors are transformed.
<i>n</i>	INTEGER. The number of rows of the matrix of eigenvectors ($n \geq 0$).
<i>ilo, ihi</i>	INTEGER. The values <i>ilo</i> and <i>ihi</i> , as returned by ?gebal. (If $n > 0$, then $1 \leq ilo \leq ihi \leq n$;

	if $n = 0$, then $ilo = 1$ and $ihi = 0$.)
<i>scale</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors Array, DIMENSION at least $\max(1, n)$. Contains details of the permutations and/or the scaling factors used to balance the original general matrix, as returned by ?gebal.
<i>m</i>	INTEGER. The number of columns of the matrix of eigenvectors ($m \geq 0$).
<i>v</i>	REAL for sgebak DOUBLE PRECISION for dgebak COMPLEX for cgebak DOUBLE COMPLEX for zgebak. Arrays: $v(ldv,*)$ contains the matrix of left or right eigenvectors to be transformed. The second dimension of v must be at least $\max(1, m)$.
<i>ldv</i>	INTEGER. The first dimension of v ; at least $\max(1, n)$.

Output Parameters

<i>v</i>	Overwritten by the transformed eigenvectors.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gebak` interface are the following:

<i>v</i>	Holds the matrix v of size (n, m) .
<i>scale</i>	Holds the vector of length (n) .
<i>ilo</i>	Default value for this argument is $ilo = 1$.
<i>ihi</i>	Default value for this argument is $ihi = n$.

job Must be 'B', 'S', 'P', or 'N'. The default value is 'B'.
side Must be 'L' or 'R'. The default value is 'L'.

Application Notes

The errors in this routine are negligible.

The approximate number of floating-point operations is approximately proportional to $m \cdot n$.

?hseqr

Computes all eigenvalues and (optionally) the Schur factorization of a matrix reduced to Hessenberg form.

Syntax

FORTRAN 77:

```
call shseqr(job, compz, n, ilo, ihi, h, ldh, wr, wi, z, ldz, work, lwork,
info)

call dhseqr(job, compz, n, ilo, ihi, h, ldh, wr, wi, z, ldz, work, lwork,
info)

call chseqr(job, compz, n, ilo, ihi, h, ldh, w, z, ldz, work, lwork, info)
call zhseqr(job, compz, n, ilo, ihi, h, ldh, w, z, ldz, work, lwork, info)
```

Fortran 95:

```
call hseqr(h, wr, wi [,ilo] [,ihi] [,z] [,job] [,compz] [,info])
call hseqr(h, w [,ilo] [,ihi] [,z] [,job] [,compz] [,info])
```

Description

This routine computes all the eigenvalues, and optionally the Schur factorization, of an upper Hessenberg matrix H : $H = Z^* T^* Z^H$, where T is an upper triangular (or, for real flavors, quasi-triangular) matrix (the Schur form of H), and Z is the unitary or orthogonal matrix whose columns are the Schur vectors z_i .

You can also use this routine to compute the Schur factorization of a general matrix A which has been reduced to upper Hessenberg form H :

$A = Q^* H^* Q^H$, where Q is unitary (orthogonal for real flavors);

$$A = (QZ)^* H^* (QZ)^H.$$

In this case, after reducing A to Hessenberg form by `?gehrd`, call `?orghr` to form Q explicitly and then pass Q to `?hseqr` with `compz = 'V'`.

You can also call `?gebal` to balance the original matrix before reducing it to Hessenberg form by `?hseqr`, so that the Hessenberg matrix H will have the structure:

$$\begin{bmatrix} H_{11} & H_{12} & H_{13} \\ 0 & H_{22} & H_{23} \\ 0 & 0 & H_{33} \end{bmatrix}$$

where H_{11} and H_{33} are upper triangular.

If so, only the central diagonal block H_{22} (in rows and columns ilo to ihi) needs to be further reduced to Schur form (the blocks H_{12} and H_{23} are also affected). Therefore the values of ilo and ihi can be supplied to `?hseqr` directly. Also, after calling this routine you must call `?gebak` to permute the Schur vectors of the balanced matrix to those of the original matrix.

If `?gebal` has not been called, however, then ilo must be set to 1 and ihi to n . Note that if the Schur factorization of A is required, `?gebal` must not be called with `job = 'S'` or `'B'`, because the balancing transformation is not unitary (for real flavors, it is not orthogonal).

`?hseqr` uses a multishift form of the upper Hessenberg QR algorithm. The Schur vectors are normalized so that $\|z_i\|_2 = 1$, but are determined only to within a complex factor of absolute value 1 (for the real flavors, to within a factor ± 1).

Input Parameters

<code>job</code>	CHARACTER*1. Must be 'E' or 'S'. If <code>job = 'E'</code>, then eigenvalues only are required. If <code>job = 'S'</code>, then the Schur form T is required.
<code>compz</code>	CHARACTER*1. Must be 'N' or 'I' or 'V'. If <code>compz = 'N'</code>, then no Schur vectors are computed (and the array z is not referenced). If <code>compz = 'I'</code>, then the Schur vectors of H are computed (and the array z is initialized by the routine).

If *compz* = 'V', then the Schur vectors of *A* are computed (and the array *z* must contain the matrix *Q* on entry).

n INTEGER. The order of the matrix *H* ($n \geq 0$).

ilo, ihi INTEGER. If *A* has been balanced by ?gebal, then *ilo* and *ihi* must contain the values returned by ?gebal. Otherwise, *ilo* must be set to 1 and *ihi* to *n*.

h, z, work REAL for shseqr
DOUBLE PRECISION for dhseqr
COMPLEX for chseqr
DOUBLE COMPLEX for zhseqr.

Arrays:
h(ldh,)* The *n*-by-*n* upper Hessenberg matrix *H*.
 The second dimension of *h* must be at least $\max(1, n)$.
z(ldz,)*
 If *compz* = 'V', then *z* must contain the matrix *Q* from the reduction to Hessenberg form.
 If *compz* = 'I', then *z* need not be set.
 If *compz* = 'N', then *z* is not referenced.
 The second dimension of *z* must be
 at least $\max(1, n)$ if *compz* = 'V' or 'I';
 at least 1 if *compz* = 'N'.
work(lwork) is a workspace array.
 The dimension of *work* must be at least $\max(1, n)$.

ldh INTEGER. The first dimension of *h*; at least $\max(1, n)$.

ldz INTEGER. The first dimension of *z*;
 If *compz* = 'N', then $ldz \geq 1$.
 If *compz* = 'V' or 'I', then $ldz \geq \max(1, n)$.

lwork INTEGER. The dimension of the array *work*.
 $lwork \geq \max(1, n)$ is sufficient, but *lwork* typically as large as $6*n$ may be required for optimal performance. A workspace query to determine the optimal workspace size is recommended.

If `lwork = -1`, then a workspace query is assumed; the routine only estimates the optimal size of the `work` array, returns this value as the first entry of the `work` array, and no error message related to `lwork` is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

<code>w</code>	COMPLEX for <code>chseqr</code> DOUBLE COMPLEX for <code>zhseqr</code>. Array, DIMENSION at least $\max(1, n)$. Contains the computed eigenvalues, unless <code>info</code>>0. The eigenvalues are stored in the same order as on the diagonal of the Schur form T (if computed).
<code>wr, wi</code>	REAL for <code>shseqr</code> DOUBLE PRECISION for <code>dhseqr</code> Arrays, DIMENSION at least $\max(1, n)$ each. Contain the real and imaginary parts, respectively, of the computed eigenvalues, unless <code>info</code> > 0. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first. The eigenvalues are stored in the same order as on the diagonal of the Schur form T (if computed).
<code>h</code>	If <code>info</code> = 0 and <code>job</code> = 'S', <code>h</code> contains the upper triangular matrix T from the Schur decomposition (the Schur form). If <code>info</code> = 0 and <code>job</code> = 'E', the contents of <code>h</code> are unspecified on exit. (The output value of <code>h</code> when <code>info</code> > 0 is given under the description of <code>info</code> below.)
<code>z</code>	If <code>compz</code> = 'V' and <code>info</code> = 0, then <code>z</code> contains Q^*Z. If <code>compz</code> = 'I' and <code>info</code> = 0, then <code>z</code> contains the unitary or orthogonal matrix Z of the Schur vectors of H. If <code>compz</code> = 'N', then <code>z</code> is not referenced.
<code>work(1)</code>	On exit, if <code>info</code> = 0, then <code>work(1)</code> returns the optimal <code>lwork</code>.
<code>info</code>	INTEGER. If <code>info</code> = 0, the execution is successful. If <code>info</code> = $-i$, the i-th parameter had an illegal value.

If $info = i$, elements $1, 2, \dots, ilo-1$ and $i+1, i+2, \dots, n$ of wr and wi contain the real and imaginary parts of those eigenvalues that have been successively found.

If $info > 0$, and $job = 'E'$, then on exit, the remaining unconverged eigenvalues are the eigenvalues of the upper Hessenberg matrix rows and columns ilo through $info$ of the final output value of H .

If $info > 0$, and $job = 'S'$, then on exit (initial value of H)* $U = U$ *(final value of H), where U is a unitary matrix. The final value of H is upper Hessenberg and triangular in rows and columns $info+1$ through ihi .

If $info > 0$, and $compz = 'V'$, then on exit (final value of Z) = (initial value of Z)* U , where U is the unitary matrix (regardless of the value of job).

If $info > 0$, and $compz = 'I'$, then on exit (final value of Z) = U , where U is the unitary matrix (regardless of the value of job).

If $info > 0$, and $compz = 'N'$, then Z is not accessed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hseqr` interface are the following:

<code>h</code>	Holds the matrix H of size (n, n) .
<code>wr</code>	Holds the vector of length (n) . Used in real flavors only.
<code>wi</code>	Holds the vector of length (n) . Used in real flavors only.
<code>w</code>	Holds the vector of length (n) . Used in complex flavors only.
<code>z</code>	Holds the matrix Z of size (n, n) .
<code>job</code>	Must be <code>'E'</code> or <code>'S'</code> . The default value is <code>'E'</code> .
<code>compz</code>	If omitted, this argument is restored based on the presence of argument z as follows: <code>compz = 'I'</code> , if z is present, <code>compz = 'N'</code> , if z is omitted. If present, <code>compz</code> must be equal to <code>'I'</code> or <code>'V'</code> and the argument z must also be present. Note that there will be an error condition if <code>compz</code> is present and z omitted.

Application Notes

The computed Schur factorization is the exact factorization of a nearby matrix $H + E$, where $\|E\|_2 < O(\epsilon) \|H\|_2 / s_i$, and ϵ is the machine precision.

If λ_i is an exact eigenvalue, and μ_i is the corresponding computed value, then $|\lambda_i - \mu_i| \leq c(n) \epsilon \|H\|_2 / s_i$, where $c(n)$ is a modestly increasing function of n , and s_i is the reciprocal condition number of λ_i . The condition numbers s_i may be computed by calling [?trsna](#).

The total number of floating-point operations depends on how rapidly the algorithm converges; typical numbers are as follows.

If only eigenvalues are computed: $7n^3$ for real flavors
 $25n^3$ for complex flavors.

If the Schur form is computed: $10n^3$ for real flavors
 $35n^3$ for complex flavors.

If the full Schur factorization is computed: $20n^3$ for real flavors
 $70n^3$ for complex flavors.

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible `lwork` sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hsein

Computes selected eigenvectors of an upper Hessenberg matrix that correspond to specified eigenvalues.

Syntax

FORTRAN 77:

```
call shsein(job, eigsrc, initv, select, n, h, ldh, wr, wi, vl, ldvl, vr, ldvr,
mm, m, work, ifaill, ifailr, info)

call dhsein(job, eigsrc, initv, select, n, h, ldh, wr, wi, vl, ldvl, vr, ldvr,
mm, m, work, ifaill, ifailr, info)

call chsein(job, eigsrc, initv, select, n, h, ldh, w, vl, ldvl, vr, ldvr, mm,
m, work, rwork, ifaill, ifailr, info)

call zhsein(job, eigsrc, initv, select, n, h, ldh, w, vl, ldvl, vr, ldvr, mm,
m, work, rwork, ifaill, ifailr, info)
```

Fortran 95:

```
call hsein(h, wr, wi, select [, vl] [,vr] [,ifaill] [,ifailr] [,initv]
[,eigsrc] [,m] [,info])

call hsein(h, w, select [,vl] [,vr] [,ifaill] [,ifailr] [,initv] [,eigsrc]
[,m] [,info])
```

Description

This routine computes left and/or right eigenvectors of an upper Hessenberg matrix H , corresponding to selected eigenvalues.

The right eigenvector x and the left eigenvector y , corresponding to an eigenvalue λ , are defined by: $H^*x = \lambda^*x$ and $y^H H = \lambda^* y^H$ (or $H^H y = \lambda^* y$). Here λ^* denotes the conjugate of λ .

The eigenvectors are computed by inverse iteration. They are scaled so that, for a real eigenvector x , $\max |x_i| = 1$, and for a complex eigenvector, $\max (|\operatorname{Re} x_i| + |\operatorname{Im} x_i|) = 1$.

If H has been formed by reduction of a general matrix A to upper Hessenberg form, then eigenvectors of H may be transformed to eigenvectors of A by [?ormhr](#) or [?unmhr](#).

Input Parameters

<i>job</i>	CHARACTER*1. Must be 'R' or 'L' or 'B'. If <i>job</i> = 'R', then only right eigenvectors are computed. If <i>job</i> = 'L', then only left eigenvectors are computed. If <i>job</i> = 'B', then all eigenvectors are computed.
<i>eigsrc</i>	CHARACTER*1. Must be 'Q' or 'N'. If <i>eigsrc</i> = 'Q', then the eigenvalues of <i>H</i> were found using ?hseqr; thus if <i>H</i> has any zero sub-diagonal elements (and so is block triangular), then the <i>j</i>-th eigenvalue can be assumed to be an eigenvalue of the block containing the <i>j</i>-th row/column. This property allows the routine to perform inverse iteration on just one diagonal block. If <i>eigsrc</i> = 'N', then no such assumption is made and the routine performs inverse iteration using the whole matrix.
<i>initv</i>	CHARACTER*1. Must be 'N' or 'U'. If <i>initv</i> = 'N', then no initial estimates for the selected eigenvectors are supplied. If <i>initv</i> = 'U', then initial estimates for the selected eigenvectors are supplied in <i>vl</i> and/or <i>vr</i>.
<i>select</i>	LOGICAL. Array, DIMENSION at least max (1, <i>n</i>). Specifies which eigenvectors are to be computed. <i>For real flavors:</i> To obtain the real eigenvector corresponding to the real eigenvalue <i>wr(j)</i>, set <i>select(j)</i> to .TRUE. To select the complex eigenvector corresponding to the complex eigenvalue (<i>wr(j)</i>, <i>wi(j)</i>) with complex conjugate (<i>wr(j+1)</i>, <i>wi(j+1)</i>), set <i>select(j)</i> and/or <i>select(j+1)</i> to .TRUE.; the eigenvector corresponding to the first eigenvalue in the pair is computed. <i>For complex flavors:</i> To select the eigenvector corresponding to the eigenvalue <i>w(j)</i>, set <i>select(j)</i> to .TRUE.
<i>n</i>	INTEGER. The order of the matrix <i>H</i> (<i>n</i> ≥ 0).
<i>h</i> , <i>vl</i> , <i>vr</i> ,	REAL for shsein DOUBLE PRECISION for dhsein

COMPLEX for chsein
DOUBLE COMPLEX for zhsein.

Arrays:
 $h(ldh,*)$ The n -by- n upper Hessenberg matrix H .
The second dimension of h must be at least $\max(1, n)$.
 $(ldvl,*)$
If $initv = 'V'$ and $job = 'L'$ or $'B'$, then vl must contain starting vectors for inverse iteration for the left eigenvectors. Each starting vector must be stored in the same column or columns as will be used to store the corresponding eigenvector.
If $initv = 'N'$, then vl need not be set.
The second dimension of vl must be at least $\max(1, mm)$ if $job = 'L'$ or $'B'$ and at least 1 if $job = 'R'$.
The array vl is not referenced if $job = 'R'$.
 $vr(ldvr,*)$
If $initv = 'V'$ and $job = 'R'$ or $'B'$, then vr must contain starting vectors for inverse iteration for the right eigenvectors. Each starting vector must be stored in the same column or columns as will be used to store the corresponding eigenvector.
If $initv = 'N'$, then vr need not be set.
The second dimension of vr must be at least $\max(1, mm)$ if $job = 'R'$ or $'B'$ and at least 1 if $job = 'L'$.
The array vr is not referenced if $job = 'L'$.
 $work(*)$ is a workspace array.
DIMENSION at least $\max(1, n*(n+2))$ for real flavors and at least $\max(1, n*n)$ for complex flavors.

ldh INTEGER. The first dimension of h ; at least $\max(1, n)$.

w COMPLEX for chsein
DOUBLE COMPLEX for zhsein.
Array, DIMENSION at least $\max(1, n)$.
Contains the eigenvalues of the matrix H .
If $eigsrc = 'Q'$, the array must be exactly as returned by ?hseqr.

wr, wi REAL for shsein
DOUBLE PRECISION for dhsein
Arrays, DIMENSION at least $\max(1, n)$ each.

	Contain the real and imaginary parts, respectively, of the eigenvalues of the matrix H . Complex conjugate pairs of values must be stored in consecutive elements of the arrays. If $eigsrc = 'Q'$, the arrays must be exactly as returned by ?hseqr.
<i>ldvl</i>	INTEGER. The first dimension of vl . If $job = 'L'$ or $'B'$, $ldvl \geq \max(1, n)$. If $job = 'R'$, $ldvl \geq 1$.
<i>ldvr</i>	INTEGER. The first dimension of vr . If $job = 'R'$ or $'B'$, $ldvr \geq \max(1, n)$. If $job = 'L'$, $ldvr \geq 1$.
<i>mm</i>	INTEGER. The number of columns in vl and/or vr . Must be at least m , the actual number of columns required (see <i>Output Parameters</i> below). For real flavors, m is obtained by counting 1 for each selected real eigenvector and 2 for each selected complex eigenvector (see <i>select</i>). For complex flavors, m is the number of selected eigenvectors (see <i>select</i>). Constraint: $0 \leq mm \leq n$.
<i>rwork</i>	REAL for chsein DOUBLE PRECISION for zhsein. Array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>select</i>	Overwritten for real flavors only. If a complex eigenvector was selected as specified above, then $select(j)$ is set to <code>.TRUE.</code> and $select(j+1)$ to <code>.FALSE.</code>
<i>w</i>	The real parts of some elements of w may be modified, as close eigenvalues are perturbed slightly in searching for independent eigenvectors.

<i>wr</i>	Some elements of <i>wr</i> may be modified, as close eigenvalues are perturbed slightly in searching for independent eigenvectors.
<i>vl, vr</i>	If <i>job</i> = 'L' or 'B', <i>vl</i> contains the computed left eigenvectors (as specified by <i>select</i>). If <i>job</i> = 'R' or 'B', <i>vr</i> contains the computed right eigenvectors (as specified by <i>select</i>). The eigenvectors are stored consecutively in the columns of the array, in the same order as their eigenvalues. <i>For real flavors</i>: a real eigenvector corresponding to a selected real eigenvalue occupies one column; a complex eigenvector corresponding to a selected complex eigenvalue occupies two columns: the first column holds the real part and the second column holds the imaginary part.
<i>m</i>	INTEGER. <i>For real flavors</i>: the number of columns of <i>vl</i> and/or <i>vr</i> required to store the selected eigenvectors. <i>For complex flavors</i>: the number of selected eigenvectors.
<i>ifaill, ifailr</i>	INTEGER. Arrays, DIMENSION at least $\max(1, mm)$ each. <i>ifaill</i>(<i>i</i>) = 0 if the <i>i</i>th column of <i>vl</i> converged; <i>ifaill</i>(<i>i</i>) = <i>j</i> > 0 if the eigenvector stored in the <i>i</i>-th column of <i>vl</i> (corresponding to the <i>j</i>th eigenvalue) failed to converge. <i>ifailr</i>(<i>i</i>) = 0 if the <i>i</i>th column of <i>vr</i> converged; <i>ifailr</i>(<i>i</i>) = <i>j</i> > 0 if the eigenvector stored in the <i>i</i>-th column of <i>vr</i> (corresponding to the <i>j</i>th eigenvalue) failed to converge. <i>For real flavors</i>: if the <i>i</i>th and (<i>i</i>+1)th columns of <i>vl</i> contain a selected complex eigenvector, then <i>ifaill</i>(<i>i</i>) and <i>ifaill</i>(<i>i</i>+1) are set to the same value. A similar rule holds for <i>vr</i> and <i>ifailr</i>. The array <i>ifaill</i> is not referenced if <i>job</i> = 'R'. The array <i>ifailr</i> is not referenced if <i>job</i> = 'L'.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.

If $info > 0$, then i eigenvectors (as indicated by the parameters *ifaill* and/or *ifailr* above) failed to converge. The corresponding columns of *vl* and/or *vr* contain no useful information.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hsein` interface are the following:

<i>h</i>	Holds the matrix H of size (n, n) .
<i>wr</i>	Holds the vector of length (n) . Used in real flavors only.
<i>wi</i>	Holds the vector of length (n) . Used in real flavors only.
<i>w</i>	Holds the vector of length (n) . Used in complex flavors only.
<i>select</i>	Holds the vector of length (n) .
<i>vl</i>	Holds the matrix VL of size (n, mm) .
<i>vr</i>	Holds the matrix VR of size (n, mm) .
<i>ifaill</i>	Holds the vector of length (mm) . Note that there will be an error condition if <i>ifaill</i> is present and <i>vl</i> is omitted.
<i>ifailr</i>	Holds the vector of length (mm) . Note that there will be an error condition if <i>ifailr</i> is present and <i>vr</i> is omitted.
<i>initv</i>	Must be 'N' or 'U'. The default value is 'N'.
<i>eigsrc</i>	Must be 'N' or 'Q'. The default value is 'N'.
<i>job</i>	Restored based on the presence of arguments <i>vl</i> and <i>vr</i> as follows: $job = 'B'$, if both <i>vl</i> and <i>vr</i> are present, $job = 'L'$, if <i>vl</i> is present and <i>vr</i> omitted, $job = 'R'$, if <i>vl</i> is omitted and <i>vr</i> present, Note that there will be an error condition if both <i>vl</i> and <i>vr</i> are omitted.

Application Notes

Each computed right eigenvector x_i is the exact eigenvector of a nearby matrix $A + E_i$, such that $\|E_i\| < O(\epsilon) \|A\|$. Hence the residual is small:

$$\|Ax_i - \lambda_i x_i\| = O(\epsilon) \|A\|.$$

However, eigenvectors corresponding to close or coincident eigenvalues may not accurately span the relevant subspaces.

Similar remarks apply to computed left eigenvectors.

?trevc

Computes selected eigenvectors of an upper (quasi-) triangular matrix computed by ?hseqr.

Syntax

FORTRAN 77:

```
call strevc(side, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, mm, m, work,
info)

call dtrevc(side, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, mm, m, work,
info)

call ctrevc(side, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, mm, m, work,
rwork, info)

call ztrevc(side, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, mm, m, work,
rwork, info)
```

Fortran 95:

```
call trevc(t [, howmny] [,select] [,vl] [,vr] [,m] [,info])
```

Description

This routine computes some or all of the right and/or left eigenvectors of an upper triangular matrix T (or, for real flavors, an upper quasi-triangular matrix T). Matrices of this type are produced by the Schur factorization of a general matrix: $A = Q^* T^* Q^H$, as computed by ?hseqr.

The right eigenvector x and the left eigenvector y of T corresponding to an eigenvalue w , are defined by:

$$T^* x = w^* x, \quad y^H T = w^* y^H, \quad \text{where } y^H \text{ denotes the conjugate transpose of } y.$$

The eigenvalues are not input to this routine, but are read directly from the diagonal blocks of T .

This routine returns the matrices X and/or Y of right and left eigenvectors of T , or the products $Q^* X$ and/or $Q^* Y$, where Q is an input matrix.

If Q is the orthogonal/unitary factor that reduces a matrix A to Schur form T , then Q^*X and Q^*Y are the matrices of right and left eigenvectors of A .

Input Parameters

<i>side</i>	CHARACTER*1. Must be 'R' or 'L' or 'B'. If <i>side</i> = 'R' , then only right eigenvectors are computed. If <i>side</i> = 'L' , then only left eigenvectors are computed. If <i>side</i> = 'B' , then all eigenvectors are computed.
<i>howmny</i>	CHARACTER*1. Must be 'A' or 'B' or 'S'. If <i>howmny</i> = 'A' , then all eigenvectors (as specified by <i>side</i>) are computed. If <i>howmny</i> = 'B' , then all eigenvectors (as specified by <i>side</i>) are computed and backtransformed by the matrices supplied in <i>vl</i> and <i>vr</i>. If <i>howmny</i> = 'S' , then selected eigenvectors (as specified by <i>side</i> and <i>select</i>) are computed.
<i>select</i>	LOGICAL. Array, DIMENSION at least max (1, <i>n</i>). If <i>howmny</i> = 'S' , <i>select</i> specifies which eigenvectors are to be computed. If <i>howmny</i> = 'A' or 'B' , <i>select</i> is not referenced. For real flavors: If <i>omega</i>(<i>j</i>) is a real eigenvalue, the corresponding real eigenvector is computed if <i>select</i>(<i>j</i>) is .TRUE.. If <i>omega</i>(<i>j</i>) and <i>omega</i>(<i>j</i>+1) are the real and imaginary parts of a complex eigenvalue, the corresponding complex eigenvector is computed if either <i>select</i>(<i>j</i>) or <i>select</i>(<i>j</i>+1) is .TRUE., and on exit <i>select</i>(<i>j</i>) is set to .TRUE. and <i>select</i>(<i>j</i>+1) is set to .FALSE.. For complex flavors: The eigenvector corresponding to the <i>j</i>-th eigenvalue is computed if <i>select</i>(<i>j</i>) is .TRUE..
<i>n</i>	INTEGER. The order of the matrix T ($n \geq 0$).
<i>t</i> , <i>vl</i> , <i>vr</i> ,	REAL for <i>strevc</i> DOUBLE PRECISION for <i>dtrevc</i> COMPLEX for <i>ctrevc</i> DOUBLE COMPLEX for <i>ztrevc</i>.

Arrays:

$t(ldt,*)$ contains the n -by- n matrix T in Schur canonical form.

The second dimension of t must be at least $\max(1, n)$.

$vl(ldvl,*)$

If $howmny = 'B'$ and $side = 'L'$ or $'B'$, then vl must contain an n -by- n matrix Q (usually the matrix of Schur vectors returned by `?hseqr`).

If $howmny = 'A'$ or $'S'$, then vl need not be set.

The second dimension of vl must be at least $\max(1, mm)$ if $side = 'L'$ or $'B'$ and at least 1 if $side = 'R'$.

The array vl is not referenced if $side = 'R'$.

$vr(ldvr,*)$

If $howmny = 'B'$ and $side = 'R'$ or $'B'$, then vr must contain an n -by- n matrix Q (usually the matrix of Schur vectors returned by `?hseqr`). .

If $howmny = 'A'$ or $'S'$, then vr need not be set.

The second dimension of vr must be at least $\max(1, mm)$ if $side = 'R'$ or $'B'$ and at least 1 if $side = 'L'$.

The array vr is not referenced if $side = 'L'$.

$work(*)$ is a workspace array.

DIMENSION at least $\max(1, 3*n)$ for real flavors and at least $\max(1, 2*n)$ for complex flavors.

ldt INTEGER. The first dimension of t ; at least $\max(1, n)$.

$ldvl$ INTEGER. The first dimension of vl .

If $side = 'L'$ or $'B'$, $ldvl \geq n$.

If $side = 'R'$, $ldvl \geq 1$.

$ldvr$ INTEGER. The first dimension of vr .

If $side = 'R'$ or $'B'$, $ldvr \geq n$.

If $side = 'L'$, $ldvr \geq 1$.

mm INTEGER. The number of columns in the arrays vl and/or vr . Must be at least m (the precise number of columns required).

If $howmny = 'A'$ or $'B'$, $m = n$.

If *howmny* = 'S': for real flavors, *m* is obtained by counting 1 for each selected real eigenvector and 2 for each selected complex eigenvector;
for complex flavors, *m* is the number of selected eigenvectors (see *select*).

Constraint: $0 \leq m \leq n$.

rwork

REAL for ctrevc
DOUBLE PRECISION for ztrevc.
Workspace array, DIMENSION at least max (1, *n*).

Output Parameters

select

If a complex eigenvector of a real matrix was selected as specified above, then *select*(*j*) is set to .TRUE. and *select*(*j*+1) to .FALSE.

vl, *vr*

If *side* = 'L' or 'B', *vl* contains the computed left eigenvectors (as specified by *howmny* and *select*).
If *side* = 'R' or 'B', *vr* contains the computed right eigenvectors (as specified by *howmny* and *select*).
The eigenvectors are stored consecutively in the columns of the array, in the same order as their eigenvalues.
For real flavors: corresponding to each real eigenvalue is a real eigenvector, occupying one column; corresponding to each complex conjugate pair of eigenvalues is a complex eigenvector, occupying two columns; the first column holds the real part and the second column holds the imaginary part.

m

INTEGER.
For complex flavors: the number of selected eigenvectors.
If *howmny* = 'A' or 'B', *m* is set to *n*.
For real flavors: the number of columns of *vl* and/or *vr* actually used to store the selected eigenvectors.
If *howmny* = 'A' or 'B', *m* is set to *n*.

info

INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `trevc` interface are the following:

<code>t</code>	Holds the matrix T of size (n, n) .
<code>select</code>	Holds the vector of length (n) .
<code>vl</code>	Holds the matrix VL of size (n, mm) .
<code>vr</code>	Holds the matrix VR of size (n, mm) .
<code>side</code>	If omitted, this argument is restored based on the presence of arguments <code>vl</code> and <code>vr</code> as follows: <code>side = 'B'</code>, if both <code>vl</code> and <code>vr</code> are present, <code>side = 'L'</code>, if <code>vr</code> is omitted, <code>side = 'R'</code>, if <code>vl</code> is omitted. Note that there will be an error condition if both <code>vl</code> and <code>vr</code> are omitted.
<code>howmny</code>	If omitted, this argument is restored based on the presence of argument <code>select</code> as follows: <code>howmny = 'V'</code>, if <code>q</code> is present, <code>howmny = 'N'</code>, if <code>q</code> is omitted. If present, <code>vect = 'V'</code> or <code>'U'</code> and the argument <code>q</code> must also be present. Note that there will be an error condition if both <code>select</code> and <code>howmny</code> are present.

Application Notes

If x_i is an exact right eigenvector and y_i is the corresponding computed eigenvector, then the angle $\theta(y_i, x_i)$ between them is bounded as follows: $\theta(y_i, x_i) \leq (c(n)\epsilon \|T\|_2) / \text{sep}_i$ where sep_i is the reciprocal condition number of x_i . The condition number sep_i may be computed by calling `?trsna`.

?trsna

Estimates condition numbers for specified eigenvalues and right eigenvectors of an upper (quasi-) triangular matrix.

Syntax

FORTRAN 77:

```
call strsna(job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, s, sep, mm,
m, work, ldwork, iwork, info)

call dtrsna(job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, s, sep, mm,
m, work, ldwork, iwork, info)

call ctrsna(job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, s, sep, mm,
m, work, ldwork, rwork, info)

call ztrsna(job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, s, sep, mm,
m, work, ldwork, rwork, info)
```

Fortran 95:

```
call trsna(t [, s] [,sep] [,vl] [,vr] [,select] [,m] [,info])
```

Description

This routine estimates condition numbers for specified eigenvalues and/or right eigenvectors of an upper triangular matrix T (or, for real flavors, upper quasi-triangular matrix T in canonical Schur form). These are the same as the condition numbers of the eigenvalues and right eigenvectors of an original matrix $A = Z^* T^* Z^H$ (with unitary or, for real flavors, orthogonal Z), from which T may have been derived.

The routine computes the reciprocal of the condition number of an eigenvalue $\lambda(i)$ as $s_i = |v_H^* u| / (||u||_E ||v||_E)$, where u and v are the right and left eigenvectors of T , respectively, corresponding to $\lambda(i)$. This reciprocal condition number always lies between zero (ill-conditioned) and one (well-conditioned).

An approximate error estimate for a computed eigenvalue $\lambda(i)$ is then given by $\epsilon * ||T|| / s_i$, where ϵ is the *machine precision*.

To estimate the reciprocal of the condition number of the right eigenvector corresponding to $\lambda(i)$, the routine first calls [?trexc](#) to reorder the eigenvalues so that $\lambda(i)$ is in the leading position:

$$T = Q \begin{bmatrix} \lambda_i & C^H \\ 0 & T_{22} \end{bmatrix} Q^H$$

The reciprocal condition number of the eigenvector is then estimated as sep_i , the smallest singular value of the matrix $T_{22} - \lambda_i I$. This number ranges from zero (ill-conditioned) to very large (well-conditioned).

An approximate error estimate for a computed right eigenvector u corresponding to λ_i is then given by $\epsilon^* ||T|| / sep_i$.

Input Parameters

<i>job</i>	CHARACTER*1. Must be 'E' or 'V' or 'B'. If <i>job</i> = 'E', then condition numbers for eigenvalues only are computed. If <i>job</i> = 'V', then condition numbers for eigenvectors only are computed. If <i>job</i> = 'B', then condition numbers for both eigenvalues and eigenvectors are computed.
<i>howmny</i>	CHARACTER*1. Must be 'A' or 'S'. If <i>howmny</i> = 'A', then the condition numbers for all eigenpairs are computed. If <i>howmny</i> = 'S', then condition numbers for selected eigenpairs (as specified by <i>select</i>) are computed.
<i>select</i>	LOGICAL. Array, DIMENSION at least $\max(1, n)$ if <i>howmny</i> = 'S' and at least 1 otherwise. Specifies the eigenpairs for which condition numbers are to be computed if <i>howmny</i> = 'S'. <i>For real flavors:</i> To select condition numbers for the eigenpair corresponding to the real eigenvalue $\lambda(j)$, <i>select</i>(<i>j</i>) must be set .TRUE.;

to select condition numbers for the eigenpair corresponding to a complex conjugate pair of eigenvalues $\lambda(j)$ and $\lambda(j+1)$, $\text{select}(j)$ and/or $\text{select}(j+1)$ must be set `.TRUE.`

For complex flavors

To select condition numbers for the eigenpair corresponding to the eigenvalue $\lambda(j)$, $\text{select}(j)$ must be set `.TRUE.`
 select is not referenced if $\text{howmny} = 'A'$.

n

INTEGER. The order of the matrix T ($n \geq 0$).

$t, vl, vr, work$

REAL for `strsna`

DOUBLE PRECISION for `dtrsna`

COMPLEX for `ctrsna`

DOUBLE COMPLEX for `ztrsna`.

Arrays:

$t(ldt,*)$ contains the n -by- n matrix T .

The second dimension of t must be at least $\max(1, n)$.

$vl(ldvl,*)$

If $\text{job} = 'E'$ or $'B'$, then vl must contain the left eigenvectors of T (or of any matrix $Q^*T^*Q^H$ with Q unitary or orthogonal) corresponding to the eigenpairs specified by howmny and select . The eigenvectors must be stored in consecutive columns of vl , as returned by [?trevc](#) or [?hsein](#).

The second dimension of vl must be at least $\max(1, mm)$ if $\text{job} = 'E'$ or $'B'$ and at least 1 if $\text{job} = 'V'$.

The array vl is not referenced if $\text{job} = 'V'$.

$vr(ldvr,*)$

If $\text{job} = 'E'$ or $'B'$, then vr must contain the right eigenvectors of T (or of any matrix $Q^*T^*Q^H$ with Q unitary or orthogonal) corresponding to the eigenpairs specified by howmny and select . The eigenvectors must be stored in consecutive columns of vr , as returned by [?trevc](#) or [?hsein](#).

The second dimension of vr must be at least $\max(1, mm)$ if $\text{job} = 'E'$ or $'B'$ and at least 1 if $\text{job} = 'V'$.

The array vr is not referenced if $\text{job} = 'V'$.

$work$ is a workspace array, its dimension $(ldwork, n+6)$.

	The array <i>work</i> is not referenced if <i>job</i> = 'E'.
<i>ldt</i>	INTEGER. The first dimension of <i>t</i> ; at least $\max(1, n)$.
<i>ldvl</i>	INTEGER. The first dimension of <i>vl</i> . If <i>job</i> = 'E' or 'B', $ldvl \geq \max(1, n)$. If <i>job</i> = 'V', $ldvl \geq 1$.
<i>ldvr</i>	INTEGER. The first dimension of <i>vr</i> . If <i>job</i> = 'E' or 'B', $ldvr \geq \max(1, n)$. If <i>job</i> = 'R', $ldvr \geq 1$.
<i>mm</i>	INTEGER. The number of elements in the arrays <i>s</i> and <i>sep</i> , and the number of columns in <i>vl</i> and <i>vr</i> (if used). Must be at least <i>m</i> (the precise number required). If <i>howmny</i> = 'A', $m = n$; if <i>howmny</i> = 'S', for real flavors <i>m</i> is obtained by counting 1 for each selected real eigenvalue and 2 for each selected complex conjugate pair of eigenvalues. for complex flavors <i>m</i> is the number of selected eigenpairs (see <i>select</i>). Constraint: $0 \leq m \leq n$.
<i>ldwork</i>	INTEGER. The first dimension of <i>work</i> . If <i>job</i> = 'V' or 'B', $ldwork \geq \max(1, n)$. If <i>job</i> = 'E', $ldwork \geq 1$.
<i>rwork</i>	REAL for <i>ctrсна</i> , <i>ztrsna</i> . Array, DIMENSION at least $\max(1, n)$.
<i>iwork</i>	INTEGER for <i>strсна</i> , <i>dtrsna</i> . Array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>s</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Array, DIMENSION at least $\max(1, mm)$ if <i>job</i> = 'E' or 'B' and at least 1 if <i>job</i> = 'V'. Contains the reciprocal condition numbers of the selected eigenvalues if <i>job</i> = 'E' or 'B', stored in consecutive elements of the array. Thus <i>s</i> (<i>j</i>), <i>sep</i> (<i>j</i>) and the <i>j</i> -th

columns of vl and vr all correspond to the same eigenpair (but not in general the j th eigenpair unless all eigenpairs have been selected).

For real flavors: for a complex conjugate pair of eigenvalues, two consecutive elements of S are set to the same value. The array s is not referenced if $job = 'V'$.

sep

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Array, DIMENSION at least $\max(1, mm)$ if $job = 'V'$ or $'B'$ and at least 1 if $job = 'E'$. Contains the estimated reciprocal condition numbers of the selected right eigenvectors if $job = 'V'$ or $'B'$, stored in consecutive elements of the array.

For real flavors: for a complex eigenvector, two consecutive elements of sep are set to the same value; if the eigenvalues cannot be reordered to compute $sep(j)$, then $sep(j)$ is set to zero; this can only occur when the true value would be very small anyway. The array sep is not referenced if $job = 'E'$.

m

INTEGER.

For complex flavors: the number of selected eigenpairs.

If $howmny = 'A'$, m is set to n .

For real flavors: the number of elements of s and/or sep actually used to store the estimated condition numbers.

If $howmny = 'A'$, m is set to n .

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `trsna` interface are the following:

t	Holds the matrix T of size (n, n) .
s	Holds the vector of length (mm) .

<i>sep</i>	Holds the vector of length (<i>mm</i>).
<i>vl</i>	Holds the matrix <i>VL</i> of size (<i>n</i> , <i>mm</i>).
<i>vr</i>	Holds the matrix <i>VR</i> of size (<i>n</i> , <i>mm</i>).
<i>select</i>	Holds the vector of length (<i>n</i>).
<i>job</i>	Restored based on the presence of arguments <i>s</i> and <i>sep</i> as follows: <i>job</i> = 'B', if both <i>s</i> and <i>sep</i> are present, <i>job</i> = 'E', if <i>s</i> is present and <i>sep</i> omitted, <i>job</i> = 'V', if <i>s</i> is omitted and <i>sep</i> present. Note an error condition if both <i>s</i> and <i>sep</i> are omitted.
<i>howmny</i>	Restored based on the presence of the argument <i>select</i> as follows: <i>howmny</i> = 'S', if <i>select</i> is present, <i>howmny</i> = 'A', if <i>select</i> is omitted.

Note that the arguments *s*, *vl*, and *vr* must either be all present or all omitted.

Otherwise, an error condition is observed.

Application Notes

The computed values sep_i may overestimate the true value, but seldom by a factor of more than 3.

?trexc

Reorders the Schur factorization of a general matrix.

Syntax

FORTRAN 77:

```
call strexc(compq, n, t, ldt, q, ldq, ifst, ilst, work, info)
call dtrexc(compq, n, t, ldt, q, ldq, ifst, ilst, work, info)
call ctrexc(compq, n, t, ldt, q, ldq, ifst, ilst, info)
call ztrexc(compq, n, t, ldt, q, ldq, ifst, ilst, info)
```

Fortran 95:

```
call trexc(t, ifst, ilst [,q] [,info])
```

Description

This routine reorders the Schur factorization of a general matrix $A = Q^*T^*Q^H$, so that the diagonal element or block of T with row index *ifst* is moved to row *ilst*.

The reordered Schur form S is computed by an unitary (or, for real flavors, orthogonal) similarity transformation: $S = Z^H * T * Z$. Optionally the updated matrix P of Schur vectors is computed as $P = Q * Z$, giving $A = P * S * P^H$.

Input Parameters

<i>compq</i>	CHARACTER*1. Must be 'V' or 'N'. If <i>compq</i> = 'V', then the Schur vectors (Q) are updated. If <i>compq</i> = 'N', then no Schur vectors are updated.
<i>n</i>	INTEGER. The order of the matrix T ($n \geq 0$).
<i>t, q</i>	REAL for strexc DOUBLE PRECISION for dtrexc COMPLEX for ctrexc DOUBLE COMPLEX for ztrexc. Arrays: <i>t</i> (<i>ldt</i> ,*) contains the n -by- n matrix T . The second dimension of <i>t</i> must be at least max(1, n). <i>q</i> (<i>ldq</i> ,*) If <i>compq</i> = 'V', then <i>q</i> must contain Q (Schur vectors). If <i>compq</i> = 'N', then <i>q</i> is not referenced. The second dimension of <i>q</i> must be at least max(1, n) if <i>compq</i> = 'V' and at least 1 if <i>compq</i> = 'N'.
<i>ldt</i>	INTEGER. The first dimension of <i>t</i> ; at least max(1, n).
<i>ldq</i>	INTEGER. The first dimension of <i>q</i> ; If <i>compq</i> = 'N', then <i>ldq</i> $\geq$ 1. If <i>compq</i> = 'V', then <i>ldq</i> $\geq$ max(1, n).
<i>ifst, ilst</i>	INTEGER. $1 \leq ifst \leq n$; $1 \leq ilst \leq n$. Must specify the reordering of the diagonal elements (or blocks, which is possible for real flavors) of the matrix T . The element (or block) with row index <i>ifst</i> is moved to row <i>ilst</i> by a sequence of exchanges between adjacent elements (or blocks).

work REAL for `strexc`
 DOUBLE PRECISION for `dtrexc`.
 Array, DIMENSION at least $\max(1, n)$.

Output Parameters

t Overwritten by the updated matrix S .

q If `compq` = 'V', q contains the updated matrix of Schur vectors.

ifst, ilst Overwritten for real flavors only.
 If *ifst* pointed to the second row of a 2 by 2 block on entry, it is changed to point to the first row; *ilst* always points to the first row of the block in its final position (which may differ from its input value by ± 1).

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `trexc` interface are the following:

t Holds the matrix T of size (n, n) .

q Holds the matrix Q of size (n, n) .

compq Restored based on the presence of the argument q as follows:
`compq` = 'V', if q is present,
`compq` = 'N', if q is omitted.

Application Notes

The computed matrix S is exactly similar to a matrix $T+E$, where $\|E\|_2 = O(\epsilon) * \|T\|_2$, and ϵ is the machine precision.

Note that if a 2 by 2 diagonal block is involved in the re-ordering, its off-diagonal elements are in general changed; the diagonal elements and the eigenvalues of the block are unchanged unless the block is sufficiently ill-conditioned, in which case they may be noticeably altered. It is possible for a 2 by 2 block to break into two 1 by 1 blocks, that is, for a pair of complex eigenvalues to become purely real.

The values of eigenvalues however are never changed by the re-ordering.

The approximate number of floating-point operations is

for real flavors:	$6n(ifst-ilst)$ if <code>compq = 'N'</code> ;
	$12n(ifst-ilst)$ if <code>compq = 'V'</code> ;
for complex flavors:	$20n(ifst-ilst)$ if <code>compq = 'N'</code> ;
	$40n(ifst-ilst)$ if <code>compq = 'V'</code> .

?trsen

Reorders the Schur factorization of a matrix and (optionally) computes the reciprocal condition numbers and invariant subspace for the selected cluster of eigenvalues.

Syntax

FORTRAN 77:

```
call strsen(job, compq, select, n, t, ldt, q, ldq, wr, wi, m, s, sep, work,
lwork, iwork, liwork, info)

call dtrsen(job, compq, select, n, t, ldt, q, ldq, wr, wi, m, s, sep, work,
lwork, iwork, liwork, info)

call ctrsen(job, compq, select, n, t, ldt, q, ldq, w, m, s, sep, work, lwork,
info)

call ztrsen(job, compq, select, n, t, ldt, q, ldq, w, m, s, sep, work, lwork,
info)
```

Fortran 95:

```
call trsen(t, select [,wr] [,wi] [,m] [,s] [,sep] [,q] [,info])

call trsen(t, select [,w] [,m] [,s] [,sep] [,q] [,info])
```

Description

This routine reorders the Schur factorization of a general matrix $A = Q^* T^* Q^H$ so that a selected cluster of eigenvalues appears in the leading diagonal elements (or, for real flavors, diagonal blocks) of the Schur form. The reordered Schur form R is computed by an unitary (orthogonal) similarity transformation: $R = Z^H * T^* Z$. Optionally the updated matrix P of Schur vectors is computed as $P = Q^* Z$, giving $A = P^* R^* P^H$.

Let

$$R = \begin{bmatrix} T_{11} & T_{12} \\ \mathbf{0} & T_{13} \end{bmatrix}$$

where the selected eigenvalues are precisely the eigenvalues of the leading m -by- m submatrix T_{11} . Let P be correspondingly partitioned as $(Q_1 \ Q_2)$ where Q_1 consists of the first m columns of Q . Then $A^* Q_1 = Q_1^* T_{11}$, and so the m columns of Q_1 form an orthonormal basis for the invariant subspace corresponding to the selected cluster of eigenvalues.

Optionally the routine also computes estimates of the reciprocal condition numbers of the average of the cluster of eigenvalues and of the invariant subspace.

Input Parameters

<i>job</i>	CHARACTER*1. Must be 'N' or 'E' or 'V' or 'B'. If <i>job</i> = 'N', then no condition numbers are required. If <i>job</i> = 'E', then only the condition number for the cluster of eigenvalues is computed. If <i>job</i> = 'V', then only the condition number for the invariant subspace is computed. If <i>job</i> = 'B', then condition numbers for both the cluster and the invariant subspace are computed.
<i>compq</i>	CHARACTER*1. Must be 'V' or 'N'. If <i>compq</i> = 'V', then Q of the Schur vectors is updated. If <i>compq</i> = 'N', then no Schur vectors are updated.
<i>select</i>	LOGICAL. Array, DIMENSION at least max (1, n).

Specifies the eigenvalues in the selected cluster. To select an eigenvalue $\lambda(j)$, $\text{select}(j)$ must be `.TRUE.`.

For real flavors: to select a complex conjugate pair of eigenvalues $\lambda(j)$ and $\lambda(j+1)$ (corresponding 2 by 2 diagonal block), $\text{select}(j)$ and/or $\text{select}(j+1)$ must be `.TRUE.`; the complex conjugate $\lambda(j)$ and $\lambda(j+1)$ must be either both included in the cluster or both excluded.

n

INTEGER. The order of the matrix T ($n \geq 0$).

$t, q, work$

REAL for strsen

DOUBLE PRECISION for dtrsen

COMPLEX for ctrsen

DOUBLE COMPLEX for ztrsen.

Arrays:

$t(ldt,*)$ The n -by- n T .

The second dimension of t must be at least $\max(1, n)$.

$q(ldq,*)$

If $compq = 'V'$, then q must contain q of Schur vectors.

If $compq = 'N'$, then q is not referenced.

The second dimension of q must be at least $\max(1, n)$ if

$compq = 'V'$ and at least 1 if $compq = 'N'$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

ldt

INTEGER. The first dimension of t ; at least $\max(1, n)$.

ldq

INTEGER. The first dimension of q ;

If $compq = 'N'$, then $ldq \geq 1$.

If $compq = 'V'$, then $ldq \geq \max(1, n)$.

$lwork$

INTEGER. The dimension of the array $work$.

If $job = 'V'$ or $'B'$, $lwork \geq \max(1, 2*m*(n-m))$.

If $job = 'E'$, then $lwork \geq \max(1, m*(n-m))$

If $job = 'N'$, then $lwork \geq 1$ for complex flavors and

$lwork \geq \max(1, n)$ for real flavors.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the `work` array, returns this value as the first entry of the `work` array, and no error message related to `lwork` is issued by `xerbla`. See *Application Notes* for details.

`iwork`

INTEGER.`iwork`(`liwork`) is a workspace array. The array `iwork` is not referenced if `job = 'N'` or `'E'`.

The actual amount of workspace required cannot exceed $n^2/2$ if `job = 'V'` or `'B'`.

`liwork`

INTEGER.

The dimension of the array `iwork`.

If `job = 'V'` or `'B'`, $liwork \geq \max(1, 2m(n-m))$.

If `job = 'E'` or `'E'`, $liwork \geq 1$.

If `liwork = -1`, then a workspace query is assumed; the routine only calculates the optimal size of the `iwork` array, returns this value as the first entry of the `iwork` array, and no error message related to `liwork` is issued by `xerbla`. See *Application Notes* for details.

Output Parameters

`t`

Overwritten by the updated matrix R .

`q`

If `compq = 'V'`, `q` contains the updated matrix of Schur vectors; the first m columns of the `q` form an orthogonal basis for the specified invariant subspace.

`w`

COMPLEX for `ctrsen`

DOUBLE COMPLEX for `ztrsen`.

Array, DIMENSION at least $\max(1, n)$. The recorded eigenvalues of R . The eigenvalues are stored in the same order as on the diagonal of R .

`wr, wi`

REAL for `strsen`

DOUBLE PRECISION for `dtrsen`

Arrays, DIMENSION at least $\max(1, n)$. Contain the real and imaginary parts, respectively, of the reordered eigenvalues of R . The eigenvalues are stored in the same order as on

the diagonal of R . Note that if a complex eigenvalue is sufficiently ill-conditioned, then its value may differ significantly from its value before reordering.

m

INTEGER.

For complex flavors: the number of the specified invariant subspaces, which is the same as the number of selected eigenvalues (see *select*).

For real flavors: the dimension of the specified invariant subspace. The value of m is obtained by counting 1 for each selected real eigenvalue and 2 for each selected complex conjugate pair of eigenvalues (see *select*).

Constraint: $0 \leq m \leq n$.

s

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

If $job = 'E'$ or $'B'$, s is a lower bound on the reciprocal condition number of the average of the selected cluster of eigenvalues.

If $m = 0$ or n , then $s = 1$.

For real flavors: if $info = 1$, then s is set to zero. s is not referenced if $job = 'N'$ or $'V'$.

sep

REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors.

If $job = 'V'$ or $'B'$, sep is the estimated reciprocal condition number of the specified invariant subspace.

If $m = 0$ or n , then $sep = |T|$.

For real flavors: if $info = 1$, then sep is set to zero.

sep is not referenced if $job = 'N'$ or $'E'$.

$work(1)$

On exit, if $info = 0$, then $work(1)$ returns the optimal size of $lwork$.

$iwork(1)$

On exit, if $info = 0$, then $iwork(1)$ returns the optimal size of $liwork$.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `trsen` interface are the following:

<code>t</code>	Holds the matrix T of size (n, n) .
<code>select</code>	Holds the vector of length (n) .
<code>wr</code>	Holds the vector of length (n) . Used in real flavors only.
<code>wi</code>	Holds the vector of length (n) . Used in real flavors only.
<code>w</code>	Holds the vector of length (n) . Used in complex flavors only.
<code>q</code>	Holds the matrix Q of size (n, n) .
<code>compq</code>	Restored based on the presence of the argument <code>q</code> as follows: <code>compq</code> = 'V', if <code>q</code> is present, <code>compq</code> = 'N', if <code>q</code> is omitted.
<code>job</code>	Restored based on the presence of arguments <code>s</code> and <code>sep</code> as follows: <code>job</code> = 'B', if both <code>s</code> and <code>sep</code> are present, <code>job</code> = 'E', if <code>s</code> is present and <code>sep</code> omitted, <code>job</code> = 'V', if <code>s</code> is omitted and <code>sep</code> present, <code>job</code> = 'N', if both <code>s</code> and <code>sep</code> are omitted.

Application Notes

The computed matrix R is exactly similar to a matrix $T+E$, where $\|E\|_2 = O(\epsilon) \|T\|_2$, and ϵ is the machine precision. The computed s cannot underestimate the true reciprocal condition number by more than a factor of $(\min(m, n-m))_{1/2}$; `sep` may differ from the true value by $(m^*n-m^2)_{1/2}$. The angle between the computed invariant subspace and the true subspace is $O(\epsilon) \|A\|_2 / \text{sep}$. Note that if a 2-by-2 diagonal block is involved in the re-ordering, its off-diagonal elements are in general changed; the diagonal elements and the eigenvalues of the block are unchanged unless the block is sufficiently ill-conditioned, in which case they may be noticeably altered. It is possible for a 2-by-2 block to break into two 1-by-1 blocks, that is, for a pair of complex eigenvalues to become purely real. The values of eigenvalues however are never changed by the re-ordering.

If it is not clear how much workspace to supply, use a generous value of `lwork` (or `liwork`) for the first run or set `lwork` = -1 (`liwork` = -1).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If *lwork* = -1 (*liwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *lwork* (*liwork*) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?trsyl

Solves Sylvester equation for real quasi-triangular or complex triangular matrices.

Syntax

FORTRAN 77:

```
call strsyl(trana, tranb, isgn, m, n, a, lda, b, ldb, c, ldc, scale, info)
call dtrsyl(trana, tranb, isgn, m, n, a, lda, b, ldb, c, ldc, scale, info)
call ctrsyl(trana, tranb, isgn, m, n, a, lda, b, ldb, c, ldc, scale, info)
call ztrsyl(trana, tranb, isgn, m, n, a, lda, b, ldb, c, ldc, scale, info)
```

Fortran 95:

```
call trsyl(a, b, c, scale [, trana] [,tranb] [,isgn] [,info])
```

Description

This routine solves the Sylvester matrix equation $\text{op}(A) * X \pm X * \text{op}(B) = \alpha * C$, where $\text{op}(A) = A$ or A^H , and the matrices A and B are upper triangular (or, for real flavors, upper quasi-triangular in canonical Schur form); $\alpha \leq 1$ is a scale factor determined by the routine to avoid overflow in X ; A is m -by- m , B is n -by- n , and C and X are both m -by- n . The matrix X is obtained by a straightforward process of back substitution.

The equation has a unique solution if and only if $\alpha_i \pm \beta_i \neq 0$, where $\{\alpha_i\}$ and $\{\beta_i\}$ are the eigenvalues of A and B , respectively, and the sign (+ or -) is the same as that used in the equation to be solved.

Input Parameters

<i>trana</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. If <i>trana</i> = 'N', then $\text{op}(A) = A$. If <i>trana</i> = 'T', then $\text{op}(A) = A^T$ (real flavors only). If <i>trana</i> = 'C' then $\text{op}(A) = A^H$.
<i>tranb</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. If <i>tranb</i> = 'N', then $\text{op}(B) = B$. If <i>tranb</i> = 'T', then $\text{op}(B) = B^T$ (real flavors only). If <i>tranb</i> = 'C', then $\text{op}(B) = B^H$.
<i>isgn</i>	INTEGER. Indicates the form of the Sylvester equation. If <i>isgn</i> = +1, $\text{op}(A)*X + X*\text{op}(B) = \alpha*C$. If <i>isgn</i> = -1, $\text{op}(A)*X - X*\text{op}(B) = \alpha*C$.
<i>m</i>	INTEGER. The order of A , and the number of rows in X and C ($m \geq 0$).
<i>n</i>	INTEGER. The order of B , and the number of columns in X and C ($n \geq 0$).
<i>a, b, c</i>	REAL for <i>strsyl</i> DOUBLE PRECISION for <i>dtrsyl</i> COMPLEX for <i>ctrsyl</i> DOUBLE COMPLEX for <i>ztrsyl</i> . Arrays: <i>a(lda,*)</i> contains the matrix A . The second dimension of <i>a</i> must be at least $\max(1, m)$. <i>b(l db,*)</i> contains the matrix B . The second dimension of <i>b</i> must be at least $\max(1, n)$. <i>c(l dc,*)</i> contains the matrix C . The second dimension of <i>c</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; at least $\max(1, m)$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; at least $\max(1, n)$.
<i>ldc</i>	INTEGER. The first dimension of <i>c</i> ; at least $\max(1, n)$.

Output Parameters

<i>c</i>	Overwritten by the solution matrix <i>X</i> .
<i>scale</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. The value of the scale factor α .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = 1, <i>A</i> and <i>B</i> have common or close eigenvalues perturbed values were used to solve the equation.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `trsyl` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>m</i> , <i>m</i>).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>n</i>).
<i>c</i>	Holds the matrix <i>C</i> of size (<i>m</i> , <i>n</i>).
<i>trana</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>tranb</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>isgn</i>	Must be +1 or -1. The default value is +1.

Application Notes

Let *X* be the exact, *Y* the corresponding computed solution, and *R* the residual matrix: $R = C - (AY \pm YB)$. Then the residual is always small:

$$\|R\|_F = O(\epsilon) * (\|A\|_F + \|B\|_F) * \|Y\|_F.$$

However, *Y* is not necessarily the exact solution of a slightly perturbed equation; in other words, the solution is not backwards stable.

For the forward error, the following bound holds:

$$\|Y - X\|_F \leq \|R\|_F / \text{sep}(A, B)$$

but this may be a considerable overestimate. See [Golub96] for a definition of $\text{sep}(A, B)$.
The approximate number of floating-point operations for real flavors is $m^*n^*(m + n)$. For complex flavors it is 4 times greater.

Generalized Nonsymmetric Eigenvalue Problems

This section describes LAPACK routines for solving generalized nonsymmetric eigenvalue problems, reordering the generalized Schur factorization of a pair of matrices, as well as performing a number of related computational tasks.

A *generalized nonsymmetric eigenvalue problem* is as follows: given a pair of nonsymmetric (or non-Hermitian) n -by- n matrices A and B , find the *generalized eigenvalues* λ and the corresponding *generalized eigenvectors* x and y that satisfy the equations

$Ax = \lambda Bx$ (right generalized eigenvectors x)

and

$y^H A = \lambda y^H B$ (left generalized eigenvectors y).

Table 4-6 lists LAPACK routines (Fortran 77 interface) used to solve the generalized nonsymmetric eigenvalue problems and the generalized Sylvester equation. Respective routine names in Fortran 95 interface are without the first symbol (see Routine Naming Conventions).

Table 4-6 Computational Routines for Solving Generalized Nonsymmetric Eigenvalue Problems

Routine name	Operation performed
?gghrd	Reduces a pair of matrices to generalized upper Hessenberg form using orthogonal/unitary transformations.
?ggbal	Balances a pair of general real or complex matrices.
?ggbak	Forms the right or left eigenvectors of a generalized eigenvalue problem.
?hgeqz	Implements the QZ method for finding the generalized eigenvalues of the matrix pair (H,T) .
?tgevc	Computes some or all of the right and/or left generalized eigenvectors of a pair of upper triangular matrices
?tgexc	Reorders the generalized Schur decomposition of a pair of matrices (A,B) so that one diagonal block of (A,B) moves to another row index.

Routine name	Operation performed
?tgsgen	Reorders the <i>generalized</i> Schur decomposition of a pair of matrices (A,B) so that a selected cluster of eigenvalues appears in the leading diagonal blocks of (A,B).
?tgssyl	Solves the generalized Sylvester equation.
?tgssna	Estimates reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a pair of matrices in generalized real Schur canonical form.

[?ggghrd](#)

Reduces a pair of matrices to generalized upper Hessenberg form using orthogonal/unitary transformations.

Syntax

FORTRAN 77:

```
call sgghrd(compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq, z, ldz, info)
call dgghrd(compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq, z, ldz, info)
call cgghrd(compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq, z, ldz, info)
call zgghrd(compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq, z, ldz, info)
```

Fortran 95:

```
call ggghrd(a, b [,ilo] [,ihi] [,q] [,z] [,compq] [,compz] [,info])
```

Description

This routine reduces a pair of real/complex matrices (A, B) to generalized upper Hessenberg form using orthogonal/unitary transformations, where A is a general matrix and B is upper triangular. The form of the generalized eigenvalue problem is $A^*x = \lambda^*B^*x$, and B is typically made upper triangular by computing its QR factorization and moving the orthogonal matrix Q to the left side of the equation.

This routine simultaneously reduces A to a Hessenberg matrix H :

$$Q^H A Z = H$$

and transforms B to another upper triangular matrix T :

$$Q^H * B * Z = T$$

in order to reduce the problem to its standard form $H^*Y = \lambda^*T^*Y$, where $Y = Z^H * X$.

The orthogonal/unitary matrices Q and Z are determined as products of Givens rotations. They may either be formed explicitly, or they may be postmultiplied into input matrices Q_1 and Z_1 , so that

$$Q_1^* A^* Z_1^H = (Q_1^* Q)^* H^* (Z_1^* Z)^H$$

$$Q_1^* B^* Z_1^H = (Q_1^* Q)^* T^* (Z_1^* Z)^H$$

If Q_1 is the orthogonal/unitary matrix from the QR factorization of B in the original equation $A^*X = \lambda^*B^*X$, then the routine `?gghrd` reduces the original problem to generalized Hessenberg form.

Input Parameters

compq CHARACTER*1. Must be 'N', 'I', or 'V'.
 If *compq* = 'N', matrix Q is not computed.
 If *compq* = 'I', Q is initialized to the unit matrix, and the orthogonal/unitary matrix Q is returned;
 If *compq* = 'V', Q must contain an orthogonal/unitary matrix Q_1 on entry, and the product Q_1^*Q is returned.

compz CHARACTER*1. Must be 'N', 'I', or 'V'.
 If *compz* = 'N', matrix Z is not computed.
 If *compz* = 'I', Z is initialized to the unit matrix, and the orthogonal/unitary matrix Z is returned;
 If *compz* = 'V', Z must contain an orthogonal/unitary matrix Z_1 on entry, and the product Z_1^*Z is returned.

n INTEGER. The order of the matrices A and B ($n \geq 0$).

ilo, ihi INTEGER. *ilo* and *ihi* mark the rows and columns of A which are to be reduced. It is assumed that A is already upper triangular in rows and columns 1:*ilo*-1 and *ihi*+1:*n*. Values of *ilo* and *ihi* are normally set by a previous call to `?ggbal`; otherwise they should be set to 1 and *n* respectively.
 Constraint:

a, b, q, z

If $n > 0$, then $1 \leq ilo \leq ihi \leq n$;
 if $n = 0$, then $ilo = 1$ and $ihi = 0$.

REAL for sgghrd
 DOUBLE PRECISION for dgghrd
 COMPLEX for cgghrd
 DOUBLE COMPLEX for zgghrd.

Arrays:
a(*lda*,*) contains the n -by- n general matrix *A*. The second dimension of *a* must be at least $\max(1, n)$.
b(*ldb*,*) contains the n -by- n upper triangular matrix *B*. The second dimension of *b* must be at least $\max(1, n)$.
q(*ldq*,*)
 If *compq* = 'N', then *q* is not referenced.
 If *compq* = 'V', then *q* must contain the orthogonal/unitary matrix Q_1 , typically from the QR factorization of *B*. The second dimension of *q* must be at least $\max(1, n)$.
z(*ldz*,*)
 If *compq* = 'N', then *z* is not referenced.
 If *compq* = 'V', then *z* must contain the orthogonal/unitary matrix Z_1 . The second dimension of *z* must be at least $\max(1, n)$.

lda INTEGER. The first dimension of *a*; at least $\max(1, n)$.
ldb INTEGER. The first dimension of *b*; at least $\max(1, n)$.
ldq INTEGER. The first dimension of *q*;
 If *compq* = 'N', then $ldq \geq 1$.
 If *compq* = 'I' or 'V', then $ldq \geq \max(1, n)$.
ldz INTEGER. The first dimension of *z*;
 If *compq* = 'N', then $ldz \geq 1$.
 If *compq* = 'I' or 'V', then $ldz \geq \max(1, n)$.

Output Parameters

a

On exit, the upper triangle and the first subdiagonal of *A* are overwritten with the upper Hessenberg matrix H , and the rest is set to zero.

<i>b</i>	On exit, overwritten by the upper triangular matrix $T = Q^H * B * Z$. The elements below the diagonal are set to zero.
<i>q</i>	If <i>compq</i> = 'I', then <i>q</i> contains the orthogonal/unitary matrix Q ; If <i>compq</i> = 'V', then <i>q</i> is overwritten by the product $Q_1 * Q$.
<i>z</i>	If <i>compz</i> = 'I', then <i>z</i> contains the orthogonal/unitary matrix Z ; If <i>compz</i> = 'V', then <i>z</i> is overwritten by the product $Z_1 * Z$.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *gghrd* interface are the following:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>b</i>	Holds the matrix B of size (n, n) .
<i>q</i>	Holds the matrix Q of size (n, n) .
<i>z</i>	Holds the matrix Z of size (n, n) .
<i>ilo</i>	Default value for this argument is <i>ilo</i> = 1.
<i>ihi</i>	Default value for this argument is <i>ihi</i> = <i>n</i> .
<i>compq</i>	If omitted, this argument is restored based on the presence of argument <i>q</i> as follows: <i>compq</i> = 'I', if <i>q</i> is present, <i>compq</i> = 'N', if <i>q</i> is omitted. If present, <i>compq</i> must be equal to 'I' or 'V' and the argument <i>q</i> must also be present. Note that there will be an error condition if <i>compq</i> is present and <i>q</i> omitted.
<i>compz</i>	If omitted, this argument is restored based on the presence of argument <i>z</i> as follows: <i>compz</i> = 'I', if <i>z</i> is present, <i>compz</i> = 'N', if <i>z</i> is omitted. If present, <i>compz</i> must be equal to 'I' or 'V' and the argument <i>z</i> must also be present. Note that there will be an error condition if <i>compz</i> is present and <i>z</i> omitted.

?ggbal

Balances a pair of general real or complex matrices.

Syntax

FORTRAN 77:

```
call sggbal(job, n, a, lda, b, ldb, ilo, ihi, lscale, rscale, work, info)
call dggbal(job, n, a, lda, b, ldb, ilo, ihi, lscale, rscale, work, info)
call cggbal(job, n, a, lda, b, ldb, ilo, ihi, lscale, rscale, work, info)
call zggbal(job, n, a, lda, b, ldb, ilo, ihi, lscale, rscale, work, info)
```

Fortran 95:

```
call ggbal(a, b [,ilo] [,ihi] [,lscale] [,rscale] [,job] [,info])
```

Description

This routine balances a pair of general real/complex matrices (A, B). This involves, first, permuting A and B by similarity transformations to isolate eigenvalues in the first 1 to $ilo-1$ and last $ihi+1$ to n elements on the diagonal; and second, applying a diagonal similarity transformation to rows and columns ilo to ihi to make the rows and columns as close in norm as possible. Both steps are optional. Balancing may reduce the 1-norm of the matrices, and improve the accuracy of the computed eigenvalues and/or eigenvectors in the generalized eigenvalue problem $A*x = \lambda*B*x$.

Input Parameters

<i>job</i>	CHARACTER*1. Specifies the operations to be performed on A and B . Must be 'N' or 'P' or 'S' or 'B'. If <i>job</i> = 'N', then no operations are done; simply set $ilo=1$, $ihi=n$, $lscale(i)=1.0$ and $rscale(i)=1.0$ for $i = 1, \dots, n$. If <i>job</i> = 'P', then permute only. If <i>job</i> = 'S', then scale only. If <i>job</i> = 'B', then both permute and scale.
<i>n</i>	INTEGER. The order of the matrices A and B ($n \geq 0$).
<i>a, b</i>	REAL for sggbal

DOUBLE PRECISION for dggbal

COMPLEX for cggbal

DOUBLE COMPLEX for zggbal.

Arrays:

$a(lda,*)$ contains the matrix A . The second dimension of a must be at least $\max(1, n)$.

$b(ldb,*)$ contains the matrix B . The second dimension of b must be at least $\max(1, n)$.

lda INTEGER. The first dimension of a ; at least $\max(1, n)$.

ldb INTEGER. The first dimension of b ; at least $\max(1, n)$.

$work$ REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Workspace array, DIMENSION at least $\max(1, 6n)$ when $job = 'S'$ or $'B'$, or at least 1 when $job = 'N'$ or $'P'$.

Output Parameters

a, b Overwritten by the balanced matrices A and B , respectively.
If $job = 'N'$, a and b are not referenced.

ilo, ihi INTEGER. ilo and ihi are set to integers such that on exit $a(i, j) = 0$ and $b(i, j) = 0$ if $i > j$ and $j = 1, \dots, ilo-1$ or $i = ihi+1, \dots, n$.
If $job = 'N'$ or $'S'$, then $ilo = 1$ and $ihi = n$.

$lscale, rscale$ REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.

Arrays, DIMENSION at least $\max(1, n)$.

$lscale$ contains details of the permutations and scaling factors applied to the left side of A and B .

If P_j is the index of the row interchanged with row j , and D_j is the scaling factor applied to row j , then

$lscale(j) = P_j$, for $j = 1, \dots, ilo-1$

$= D_j$, for $j = ilo, \dots, ihi$

$= P_j$, for $j = ihi+1, \dots, n$.

$rscale$ contains details of the permutations and scaling factors applied to the right side of A and B .

If P_j is the index of the column interchanged with column j , and D_j is the scaling factor applied to column j , then

$rscale(j) = P_j$, for $j = 1, \dots, ilo-1$
 $= D_j$, for $j = ilo, \dots, ihi$
 $= P_j$, for $j = ihi+1, \dots, n$

The order in which the interchanges are made is n to $ihl+1$, then 1 to $ilo-1$.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ggbal` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (n, n) .
<i>b</i>	Holds the matrix <i>B</i> of size (n, n) .
<i>lscale</i>	Holds the vector of length (n) .
<i>rscale</i>	Holds the vector of length (n) .
<i>ilo</i>	Default value for this argument is <i>ilo</i> = 1.
<i>ihi</i>	Default value for this argument is <i>ihi</i> = <i>n</i> .
<i>job</i>	Must be 'B', 'S', 'P', or 'N'. The default value is 'B'.

?ggbak

Forms the right or left eigenvectors of a generalized eigenvalue problem.

Syntax

FORTRAN 77:

```

call sggbak(job, side, n, ilo, ihi, lscale, rscale, m, v, ldv, info)
call dggbak(job, side, n, ilo, ihi, lscale, rscale, m, v, ldv, info)
call cggbak(job, side, n, ilo, ihi, lscale, rscale, m, v, ldv, info)
call zggbak(job, side, n, ilo, ihi, lscale, rscale, m, v, ldv, info)

```

Fortran 95:

```
call ggbak(v [, ilo] [,ihi] [,lscale] [,rscale] [,job] [,info])
```

Description

This routine forms the right or left eigenvectors of a real/complex generalized eigenvalue problem

$$A*x = \lambda*B*x$$

by backward transformation on the computed eigenvectors of the balanced pair of matrices output by `?ggbal`.

Input Parameters

<i>job</i>	CHARACTER*1. Specifies the type of backward transformation required. Must be 'N', 'P', 'S', or 'B'. If <i>job</i> = 'N', then no operations are done; return. If <i>job</i> = 'P', then do backward transformation for permutation only. If <i>job</i> = 'S', then do backward transformation for scaling only. If <i>job</i> = 'B', then do backward transformation for both permutation and scaling. This argument must be the same as the argument <i>job</i> supplied to <code>?ggbal</code>.
<i>side</i>	CHARACTER*1. Must be 'L' or 'R'. If <i>side</i> = 'L', then <i>v</i> contains left eigenvectors. If <i>side</i> = 'R', then <i>v</i> contains right eigenvectors.
<i>n</i>	INTEGER. The number of rows of the matrix <i>v</i> ($n \geq 0$).
<i>ilo, ihi</i>	INTEGER. The integers <i>ilo</i> and <i>ihi</i> determined by <code>?ggbal</code>. Constraint: If $n > 0$, then $1 \leq ilo \leq ihi \leq n$; if $n = 0$, then $ilo = 1$ and $ihi = 0$.
<i>lscale, rscale</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least $\max(1, n)$. The array <i>lscale</i> contains details of the permutations and/or scaling factors applied to the left side of <i>A</i> and <i>B</i>, as returned by <code>?ggbal</code>.

	The array <i>rscale</i> contains details of the permutations and/or scaling factors applied to the right side of <i>A</i> and <i>B</i> , as returned by ?ggbal.
<i>m</i>	INTEGER. The number of columns of the matrix <i>V</i> ($m \geq 0$).
<i>v</i>	REAL for sggbak DOUBLE PRECISION for dggbak COMPLEX for cggbak DOUBLE COMPLEX for zggbak. Array <i>v</i> (<i>ldv</i> ,*). Contains the matrix of right or left eigenvectors to be transformed, as returned by ?tgevc. The second dimension of <i>v</i> must be at least max(1, <i>m</i>).
<i>ldv</i>	INTEGER. The first dimension of <i>v</i> ; at least max(1, <i>n</i>).

Output Parameters

<i>v</i>	Overwritten by the transformed eigenvectors
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine ggbak interface are the following:

<i>v</i>	Holds the matrix <i>V</i> of size (<i>n</i> , <i>m</i>).
<i>lscale</i>	Holds the vector of length (<i>n</i>).
<i>rscale</i>	Holds the vector of length (<i>n</i>).
<i>ilo</i>	Default value for this argument is <i>ilo</i> = 1.
<i>ihi</i>	Default value for this argument is <i>ihi</i> = <i>n</i> .
<i>job</i>	Must be 'B', 'S', 'P', or 'N'. The default value is 'B'.
<i>side</i>	If omitted, this argument is restored based on the presence of arguments <i>lscale</i> and <i>rscale</i> as follows: <i>side</i> = 'L', if <i>lscale</i> is present and <i>rscale</i> omitted,

`side = 'R'`, if `lscale` is omitted and `rscale` present.

Note that there will be an error condition if both `lscale` and `rscale` are present or if they both are omitted.

?hgeqz

Implements the QZ method for finding the generalized eigenvalues of the matrix pair (H, T) .

Syntax

FORTRAN 77:

```
call shgeqz(job, compq, compz, n, ilo, ihi, h, ldh, t, ldt, alphas, alphai,
beta, q, ldq, z, ldz, work, lwork, info)

call dhgeqz(job, compq, compz, n, ilo, ihi, h, ldh, t, ldt, alphas, alphai,
beta, q, ldq, z, ldz, work, lwork, info)

call chgeqz(job, compq, compz, n, ilo, ihi, h, ldh, t, ldt, alpha, beta, q,
ldq, z, ldz, work, lwork, rwork, info)

call zhgeqz(job, compq, compz, n, ilo, ihi, h, ldh, t, ldt, alpha, beta, q,
ldq, z, ldz, work, lwork, rwork, info)
```

Fortran 95:

```
call hgeqz(h, t [,ilo] [,ihi] [,alphar] [,alphai] [,beta] [,q] [,z] [,job]
[,compq] [,compz] [,info])

call hgeqz(h, t [,ilo] [,ihi] [,alpha] [,beta] [,q] [,z] [,job] [,compq] [,
compz] [,info])
```

Description

This routine computes the eigenvalues of a real/complex matrix pair (H, T) , where H is an upper Hessenberg matrix and T is upper triangular, using the double-shift version (for real flavors) or single-shift version (for complex flavors) of the QZ method. Matrix pairs of this type are produced by the reduction to generalized upper Hessenberg form of a real/complex matrix pair (A, B) :

$$A = Q_1^* H Z_1^H, B = Q_1^* T Z_1^H,$$

as computed by ?gghrd.

For real flavors:

If *job* = 'S', then the Hessenberg-triangular pair (*H*, *T*) is also reduced to generalized Schur form,

$$H = Q^* S^* Z^T, T = Q^* P^* Z^T,$$

where *Q* and *Z* are orthogonal matrices, *P* is an upper triangular matrix, and *S* is a quasi-triangular matrix with 1-by-1 and 2-by-2 diagonal blocks. The 1-by-1 blocks correspond to real eigenvalues of the matrix pair (*H*, *T*) and the 2-by-2 blocks correspond to complex conjugate pairs of eigenvalues.

Additionally, the 2-by-2 upper triangular diagonal blocks of *P* corresponding to 2-by-2 blocks of *S* are reduced to positive diagonal form, that is, if *S*(*j*+1,*j*) is non-zero, then *P*(*j*+1, *j*) = *P*(*j*, *j*+1) = 0, *P*(*j*, *j*) > 0, and *P*(*j*+1, *j*+1) > 0.

For complex flavors:

If *job* = 'S', then the Hessenberg-triangular pair (*H*, *T*) is also reduced to generalized Schur form,

$$H = Q^* S^* Z^H, T = Q^* P^* Z^H,$$

where *Q* and *Z* are unitary matrices, and *S* and *P* are upper triangular.

For all function flavors:

Optionally, the orthogonal/unitary matrix *Q* from the generalized Schur factorization may be postmultiplied into an input matrix *Q*₁, and the orthogonal/unitary matrix *Z* may be postmultiplied into an input matrix *Z*₁.

If *Q*₁ and *Z*₁ are the orthogonal/unitary matrices from ?gghrd that reduced the matrix pair (*A*, *B*) to generalized upper Hessenberg form, then the output matrices *Q*₁*Q* and *Z*₁*Z* are the orthogonal/unitary factors from the generalized Schur factorization of (*A*, *B*):

$$A = (Q_1 Q)^* S^* (Z_1 Z)^H, B = (Q_1 Q)^* P^* (Z_1 Z)^H.$$

To avoid overflow, eigenvalues of the matrix pair (*H*, *T*) (equivalently, of (*A*, *B*)) are computed as a pair of values (*alpha*, *beta*). For *chgeqz*/*zhgeqz*, *alpha* and *beta* are complex, and for *shgeqz*/*dhgeqz*, *alpha* is complex and *beta* real. If *beta* is nonzero, $\lambda = \alpha/\beta$ is an eigenvalue of the generalized nonsymmetric eigenvalue problem (GNEP)

$$A^* x = \lambda^* B^* x$$

and if *alpha* is nonzero, $\mu = \beta/\alpha$ is an eigenvalue of the alternate form of the GNEP

$$\mu^* A^* y = B^* y.$$

Real eigenvalues (for real flavors) or the values of *alpha* and *beta* for the *i*-th eigenvalue (for complex flavors) can be read directly from the generalized Schur form:

$$\alpha = S(i,i), \beta = P(i,i).$$

Input Parameters

<i>job</i>	CHARACTER*1. Specifies the operations to be performed. Must be 'E' or 'S'. If <i>job</i> = 'E', then compute eigenvalues only; If <i>job</i> = 'S', then compute eigenvalues and the Schur form.
<i>compq</i>	CHARACTER*1. Must be 'N', 'I', or 'V'. If <i>compq</i> = 'N', left Schur vectors (<i>q</i>) are not computed; If <i>compq</i> = 'I', <i>q</i> is initialized to the unit matrix and the matrix of left Schur vectors of (<i>H</i> , <i>T</i>) is returned; If <i>compq</i> = 'V', <i>q</i> must contain an orthogonal/unitary matrix Q_1 on entry and the product Q_1^*Q is returned.
<i>compz</i>	CHARACTER*1. Must be 'N', 'I', or 'V'. If <i>compz</i> = 'N', right Schur vectors (<i>z</i>) are not computed; If <i>compz</i> = 'I', <i>z</i> is initialized to the unit matrix and the matrix of right Schur vectors of (<i>H</i> , <i>T</i>) is returned; If <i>compz</i> = 'V', <i>z</i> must contain an orthogonal/unitary matrix Z_1 on entry and the product Z_1^*Z is returned.
<i>n</i>	INTEGER. The order of the matrices <i>H</i> , <i>T</i> , <i>Q</i> , and <i>Z</i> ($n \geq 0$).
<i>ilo, ihi</i>	INTEGER. <i>ilo</i> and <i>ihi</i> mark the rows and columns of <i>H</i> which are in Hessenberg form. It is assumed that <i>H</i> is already upper triangular in rows and columns 1: <i>ilo</i> -1 and <i>ihi</i> +1: <i>n</i> . Constraint: If $n > 0$, then $1 \leq ilo \leq ihi \leq n$; if $n = 0$, then $ilo = 1$ and $ihi = 0$.
<i>h, t, q, z, work</i>	REAL for shgeqz DOUBLE PRECISION for dhgeqz COMPLEX for chgeqz

DOUBLE COMPLEX for zhgeqz.

Arrays:

On entry, $h(ldh,*)$ contains the n -by- n upper Hessenberg matrix H .

The second dimension of h must be at least $\max(1, n)$.

On entry, $t(ldt,*)$ contains the n -by- n upper triangular matrix T .

The second dimension of t must be at least $\max(1, n)$.

$q(ldq,*)$:

On entry, if $compq = 'V'$, this array contains the orthogonal/unitary matrix Q_1 used in the reduction of (A, B) to generalized Hessenberg form.

If $compq = 'N'$, then q is not referenced.

The second dimension of q must be at least $\max(1, n)$.

$z(ldz,*)$:

On entry, if $compz = 'V'$, this array contains the orthogonal/unitary matrix Z_1 used in the reduction of (A, B) to generalized Hessenberg form.

If $compz = 'N'$, then z is not referenced.

The second dimension of z must be at least $\max(1, n)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

ldh

INTEGER. The first dimension of h ; at least $\max(1, n)$.

ldt

INTEGER. The first dimension of t ; at least $\max(1, n)$.

ldq

INTEGER. The first dimension of q ;

If $compq = 'N'$, then $ldq \geq 1$.

If $compq = 'I'$ or $'V'$, then $ldq \geq \max(1, n)$.

ldz

INTEGER. The first dimension of z ;

If $compz = 'N'$, then $ldz \geq 1$.

If $compz = 'I'$ or $'V'$, then $ldz \geq \max(1, n)$.

$lwork$

INTEGER. The dimension of the array $work$.

$lwork \geq \max(1, n)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the `work` array, returns this value as the first entry of the `work` array, and no error message related to `lwork` is issued by `xerbla`. See *Application Notes* for details.

`rwork`

REAL for `chgeqz`

DOUBLE PRECISION for `zhgeqz`.

Workspace array, DIMENSION at least $\max(1, n)$. Used in complex flavors only.

Output Parameters

`h`

For real flavors:

If `job = 'S'`, then, on exit, `h` contains the upper quasi-triangular matrix `s` from the generalized Schur factorization; 2-by-2 diagonal blocks (corresponding to complex conjugate pairs of eigenvalues) are returned in standard form, with $h(i, i) = h(i+1, i+1)$ and $h(i+1, i) * h(i, i+1) < 0$.

If `job = 'E'`, then on exit the diagonal blocks of `h` match those of `s`, but the rest of `h` is unspecified.

For complex flavors:

If `job = 'S'`, then, on exit, `h` contains the upper triangular matrix `s` from the generalized Schur factorization.

If `job = 'E'`, then on exit the diagonal of `h` matches that of `s`, but the rest of `h` is unspecified.

`t`

If `job = 'S'`, then, on exit, `t` contains the upper triangular matrix `p` from the generalized Schur factorization.

For real flavors:

2-by-2 diagonal blocks of `p` corresponding to 2-by-2 blocks of `s` are reduced to positive diagonal form, that is, if $h(j+1, j)$ is non-zero, then $t(j+1, j) = t(j, j+1) = 0$ and $t(j, j)$ and $t(j+1, j+1)$ will be positive.

If `job = 'E'`, then on exit the diagonal blocks of `t` match those of `p`, but the rest of `t` is unspecified.

For complex flavors:

if `job = 'E'`, then on exit the diagonal of `t` matches that of `p`, but the rest of `t` is unspecified.

alphar, alphai

REAL for shgeqz;
 DOUBLE PRECISION for dhgeqz.
 Arrays, DIMENSION at least $\max(1, n)$. The real and imaginary parts, respectively, of each scalar *alpha* defining an eigenvalue of GNEP.
 If *alphai*(*j*) is zero, then the *j*-th eigenvalue is real; if positive, then the *j*-th and (*j*+1)-th eigenvalues are a complex conjugate pair, with
 $\text{alphai}(j+1) = -\text{alphai}(j)$.

alpha

COMPLEX for chgeqz;
 DOUBLE COMPLEX for zhgeqz.
 Array, DIMENSION at least $\max(1, n)$.
 The complex scalars *alpha* that define the eigenvalues of GNEP. $\text{alphai}(i) = S(i, i)$ in the generalized Schur factorization.

beta

REAL for shgeqz
 DOUBLE PRECISION for dhgeqz
 COMPLEX for chgeqz
 DOUBLE COMPLEX for zhgeqz.
 Array, DIMENSION at least $\max(1, n)$.
For real flavors:
 The scalars *beta* that define the eigenvalues of GNEP. Together, the quantities $\alpha = (\text{alphar}(j), \text{alphai}(j))$ and $\beta = \text{beta}(j)$ represent the *j*-th eigenvalue of the matrix pair (*A*,*B*), in one of the forms $\lambda = \alpha/\beta$ or $\mu = \beta/\alpha$. Since either λ or μ may overflow, they should not, in general, be computed.
For complex flavors:
 The real non-negative scalars *beta* that define the eigenvalues of GNEP.
 $\text{beta}(i) = P(i, i)$ in the generalized Schur factorization. Together, the quantities $\alpha = \text{alpha}(j)$ and $\beta = \text{beta}(j)$ represent the *j*-th eigenvalue of the matrix pair (*A*,*B*), in one of the forms $\lambda = \alpha/\beta$ or $\mu = \beta/\alpha$. Since either λ or μ may overflow, they should not, in general, be computed.

<i>q</i>	On exit, if <i>compq</i> = 'I', <i>q</i> is overwritten by the orthogonal/unitary matrix of left Schur vectors of the pair (<i>H</i> , <i>T</i>), and if <i>compq</i> = 'V', <i>q</i> is overwritten by the orthogonal/unitary matrix of left Schur vectors of (<i>A</i> , <i>B</i>).
<i>z</i>	On exit, if <i>compz</i> = 'I', <i>z</i> is overwritten by the orthogonal/unitary matrix of right Schur vectors of the pair (<i>H</i> , <i>T</i>), and if <i>compz</i> = 'V', <i>z</i> is overwritten by the orthogonal/unitary matrix of right Schur vectors of (<i>A</i> , <i>B</i>).
<i>work</i> (1)	If <i>info</i> ≥ 0, on exit, <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = 1, ..., <i>n</i> , the <i>QZ</i> iteration did not converge. (<i>H</i> , <i>T</i>) is not in Schur form, but <i>alphar</i> (<i>i</i>), <i>alphai</i> (<i>i</i>) (for real flavors), <i>alpha</i> (<i>i</i>) (for complex flavors), and <i>beta</i> (<i>i</i>), <i>i</i> = <i>info</i> +1, ..., <i>n</i> should be correct. If <i>info</i> = <i>n</i> +1, ..., 2 <i>n</i> , the shift calculation failed. (<i>H</i> , <i>T</i>) is not in Schur form, but <i>alphar</i> (<i>i</i>), <i>alphai</i> (<i>i</i>) (for real flavors), <i>alpha</i> (<i>i</i>) (for complex flavors), and <i>beta</i> (<i>i</i>), <i>i</i> = <i>info</i> - <i>n</i> +1, ..., <i>n</i> should be correct.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *hgeqz* interface are the following:

<i>h</i>	Holds the matrix <i>H</i> of size (<i>n</i> , <i>n</i>).
<i>t</i>	Holds the matrix <i>T</i> of size (<i>n</i> , <i>n</i>).
<i>alphar</i>	Holds the vector of length (<i>n</i>). Used in real flavors only.
<i>alphai</i>	Holds the vector of length (<i>n</i>). Used in real flavors only.
<i>alpha</i>	Holds the vector of length (<i>n</i>). Used in complex flavors only.
<i>beta</i>	Holds the vector of length (<i>n</i>).
<i>q</i>	Holds the matrix <i>Q</i> of size (<i>n</i> , <i>n</i>).

<i>z</i>	Holds the matrix <i>z</i> of size (n, n) .
<i>ilo</i>	Default value for this argument is <i>ilo</i> = 1.
<i>ihi</i>	Default value for this argument is <i>ihi</i> = <i>n</i> .
<i>job</i>	Must be 'E' or 'S'. The default value is 'E'.
<i>compq</i>	If omitted, this argument is restored based on the presence of argument <i>q</i> as follows: <i>compq</i> = 'I', if <i>q</i> is present, <i>compq</i> = 'N', if <i>q</i> is omitted. If present, <i>compq</i> must be equal to 'I' or 'V' and the argument <i>q</i> must also be present. Note that there will be an error condition if <i>compq</i> is present and <i>q</i> omitted.
<i>compz</i>	If omitted, this argument is restored based on the presence of argument <i>z</i> as follows: <i>compz</i> = 'I', if <i>z</i> is present, <i>compz</i> = 'N', if <i>z</i> is omitted. If present, <i>compz</i> must be equal to 'I' or 'V' and the argument <i>z</i> must also be present. Note an error condition if <i>compz</i> is present and <i>z</i> is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?tgevc

Computes some or all of the right and/or left generalized eigenvectors of a pair of upper triangular matrices.

Syntax

FORTRAN 77:

```
call stgevc(side, howmny, select, n, s, lds, p, ldp, vl, ldvl, vr, ldvr, mm,
m, work, info)

call dtgevc(side, howmny, select, n, s, lds, p, ldp, vl, ldvl, vr, ldvr, mm,
m, work, info)

call ctgevc(side, howmny, select, n, s, lds, p, ldp, vl, ldvl, vr, ldvr, mm,
m, work, rwork, info)

call ztgevc(side, howmny, select, n, s, lds, p, ldp, vl, ldvl, vr, ldvr, mm,
m, work, rwork, info)
```

Fortran 95:

```
call tgevc(s, p [,howmny] [,select] [,vl] [,vr] [,m] [,info])
```

Description

This routine computes some or all of the right and/or left eigenvectors of a pair of real/complex matrices (S, P) , where S is quasi-triangular (for real flavors) or upper triangular (for complex flavors) and P is upper triangular.

Matrix pairs of this type are produced by the generalized Schur factorization of a real/complex matrix pair (A, B) :

$$A = Q^* S^* Z^H, B = Q^* P^* Z^H$$

as computed by ?gghrd plus ?hgeqz.

The right eigenvector x and the left eigenvector y of (S, P) corresponding to an eigenvalue w are defined by:

$$S^* x = w^* P^* x, y^H S = w^* y^H P$$

The eigenvalues are not input to this routine, but are computed directly from the diagonal blocks or diagonal elements of S and P .

This routine returns the matrices X and/or Y of right and left eigenvectors of (S, P) , or the products Z^*X and/or Q^*Y , where Z and Q are input matrices.

If Q and Z are the orthogonal/unitary factors from the generalized Schur factorization of a matrix pair (A, B) , then Z^*X and Q^*Y are the matrices of right and left eigenvectors of (A, B) .

Input Parameters

<i>side</i>	CHARACTER*1. Must be 'R', 'L', or 'B'. If <i>side</i> = 'R', compute right eigenvectors only. If <i>side</i> = 'L', compute left eigenvectors only. If <i>side</i> = 'B', compute both right and left eigenvectors.
<i>howmny</i>	CHARACTER*1. Must be 'A', 'B', or 'S'. If <i>howmny</i> = 'A', compute all right and/or left eigenvectors. If <i>howmny</i> = 'B', compute all right and/or left eigenvectors, backtransformed by the matrices in <i>vr</i> and/or <i>vl</i>. If <i>howmny</i> = 'S', compute selected right and/or left eigenvectors, specified by the logical array <i>select</i>.
<i>select</i>	LOGICAL. Array, DIMENSION at least max (1, <i>n</i>). If <i>howmny</i> = 'S', <i>select</i> specifies the eigenvectors to be computed. If <i>howmny</i> = 'A' or 'B', <i>select</i> is not referenced. For real flavors: If <i>omega</i>(<i>j</i>) is a real eigenvalue, the corresponding real eigenvector is computed if <i>select</i>(<i>j</i>) is .TRUE.. If <i>omega</i>(<i>j</i>) and <i>omega</i>(<i>j</i>+1) are the real and imaginary parts of a complex eigenvalue, the corresponding complex eigenvector is computed if either <i>select</i>(<i>j</i>) or <i>select</i>(<i>j</i>+1) is .TRUE., and on exit <i>select</i>(<i>j</i>) is set to .TRUE. and <i>select</i>(<i>j</i>+1) is set to .FALSE.. For complex flavors: The eigenvector corresponding to the <i>j</i>-th eigenvalue is computed if <i>select</i>(<i>j</i>) is .TRUE..
<i>n</i>	INTEGER. The order of the matrices <i>S</i> and <i>P</i> (<i>n</i> ≥ 0).
<i>s, p, vl, vr, work</i>	REAL for stgevc

DOUBLE PRECISION for dtgevc
 COMPLEX for ctgevc
 DOUBLE COMPLEX for ztgevc.

Arrays:

$s(lds,*)$ contains the matrix S from a generalized Schur factorization as computed by ?hgeqz. This matrix is upper quasi-triangular for real flavors, and upper triangular for complex flavors.

The second dimension of s must be at least $\max(1, n)$.

$p(ldp,*)$ contains the upper triangular matrix P from a generalized Schur factorization as computed by ?hgeqz.

For real flavors, 2-by-2 diagonal blocks of P corresponding to 2-by-2 blocks of S must be in positive diagonal form.

For complex flavors, P must have real diagonal elements.

The second dimension of p must be at least $\max(1, n)$.

If $side = 'L'$ or $'B'$ and $howmny = 'B'$, $vl(ldvl,*)$ must contain an n -by- n matrix Q (usually the orthogonal/unitary matrix Q of left Schur vectors returned by ?hgeqz). The second dimension of vl must be at least $\max(1, mm)$.

If $side = 'R'$, vl is not referenced.

If $side = 'R'$ or $'B'$ and $howmny = 'B'$, $vr(ldvr,*)$ must contain an n -by- n matrix Z (usually the orthogonal/unitary matrix Z of right Schur vectors returned by ?hgeqz). The second dimension of vr must be at least $\max(1, mm)$.

If $side = 'L'$, vr is not referenced.

$work(*)$ is a workspace array.

DIMENSION at least $\max(1, 6*n)$ for real flavors and at least $\max(1, 2*n)$ for complex flavors.

lds INTEGER. The first dimension of s ; at least $\max(1, n)$.

ldp INTEGER. The first dimension of p ; at least $\max(1, n)$.

$ldvl$ INTEGER. The first dimension of vl ;

If $side = 'L'$ or $'B'$, then $ldvl \geq n$.

If $side = 'R'$, then $ldvl \geq 1$.

$ldvr$ INTEGER. The first dimension of vr ;

If $side = 'R'$ or $'B'$, then $ldvr \geq n$.

If $side = 'L'$, then $ldvr \geq 1$.

mm INTEGER. The number of columns in the arrays *vl* and/or *vr* ($mm \geq m$).

rwork REAL for ctgevc DOUBLE PRECISION for ztgevc.
Workspace array, DIMENSION at least $\max(1, 2*n)$. Used in complex flavors only.

Output Parameters

vl On exit, if *side* = 'L' or 'B', *vl* contains:
if *howmny* = 'A', the matrix *Y* of left eigenvectors of (*S*,*P*);
if *howmny* = 'B', the matrix *Q***Y*;
if *howmny* = 'S', the left eigenvectors of (*S*,*P*) specified by *select*, stored consecutively in the columns of *vl*, in the same order as their eigenvalues.
For real flavors:
A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive columns, the first holding the real part, and the second the imaginary part.

vr On exit, if *side* = 'R' or 'B', *vr* contains:
if *howmny* = 'A', the matrix *X* of right eigenvectors of (*S*,*P*);
if *howmny* = 'B', the matrix *Z***X*;
if *howmny* = 'S', the right eigenvectors of (*S*,*P*) specified by *select*, stored consecutively in the columns of *vr*, in the same order as their eigenvalues.
For real flavors:
A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive columns, the first holding the real part, and the second the imaginary part.

m INTEGER. The number of columns in the arrays *vl* and/or *vr* actually used to store the eigenvectors.
If *howmny* = 'A' or 'B', *m* is set to *n*.
For real flavors:
Each selected real eigenvector occupies one column and each selected complex eigenvector occupies two columns.
For complex flavors:
Each selected eigenvector occupies one column.

info INTEGER.
If *info* = 0, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

For real flavors:

if $info = i > 0$, the 2-by-2 block ($i:i+1$) does not have a complex eigenvalue.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tgevc` interface are the following:

<i>s</i>	Holds the matrix <i>S</i> of size (n, n) .
<i>p</i>	Holds the matrix <i>P</i> of size (n, n) .
<i>select</i>	Holds the vector of length (n) .
<i>vl</i>	Holds the matrix <i>VL</i> of size (n, mm) .
<i>vr</i>	Holds the matrix <i>VR</i> of size (n, mm) .
<i>side</i>	Restored based on the presence of arguments <i>vl</i> and <i>vr</i> as follows: <i>side</i> = 'B', if both <i>vl</i> and <i>vr</i> are present, <i>side</i> = 'L', if <i>vl</i> is present and <i>vr</i> omitted, <i>side</i> = 'R', if <i>vl</i> is omitted and <i>vr</i> present, Note that there will be an error condition if both <i>vl</i> and <i>vr</i> are omitted.
<i>howmny</i>	If omitted, this argument is restored based on the presence of argument <i>select</i> as follows: <i>howmny</i> = 'S', if <i>select</i> is present, <i>howmny</i> = 'A', if <i>select</i> is omitted. If present, <i>howmny</i> must be equal to 'A' or 'B' and the argument <i>select</i> must be omitted. Note that there will be an error condition if both <i>howmny</i> and <i>select</i> are present.

?tgexc

Reorders the generalized Schur decomposition of a pair of matrices (A,B) so that one diagonal block of (A,B) moves to another row index.

Syntax

FORTRAN 77:

```
call stgexc(wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, ifst, ilst, work,
            lwork, info)

call dtgexc(wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, ifst, ilst, work,
            lwork, info)

call ctgexc(wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, ifst, ilst, info)

call ztgexc(wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, ifst, ilst, info)
```

Fortran 95:

```
call tgexc(a, b [,ifst] [,ilst] [,z] [,q] [,info])
```

Description

This routine reorders the generalized real-Schur/Schur decomposition of a real/complex matrix pair (A,B) using an orthogonal/unitary equivalence transformation

$$(A, B) = Q^* (A, B) * Z^H,$$

so that the diagonal block of (A, B) with row index *ifst* is moved to row *ilst*. Matrix pair (A, B) must be in a generalized real-Schur/Schur canonical form (as returned by ?gges), that is, A is block upper triangular with 1-by-1 and 2-by-2 diagonal blocks and B is upper triangular. Optionally, the matrices Q and Z of generalized Schur vectors are updated.

$Q(\text{in}) * A(\text{in}) * Z(\text{in})^T = Q(\text{out}) * A(\text{out}) * Z(\text{out})^T$

$Q(\text{in}) * B(\text{in}) * Z(\text{in})^T = Q(\text{out}) * B(\text{out}) * Z(\text{out})^T$.

Input Parameters

<i>wantq, wantz</i>	LOGICAL. If <i>wantq</i> = .TRUE., update the left transformation matrix Q; If <i>wantq</i> = .FALSE., do not update Q;

If *wantz* = .TRUE., update the right transformation matrix *Z*;
 If *wantz* = .FALSE., do not update *Z*.

n
 INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).

a, *b*, *q*,
 REAL for stgexc
 DOUBLE PRECISION for dtgexc
 COMPLEX for ctgexc
 DOUBLE COMPLEX for ztgexc.

Arrays:
a(*lda*,*) contains the matrix *A*.
 The second dimension of *a* must be at least $\max(1, n)$.
b(*ldb*,*) contains the matrix *B*. The second dimension of *b*
 must be at least $\max(1, n)$.
q(*ldq*,*)
 If *wantq* = .FALSE., then *q* is not referenced.
 If *wantq* = .TRUE., then *q* must contain the
 orthogonal/unitary matrix *Q*.
 The second dimension of *q* must be at least $\max(1, n)$.
z(*ldz*,*)
 If *wantz* = .FALSE., then *z* is not referenced.
 If *wantz* = .TRUE., then *z* must contain the
 orthogonal/unitary matrix *Z*.
 The second dimension of *z* must be at least $\max(1, n)$.

lda
 INTEGER. The first dimension of *a*; at least $\max(1, n)$.

ldb
 INTEGER. The first dimension of *b*; at least $\max(1, n)$.

ldq
 INTEGER. The first dimension of *q*;
 If *wantq* = .FALSE., then $ldq \geq 1$.
 If *wantq* = .TRUE., then $ldq \geq \max(1, n)$.

ldz
 INTEGER. The first dimension of *z*;
 If *wantz* = .FALSE., then $ldz \geq 1$.
 If *wantz* = .TRUE., then $ldz \geq \max(1, n)$.

ifst, *ilst*
 INTEGER. Specify the reordering of the diagonal blocks of
 (*A*, *B*). The block with row index *ifst* is moved to row *ilst*,
 by a sequence of swapping between adjacent blocks.
 Constraint: $1 \leq ifst, ilst \leq n$.

<i>work</i>	REAL for <code>stgexc</code> ; DOUBLE PRECISION for <code>dtgexc</code> . Workspace array, DIMENSION (<i>lwork</i>). Used in real flavors only.
<i>lwork</i>	INTEGER. The dimension of <i>work</i> ; must be at least $4n + 16$. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla . See <i>Application Notes</i> for details.

Output Parameters

<i>a, b</i>	Overwritten by the updated matrices <i>A</i> and <i>B</i> .
<i>ifst, ilst</i>	Overwritten for real flavors only. If <i>ifst</i> pointed to the second row of a 2 by 2 block on entry, it is changed to point to the first row; <i>ilst</i> always points to the first row of the block in its final position (which may differ from its input value by ± 1).
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = 1, the transformed matrix pair (<i>A</i> , <i>B</i>) would be too far from generalized Schur form; the problem is ill-conditioned. (<i>A</i> , <i>B</i>) may have been partially reordered, and <i>ilst</i> points to the first row of the current position of the block being moved.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tgexc` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>n</i>).
<i>z</i>	Holds the matrix <i>Z</i> of size (<i>n</i> , <i>n</i>).

q	Holds the matrix Q of size (n, n) .
$wantq$	Restored based on the presence of the argument q as follows: $wantq = .TRUE$, if q is present, $wantq = .FALSE$, if q is omitted.
$wantz$	Restored based on the presence of the argument z as follows: $wantz = .TRUE$, if z is present, $wantz = .FALSE$, if z is omitted.

Application Notes

If it is not clear how much workspace to supply, use a generous value of $lwork$ for the first run, or set $lwork = -1$.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if $lwork$ is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?tgsgen

Reorders the generalized Schur decomposition of a pair of matrices (A,B) so that a selected cluster of eigenvalues appears in the leading diagonal blocks of (A,B).

Syntax

FORTTRAN 77:

```
call stgsen(ijob, wantq, wantz, select, n, a, lda, b, ldb, alphas, alphas,
beta, q, ldq, z, ldz, m, pl, pr, dif, work, lwork, iwork, liwork, info)

call dtgsen(ijob, wantq, wantz, select, n, a, lda, b, ldb, alphas, alphas,
beta, q, ldq, z, ldz, m, pl, pr, dif, work, lwork, iwork, liwork, info)

call ctgsen(ijob, wantq, wantz, select, n, a, lda, b, ldb, alpha, beta, q,
ldq, z, ldz, m, pl, pr, dif, work, lwork, iwork, liwork, info)

call ztgsen(ijob, wantq, wantz, select, n, a, lda, b, ldb, alpha, beta, q,
ldq, z, ldz, m, pl, pr, dif, work, lwork, iwork, liwork, info)
```

Fortran 95:

```
call tgsgen(a, b, select [,alphas] [,alphas] [,beta] [,ijob] [,q] [,z] [,pl]
[,pr] [,dif] [,m] [,info])

call tgsgen(a, b, select [,alpha] [,beta] [,ijob] [,q] [,z] [,pl] [,pr] [, dif]
[,m] [,info])
```

Description

This routine reorders the generalized real-Schur/Schur decomposition of a real/complex matrix pair (A, B) (in terms of an orthogonal/unitary equivalence transformation $Q^* (A, B) Z$, so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the pair (A, B). The leading columns of Q and Z form orthonormal/unitary bases of the corresponding left and right eigenspaces (deflating subspaces).

(A, B) must be in generalized real-Schur/Schur canonical form (as returned by ?gges), that is, A and B are both upper triangular.

?tgsgen also computes the generalized eigenvalues

$\omega_j = (\text{alphas}(j) + \text{alphas}(j)*i)/\text{beta}(j)$ (for real flavors)

$\omega_j = \alpha(j)/\beta(j)$ (for complex flavors)

of the reordered matrix pair (A, B) .

Optionally, the routine computes the estimates of reciprocal condition numbers for eigenvalues and eigenspaces. These are $\text{Difu}[(A_{11}, B_{11}), (A_{22}, B_{22})]$ and $\text{Difl}[(A_{11}, B_{11}), (A_{22}, B_{22})]$, that is, the separation(s) between the matrix pairs (A_{11}, B_{11}) and (A_{22}, B_{22}) that correspond to the selected cluster and the eigenvalues outside the cluster, respectively, and norms of "projections" onto left and right eigenspaces with respect to the selected cluster in the (1,1)-block.

Input Parameters

ijob INTEGER. Specifies whether condition numbers are required for the cluster of eigenvalues (*pl* and *pr*) or the deflating subspaces *Difu* and *Difl*.
 If *ijob* = 0, only reorder with respect to *select*;
 If *ijob* = 1, reciprocal of norms of "projections" onto left and right eigenspaces with respect to the selected cluster (*pl* and *pr*);
 If *ijob* = 2, compute upper bounds on *Difu* and *Difl*, using F-norm-based estimate (*dif* (1:2));
 If *ijob* = 3, compute estimate of *Difu* and *Difl*, using 1-norm-based estimate (*dif* (1:2)). This option is about 5 times as expensive as *ijob* = 2;
 If *ijob* = 4, compute *pl*, *pr* and *dif* (i.e., options 0, 1 and 2 above). This is an economic version to get it all;
 If *ijob* = 5, compute *pl*, *pr* and *dif* (i.e., options 0, 1 and 3 above).

wantq, wantz LOGICAL.
 If *wantq* = .TRUE., update the left transformation matrix *Q*;
 If *wantq* = .FALSE., do not update *Q*;
 If *wantz* = .TRUE., update the right transformation matrix *Z*;
 If *wantz* = .FALSE., do not update *Z*.

select LOGICAL.
 Array, DIMENSION at least max (1, *n*). Specifies the eigenvalues in the selected cluster.

To select an eigenvalue $\omega(j)$, $\text{select}(j)$ must be `.TRUE.` For real flavors: to select a complex conjugate pair of eigenvalues $\omega(j)$ and $\omega(j+1)$ (corresponding 2 by 2 diagonal block), $\text{select}(j)$ and/or $\text{select}(j+1)$ must be set to `.TRUE.`; the complex conjugate $\omega(j)$ and $\omega(j+1)$ must be either both included in the cluster or both excluded.

n INTEGER. The order of the matrices A and B ($n \geq 0$).

a, b, q, z, work REAL for stgsen
DOUBLE PRECISION for dtgsen
COMPLEX for ctgsen
DOUBLE COMPLEX for ztgsen.

Arrays:
 $a(\text{lda},*)$ contains the matrix A .
 For real flavors: A is upper quasi-triangular, with (A, B) in generalized real Schur canonical form.
 For complex flavors: A is upper triangular, in generalized Schur canonical form.
 The second dimension of a must be at least $\max(1, n)$.
 $b(\text{ldb},*)$ contains the matrix B .
 For real flavors: B is upper triangular, with (A, B) in generalized real Schur canonical form.
 For complex flavors: B is upper triangular, in generalized Schur canonical form. The second dimension of b must be at least $\max(1, n)$.
 $q(\text{ldq},*)$
 If $\text{wantq} = \text{.TRUE.}$, then q is an n -by- n matrix;
 If $\text{wantq} = \text{.FALSE.}$, then q is not referenced.
 The second dimension of q must be at least $\max(1, n)$.
 $z(\text{ldz},*)$
 If $\text{wantz} = \text{.TRUE.}$, then z is an n -by- n matrix;
 If $\text{wantz} = \text{.FALSE.}$, then z is not referenced.
 The second dimension of z must be at least $\max(1, n)$.
 work is a workspace array, its dimension $\max(1, \text{lwork})$.

lda INTEGER. The first dimension of a ; at least $\max(1, n)$.

ldb INTEGER. The first dimension of b ; at least $\max(1, n)$.

<i>ldq</i>	INTEGER. The first dimension of <i>q</i>; $ldq \geq 1$. If <i>wantq</i> = <code>.TRUE.</code>, then $ldq \geq \max(1, n)$.
<i>ldz</i>	INTEGER. The first dimension of <i>z</i>; $ldz \geq 1$. If <i>wantz</i> = <code>.TRUE.</code>, then $ldz \geq \max(1, n)$.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i>. <i>For real flavors:</i> If <i>ijob</i> = 1, 2, or 4, $lwork \geq \max(4n+16, 2m(n-m))$. If <i>ijob</i> = 3 or 5, $lwork \geq \max(4n+16, 4m(n-m))$. <i>For complex flavors:</i> If <i>ijob</i> = 1, 2, or 4, $lwork \geq \max(1, 2m(n-m))$. If <i>ijob</i> = 3 or 5, $lwork \geq \max(1, 4m(n-m))$. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla. See <i>Application Notes</i> for details.
<i>iwork</i>	INTEGER. Workspace array, its dimension $\max(1, liwork)$. If <i>ijob</i> = 0, <i>iwork</i> is not referenced.
<i>liwork</i>	INTEGER. The dimension of the array <i>iwork</i>. <i>For real flavors:</i> If <i>ijob</i> = 1, 2, or 4, $liwork \geq n+6$. If <i>ijob</i> = 3 or 5, $liwork \geq \max(n+6, 2m(n-m))$. <i>For complex flavors:</i> If <i>ijob</i> = 1, 2, or 4, $liwork \geq n+2$. If <i>ijob</i> = 3 or 5, $liwork \geq \max(n+2, 2m(n-m))$. If <i>liwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>iwork</i> array, returns this value as the first entry of the <i>iwork</i> array, and no error message related to <i>liwork</i> is issued by xerbla. See <i>Application Notes</i> for details.

Output Parameters

<i>a</i> , <i>b</i>	Overwritten by the reordered matrices <i>A</i> and <i>B</i> , respectively.

<i>alphar, alphai</i>	REAL for stgsen; DOUBLE PRECISION for dtgsen. Arrays, DIMENSION at least $\max(1, n)$. Contain values that form generalized eigenvalues in real flavors. See <i>beta</i>.
<i>alpha</i>	COMPLEX for ctgsen; DOUBLE COMPLEX for ztgsen. Array, DIMENSION at least $\max(1, n)$. Contain values that form generalized eigenvalues in complex flavors. See <i>beta</i>.
<i>beta</i>	REAL for stgsen DOUBLE PRECISION for dtgsen COMPLEX for ctgsen DOUBLE COMPLEX for ztgsen. Array, DIMENSION at least $\max(1, n)$. For real flavors: On exit, $(\text{alphar}(j) + \text{alphai}(j)*i)/\text{beta}(j)$, $j=1, \dots, n$, will be the generalized eigenvalues. $\text{alphar}(j) + \text{alphai}(j)*i$ and $\text{beta}(j)$, $j=1, \dots, n$ are the diagonals of the complex Schur form (S, T) that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of (A, B) were further reduced to triangular form using complex unitary transformations. If $\text{alphai}(j)$ is zero, then the j-th eigenvalue is real; if positive, then the j-th and $(j+1)$-st eigenvalues are a complex conjugate pair, with $\text{alphai}(j+1)$ negative. For complex flavors: The diagonal elements of A and B, respectively, when the pair (A, B) has been reduced to generalized Schur form. $\text{alpha}(i)/\text{beta}(i)$, $i=1, \dots, n$ are the generalized eigenvalues.
<i>q</i>	If <i>wantq</i> = .TRUE., then, on exit, <i>q</i> has been postmultiplied by the left orthogonal transformation matrix which reorder (A, B). The leading m columns of <i>q</i> form orthonormal bases for the specified pair of left eigenspaces (deflating subspaces).

<i>z</i>	If <i>wantz</i> = .TRUE., then, on exit, <i>z</i> has been postmultiplied by the left orthogonal transformation matrix which reorder (<i>A</i>, <i>B</i>). The leading <i>m</i> columns of <i>z</i> form orthonormal bases for the specified pair of left eigenspaces (deflating subspaces).
<i>m</i>	INTEGER. The dimension of the specified pair of left and right eigen-spaces (deflating subspaces); $0 \leq m \leq n$.
<i>pl, pr</i>	REAL for single precision flavors; DOUBLE PRECISION for double precision flavors. If <i>ijob</i> = 1, 4, or 5, <i>pl</i> and <i>pr</i> are lower bounds on the reciprocal of the norm of "projections" onto left and right eigenspaces with respect to the selected cluster. $0 < pl, pr \leq 1$. If <i>m</i> = 0 or <i>m</i> = <i>n</i>, <i>pl</i> = <i>pr</i> = 1. If <i>ijob</i> = 0, 2 or 3, <i>pl</i> and <i>pr</i> are not referenced
<i>dif</i>	REAL for single precision flavors; DOUBLE PRECISION for double precision flavors. Array, DIMENSION (2). If <i>ijob</i> ≥ 2, <i>dif</i>(1:2) store the estimates of <i>Difu</i> and <i>Difl</i>. If <i>ijob</i> = 2 or 4, <i>dif</i>(1:2) are F-norm-based upper bounds on <i>Difu</i> and <i>Difl</i>. If <i>ijob</i> = 3 or 5, <i>dif</i>(1:2) are 1-norm-based estimates of <i>Difu</i> and <i>Difl</i>. If <i>m</i> = 0 or <i>n</i>, <i>dif</i>(1:2) = F-norm([<i>A</i>, <i>B</i>]). If <i>ijob</i> = 0 or 1, <i>dif</i> is not referenced.
<i>work</i> (1)	If <i>ijob</i> is not 0 and <i>info</i> = 0, on exit, <i>work</i>(1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>iwork</i> (1)	If <i>ijob</i> is not 0 and <i>info</i> = 0, on exit, <i>iwork</i>(1) contains the minimum value of <i>liwork</i> required for optimum performance. Use this <i>liwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.

If *info* = 1, Reordering of (*A*, *B*) failed because the transformed matrix pair (*A*, *B*) would be too far from generalized Schur form; the problem is very ill-conditioned. (*A*, *B*) may have been partially reordered.
If requested, 0 is returned in *dif*(*), *p1* and *pr*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *tgssn* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>n</i>).
<i>select</i>	Holds the vector of length (<i>n</i>).
<i>alphar</i>	Holds the vector of length (<i>n</i>). Used in real flavors only.
<i>alphai</i>	Holds the vector of length (<i>n</i>). Used in real flavors only.
<i>alpha</i>	Holds the vector of length (<i>n</i>). Used in complex flavors only.
<i>beta</i>	Holds the vector of length (<i>n</i>).
<i>q</i>	Holds the matrix <i>Q</i> of size (<i>n</i> , <i>n</i>).
<i>z</i>	Holds the matrix <i>Z</i> of size (<i>n</i> , <i>n</i>).
<i>dif</i>	Holds the vector of length (2).
<i>ijob</i>	Must be 0, 1, 2, 3, 4, or 5. The default value is 0.
<i>wantq</i>	Restored based on the presence of the argument <i>q</i> as follows: <i>wantq</i> = .TRUE, if <i>q</i> is present, <i>wantq</i> = .FALSE, if <i>q</i> is omitted.
<i>wantz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>wantz</i> = .TRUE, if <i>z</i> is present, <i>wantz</i> = .FALSE, if <i>z</i> is omitted.

Application Notes

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = -1 (*liwork* = -1).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If *lwork* = -1 (*liwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *lwork* (*liwork*) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?tgsyl

Solves the generalized Sylvester equation.

Syntax

FORTRAN 77:

```
call stgsyl(trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf,
scale, dif, work, lwork, iwork, info)
```

```
call dtgsyl(trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf,
scale, dif, work, lwork, iwork, info)
```

```
call ctgsyl(trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf,
scale, dif, work, lwork, iwork, info)
```

```
call ztgsyl(trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf,
scale, dif, work, lwork, iwork, info)
```

Fortran 95:

```
call tgsyl(a, b, c, d, e, f [,ijob] [,trans] [,scale] [,dif] [,info])
```

Description

This routine solves the generalized Sylvester equation:

$$A^*R - L^*B = \text{scale} * C$$

$$D^*R - L^*E = \text{scale} * F$$

where R and L are unknown m -by- n matrices, (A, D) , (B, E) and (C, F) are given matrix pairs of size m -by- m , n -by- n and m -by- n , respectively, with real/complex entries. (A, D) and (B, E) must be in generalized real-Schur/Schur canonical form, that is, A, B are upper quasi-triangular/triangular and D, E are upper triangular.

The solution (R, L) overwrites (C, F) . The factor $scale$, $0 \leq scale \leq 1$, is an output scaling factor chosen to avoid overflow.

In matrix notation the above equation is equivalent to the following: solve $Z * x = scale * b$, where Z is defined as

$$Z = \begin{pmatrix} \text{kron}(I_n, A) - \text{kron}(B', I_m) \\ \text{kron}(I_n, D) - \text{kron}(E', I_m) \end{pmatrix}$$

Here I_k is the identity matrix of size k and x' is the transpose/conjugate-transpose of x . $\text{kron}(X, Y)$ is the Kronecker product between the matrices X and Y .

If $trans = 'T'$ (for real flavors), or $trans = 'C'$ (for complex flavors), the routine `?tgstyl` solves the transposed/conjugate-transposed system $Z' * y = scale * b$, which is equivalent to solve for R and L in

$$A' * R + D' * L = scale * C$$

$$R * B' + L * E' = scale * (-F)$$

This case ($trans = 'T'$ for `stgsyl/dtgsyl` or $trans = 'C'$ for `ctgsyl/ztgsyl`) is used to compute an one-norm-based estimate of $\text{Dif}[(A, D), (B, E)]$, the separation between the matrix pairs (A, D) and (B, E) , using [slacon/clacon](#).

If $ijob \geq 1$, `?tgstyl` computes a Frobenius norm-based estimate of $\text{Dif}[(A, D), (B, E)]$. That is, the reciprocal of a lower bound on the reciprocal of the smallest singular value of Z . This is a level 3 BLAS algorithm.

Input Parameters

trans CHARACTER*1. Must be 'N', 'T', or 'C'.
 If *trans* = 'N', solve the generalized Sylvester equation.
 If *trans* = 'T', solve the 'transposed' system (for real flavors only).

If *trans* = 'C', solve the 'conjugate transposed' system (for complex flavors only).

ijob INTEGER. Specifies what kind of functionality to be performed:
 If *ijob* = 0, solve the generalized Sylvester equation only;
 If *ijob* = 1, perform the functionality of *ijob* = 0 and *ijob* = 3;
 If *ijob* = 2, perform the functionality of *ijob* = 0 and *ijob* = 4;
 If *ijob* = 3, only an estimate of $\text{Dif}[(A, D), (B, E)]$ is computed (look ahead strategy is used);
 If *ijob* = 4, only an estimate of $\text{Dif}[(A, D), (B, E)]$ is computed (?gecon on sub-systems is used). If *trans* = 'T' or 'C', *ijob* is not referenced.

m INTEGER. The order of the matrices *A* and *D*, and the row dimension of the matrices *C*, *F*, *R* and *L*.

n INTEGER. The order of the matrices *B* and *E*, and the column dimension of the matrices *C*, *F*, *R* and *L*.

a, *b*, *c*, *d*, *e*, *f*, *work* REAL for stgsyl
 DOUBLE PRECISION for dtgsyl
 COMPLEX for ctgsyl
 DOUBLE COMPLEX for ztgsyl.

Arrays:
a(*lda*,*) contains the upper quasi-triangular (for real flavors) or upper triangular (for complex flavors) matrix *A*.
 The second dimension of *a* must be at least $\max(1, m)$.
b(*ldb*,*) contains the upper quasi-triangular (for real flavors) or upper triangular (for complex flavors) matrix *B*. The second dimension of *b* must be at least $\max(1, n)$.
c(*ldc*,*) contains the right-hand-side of the first matrix equation in the generalized Sylvester equation (as defined by *trans*)
 The second dimension of *c* must be at least $\max(1, n)$.
d(*ldd*,*) contains the upper triangular matrix *D*.
 The second dimension of *d* must be at least $\max(1, m)$.
e(*lde*,*) contains the upper triangular matrix *E*.
 The second dimension of *e* must be at least $\max(1, n)$.

$f(ldf,*)$ contains the right-hand-side of the second matrix equation in the generalized Sylvester equation (as defined by *trans*)

The second dimension of f must be at least $\max(1, n)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda	INTEGER. The first dimension of a ; at least $\max(1, m)$.
ldb	INTEGER. The first dimension of b ; at least $\max(1, n)$.
ldc	INTEGER. The first dimension of c ; at least $\max(1, m)$.
ldd	INTEGER. The first dimension of d ; at least $\max(1, m)$.
lde	INTEGER. The first dimension of e ; at least $\max(1, n)$.
ldf	INTEGER. The first dimension of f ; at least $\max(1, m)$.
$lwork$	INTEGER.

The dimension of the array $work$. $lwork \geq 1$.

If $ijob = 1$ or 2 and $trans = 'N'$, $lwork \geq \max(1, 2*m*n)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#). See *Application Notes* for details.

$iwork$	INTEGER. Workspace array, DIMENSION at least $(m+n+6)$ for real flavors, and at least $(m+n+2)$ for complex flavors.

Output Parameters

c	If $ijob=0, 1$, or 2 , overwritten by the solution R . If $ijob=3$ or 4 and $trans = 'N'$, c holds R , the solution achieved during the computation of the Dif-estimate.
f	If $ijob=0, 1$, or 2 , overwritten by the solution L . If $ijob=3$ or 4 and $trans = 'N'$, f holds L , the solution achieved during the computation of the Dif-estimate.
dif	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors.

	On exit, <i>dif</i> is the reciprocal of a lower bound of the reciprocal of the Dif-function, that is, <i>dif</i> is an upper bound of $\text{Dif}[(A, D), (B, E)] = \sigma_{\min}(Z)$, where Z as in (2).
	If <i>ijob</i> = 0, or <i>trans</i> = 'T' (for real flavors), or <i>trans</i> = 'C' (for complex flavors), <i>dif</i> is not touched.
<i>scale</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. On exit, <i>scale</i> is the scaling factor in the generalized Sylvester equation. If $0 < \text{scale} < 1$, <i>c</i> and <i>f</i> hold the solutions <i>R</i> and <i>L</i> , respectively, to a slightly perturbed system but the input matrices <i>A</i> , <i>B</i> , <i>D</i> and <i>E</i> have not been changed. If <i>scale</i> = 0, <i>c</i> and <i>f</i> hold the solutions <i>R</i> and <i>L</i> , respectively, to the homogeneous system with $C = F = 0$. Normally, <i>scale</i> = 1.
<i>work</i> (1)	If <i>info</i> = 0, <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> > 0, (<i>A</i> , <i>D</i>) and (<i>B</i> , <i>E</i>) have common or close eigenvalues.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tgstyl` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>m</i> , <i>m</i>).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>n</i>).
<i>c</i>	Holds the matrix <i>C</i> of size (<i>m</i> , <i>n</i>).
<i>d</i>	Holds the matrix <i>D</i> of size (<i>m</i> , <i>m</i>).
<i>e</i>	Holds the matrix <i>E</i> of size (<i>n</i> , <i>n</i>).

<i>f</i>	Holds the matrix <i>F</i> of size (m,n) .
<i>ijob</i>	Must be 0, 1, 2, 3, or 4. The default value is 0.
<i>trans</i>	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?tgsna

Estimates reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a pair of matrices in generalized real Schur canonical form.

Syntax

FORTRAN 77:

```
call stgsna(job, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr, ldvr, s,
dif, mm, m, work, lwork, iwork, info)

call dtgsna(job, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr, ldvr, s,
dif, mm, m, work, lwork, iwork, info)

call ctgsna(job, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr, ldvr, s,
dif, mm, m, work, lwork, iwork, info)

call ztgsna(job, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr, ldvr, s,
dif, mm, m, work, lwork, iwork, info)
```

Fortran 95:

```
call tgsna(a, b [,s] [,dif] [,vl] [,vr] [,select] [,m] [,info])
```

Description

The real flavors `stgsna/dtgsna` of this routine estimate reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a matrix pair (A, B) in generalized real Schur canonical form (or of any matrix pair $(Q^*A^*Z^T, Q^*B^*Z^T)$ with orthogonal matrices Q and Z).

(A, B) must be in generalized real Schur form (as returned by `sgges/dgges`), that is, A is block upper triangular with 1-by-1 and 2-by-2 diagonal blocks. B is upper triangular.

The complex flavors `ctgsna/ztgsna` estimate reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a matrix pair (A, B) . (A, B) must be in generalized Schur canonical form, that is, A and B are both upper triangular.

Input Parameters

job CHARACTER*1. Specifies whether condition numbers are required for eigenvalues or eigenvectors. Must be 'E' or 'V' or 'B'.
 If *job* = 'E', for eigenvalues only (compute *s*).
 If *job* = 'V', for eigenvectors only (compute *dif*).
 If *job* = 'B', for both eigenvalues and eigenvectors (compute both *s* and *dif*).

howmny CHARACTER*1. Must be 'A' or 'S'.
 If *howmny* = 'A', compute condition numbers for all eigenpairs.
 If *howmny* = 'S', compute condition numbers for selected eigenpairs specified by the logical array *select*.

select LOGICAL.
 Array, DIMENSION at least $\max(1, n)$.
 If *howmny* = 'S', *select* specifies the eigenpairs for which condition numbers are required.
 If *howmny* = 'A', *select* is not referenced.
For real flavors:
 To select condition numbers for the eigenpair corresponding to a real eigenvalue $\omega(j)$, *select*(*j*) must be set to .TRUE.; to select condition numbers corresponding to a

complex conjugate pair of eigenvalues $\text{omega}(j)$ and $\text{omega}(j+1)$, either $\text{select}(j)$ or $\text{select}(j+1)$ must be set to `.TRUE.`

For complex flavors:
 To select condition numbers for the corresponding j -th eigenvalue and/or eigenvector, $\text{select}(j)$ must be set to `.TRUE.`

n INTEGER. The order of the square matrix pair (A, B) ($n \geq 0$).

$a, b, vl, vr, work$ REAL for stgsna
 DOUBLE PRECISION for dtgsna
 COMPLEX for ctgsna
 DOUBLE COMPLEX for ztgsna.

Arrays:
 $a(lda,*)$ contains the upper quasi-triangular (for real flavors) or upper triangular (for complex flavors) matrix A in the pair (A, B) .
 The second dimension of a must be at least $\max(1, n)$.
 $b(l db,*)$ contains the upper triangular matrix B in the pair (A, B) . The second dimension of b must be at least $\max(1, n)$.
 If $job = 'E'$ or $'B'$, $vl(ldvl,*)$ must contain left eigenvectors of (A, B) , corresponding to the eigenpairs specified by $howmny$ and $select$. The eigenvectors must be stored in consecutive columns of vl , as returned by `?tgevc`.
 If $job = 'V'$, vl is not referenced. The second dimension of vl must be at least $\max(1, m)$.
 If $job = 'E'$ or $'B'$, $vr(ldvr,*)$ must contain right eigenvectors of (A, B) , corresponding to the eigenpairs specified by $howmny$ and $select$. The eigenvectors must be stored in consecutive columns of vr , as returned by `?tgevc`.
 If $job = 'V'$, vr is not referenced. The second dimension of vr must be at least $\max(1, m)$.
 $work$ is a workspace array, its dimension $\max(1, lwork)$.
 If $job = 'E'$, $work$ is not referenced.

lda INTEGER. The first dimension of a ; at least $\max(1, n)$.

<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; at least $\max(1, n)$.
<i>ldvl</i>	INTEGER. The first dimension of <i>vl</i> ; $ldvl \geq 1$. If <i>job</i> = 'E' or 'B', then $ldvl \geq \max(1, n)$.
<i>ldvr</i>	INTEGER. The first dimension of <i>vr</i> ; $ldvr \geq 1$. If <i>job</i> = 'E' or 'B', then $ldvr \geq \max(1, n)$.
<i>mm</i>	INTEGER. The number of elements in the arrays <i>s</i> and <i>dif</i> ($mm \geq m$).
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . $lwork \geq \max(1, n)$. If <i>job</i> = 'V' or 'B', $lwork \geq 2*n*(n+2)+16$ for real flavors, and $lwork \geq \max(1, 2*n*n)$ for complex flavors. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla . See <i>Application Notes</i> for details.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $(n+6)$ for real flavors, and at least $(n+2)$ for complex flavors. If <i>ijob</i> = 'E', <i>iwork</i> is not referenced.

Output Parameters

<i>s</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Array, DIMENSION (<i>mm</i>). If <i>job</i> = 'E' or 'B', contains the reciprocal condition numbers of the selected eigenvalues, stored in consecutive elements of the array. If <i>job</i> = 'V', <i>s</i> is not referenced. <i>For real flavors:</i> For a complex conjugate pair of eigenvalues two consecutive elements of <i>s</i> are set to the same value. Thus, <i>s</i> (<i>j</i>), <i>dif</i> (<i>j</i>), and the <i>j</i> -th columns of <i>vl</i> and <i>vr</i> all correspond to the same eigenpair (but not in general the <i>j</i> -th eigenpair, unless all eigenpairs are selected).

<i>dif</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Array, DIMENSION (<i>mm</i>). If <i>job</i> = 'V' or 'B', contains the estimated reciprocal condition numbers of the selected eigenvectors, stored in consecutive elements of the array. If the eigenvalues cannot be reordered to compute <i>dif</i>(<i>j</i>), <i>dif</i>(<i>j</i>) is set to 0; this can only occur when the true value would be very small anyway. If <i>job</i> = 'E', <i>dif</i> is not referenced. For real flavors: For a complex eigenvector, two consecutive elements of <i>dif</i> are set to the same value. For complex flavors: For each eigenvalue/vector specified by <i>select</i>, <i>dif</i> stores a Frobenius norm-based estimate of Difl.
<i>m</i>	INTEGER. The number of elements in the arrays <i>s</i> and <i>dif</i> used to store the specified condition numbers; for each selected eigenvalue one element is used. If <i>howmny</i> = 'A', <i>m</i> is set to <i>n</i>.
<i>work</i> (1)	<i>work</i>(1) If <i>job</i> is not 'E' and <i>info</i> = 0, on exit, <i>work</i>(1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tgсна` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>n</i>).
<i>s</i>	Holds the vector of length (<i>mm</i>).

<i>dif</i>	Holds the vector of length (<i>mm</i>).
<i>vl</i>	Holds the matrix <i>VL</i> of size (<i>n</i> , <i>mm</i>).
<i>vr</i>	Holds the matrix <i>VR</i> of size (<i>n</i> , <i>mm</i>).
<i>select</i>	Holds the vector of length (<i>n</i>).
<i>howmny</i>	Restored based on the presence of the argument <i>select</i> as follows: <i>howmny</i> = 'S', if <i>select</i> is present, <i>howmny</i> = 'A', if <i>select</i> is omitted.
<i>job</i>	Restored based on the presence of arguments <i>s</i> and <i>dif</i> as follows: <i>job</i> = 'B', if both <i>s</i> and <i>dif</i> are present, <i>job</i> = 'E', if <i>s</i> is present and <i>dif</i> omitted, <i>job</i> = 'V', if <i>s</i> is omitted and <i>dif</i> present, Note that there will be an error condition if both <i>s</i> and <i>dif</i> are omitted.

Application Notes

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Generalized Singular Value Decomposition

This section describes LAPACK computational routines used for finding the generalized singular value decomposition (GSVD) of two matrices *A* and *B* as

$$U^H A Q = D_1 * (0 \ R),$$

$$V^H B Q = D_2 * (0 \ R),$$

where *U*, *V*, and *Q* are orthogonal/unitary matrices, *R* is a nonsingular upper triangular matrix, and *D*₁, *D*₂ are "diagonal" matrices of the structure detailed in the routines description section.

Table 4-7 lists LAPACK routines (Fortran 77 interface) that perform generalized singular value decomposition of matrices. Respective routine names in Fortran 95 interface are without the first symbol (see [Routine Naming Conventions](#)).

Table 4-7 Computational Routines for Generalized Singular Value Decomposition

Routine name	Operation performed
?ggsvp	Computes the preprocessing decomposition for the generalized SVD
?tgsja	Computes the generalized SVD of two upper triangular or trapezoidal matrices

You can use routines listed in the above table as well as the driver routine [?ggsvd](#) to find the GSVD of a pair of general rectangular matrices.

[?ggsvp](#)

Computes the preprocessing decomposition for the generalized SVD.

Syntax

FORTRAN 77:

```
call sggsvp(jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb, k, l, u,
ldu, v, ldv, q, ldq, iwork, tau, work, info)

call dggsvp(jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb, k, l, u,
ldu, v, ldv, q, ldq, iwork, tau, work, info)

call cggsvp (jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb, k, l, u,
ldu, v, ldv, q, ldq, iwork, rwork, tau, work, info)

call zggsvp(jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb, k, l, u,
ldu, v, ldv, q, ldq, iwork, rwork, tau, work, info)
```

Fortran 95:

```
call ggsvp(a, b, tola, tolb [, k] [,l] [,u] [,v] [,q] [,info])
```

Description

This routine computes orthogonal matrices U , V and Q such that

$$U^H A Q = \begin{matrix} & n-k-l & k & l \\ & k & & \\ & l & & \\ m-k-l & & & \end{matrix} \begin{pmatrix} 0 & A_{12} & A_{13} \\ 0 & 0 & A_{23} \\ 0 & 0 & 0 \end{pmatrix}, \quad \text{if } m-k-l \geq 0$$

$$= \begin{matrix} & n-k-l & k & l \\ & k & & \\ m-k & & & \end{matrix} \begin{pmatrix} 0 & A_{12} & A_{13} \\ 0 & 0 & A_{23} \end{pmatrix}, \quad \text{if } m-k-l < 0$$

$$V^H B Q = \begin{matrix} & n-k-l & k & l \\ & l & & \\ p-l & & & \end{matrix} \begin{pmatrix} 0 & 0 & B_{13} \\ 0 & 0 & 0 \end{pmatrix}$$

where the k -by- k matrix A_{12} and l -by- l matrix B_{13} are nonsingular upper triangular; A_{23} is l -by- l upper triangular if $m-k-l \geq 0$, otherwise A_{23} is $(m-k)$ -by- l upper trapezoidal. The sum $k+l$ is equal to the effective numerical rank of the $(m+p)$ -by- n matrix $(A^H, B^H)^H$.

This decomposition is the preprocessing step for computing the Generalized Singular Value Decomposition (GSVD), see subroutine [?ggsvd](#).

Input Parameters

jobu CHARACTER*1. Must be 'U' or 'N'.
 If *jobu* = 'U', orthogonal/unitary matrix U is computed.
 If *jobu* = 'N', U is not computed.

jobv CHARACTER*1. Must be 'V' or 'N'.
 If *jobv* = 'V', orthogonal/unitary matrix V is computed.

<i>jobv</i>	If <i>jobv</i> = 'N', <i>v</i> is not computed. CHARACTER*1. Must be 'Q' or 'N'. If <i>jobq</i> = 'Q', orthogonal/unitary matrix <i>q</i> is computed. If <i>jobq</i> = 'N', <i>q</i> is not computed.
<i>m</i>	INTEGER. The number of rows of the matrix <i>A</i> ($m \geq 0$).
<i>p</i>	INTEGER. The number of rows of the matrix <i>B</i> ($p \geq 0$).
<i>n</i>	INTEGER. The number of columns of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>a, b, tau, work</i>	REAL for sggsvp DOUBLE PRECISION for dggsvp COMPLEX for cggsvp DOUBLE COMPLEX for zggsvp. Arrays: <i>a</i>(<i>lda</i>,*) contains the <i>m</i>-by-<i>n</i> matrix <i>A</i>. The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>b</i>(<i>ldb</i>,*) contains the <i>p</i>-by-<i>n</i> matrix <i>B</i>. The second dimension of <i>b</i> must be at least $\max(1, n)$. <i>tau</i>(*) is a workspace array. The dimension of <i>tau</i> must be at least $\max(1, n)$. <i>work</i>(*) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, 3n, m, p)$.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; at least $\max(1, m)$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; at least $\max(1, p)$.
<i>tola, tolb</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. <i>tola</i> and <i>tolb</i> are the thresholds to determine the effective numerical rank of matrix <i>B</i> and a subblock of <i>A</i>. Generally, they are set to $tola = \max(m, n) * A * \text{MACHEPS}$, $tolb = \max(p, n) * B * \text{MACHEPS}$. The size of <i>tola</i> and <i>tolb</i> may affect the size of backward errors of the decomposition.
<i>ldu</i>	INTEGER. The first dimension of the output array <i>u</i>. <i>ldu</i> $\geq \max(1, m)$ if <i>jobu</i> = 'U'; <i>ldu</i> ≥ 1 otherwise.

<i>ldv</i>	INTEGER. The first dimension of the output array <i>v</i> . <i>ldv</i> $\geq \max(1, p)$ if <i>jobv</i> = 'V'; <i>ldv</i> ≥ 1 otherwise.
<i>ldq</i>	INTEGER. The first dimension of the output array <i>q</i> . <i>ldq</i> $\geq \max(1, n)$ if <i>jobq</i> = 'Q'; <i>ldq</i> ≥ 1 otherwise.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.
<i>rwork</i>	REAL for cggsvp DOUBLE PRECISION for zggsvp. Workspace array, DIMENSION at least $\max(1, 2n)$. Used in complex flavors only.

Output Parameters

<i>a</i>	Overwritten by the triangular (or trapezoidal) matrix described in the <i>Description</i> section.
<i>b</i>	Overwritten by the triangular matrix described in the <i>Description</i> section.
<i>k, l</i>	INTEGER. On exit, <i>k</i> and <i>l</i> specify the dimension of subblocks. The sum <i>k</i> + <i>l</i> is equal to effective numerical rank of $(A^H, B^H)^H$.
<i>u, v, q</i>	REAL for sggsvp DOUBLE PRECISION for dggsvp COMPLEX for cggsvp DOUBLE COMPLEX for zggsvp. Arrays: If <i>jobu</i> = 'U', <i>u</i> (<i>ldu</i> ,*) contains the orthogonal/unitary matrix <i>U</i> . The second dimension of <i>u</i> must be at least $\max(1, m)$. If <i>jobu</i> = 'N', <i>u</i> is not referenced. If <i>jobv</i> = 'V', <i>v</i> (<i>ldv</i> ,*) contains the orthogonal/unitary matrix <i>V</i> . The second dimension of <i>v</i> must be at least $\max(1, m)$. If <i>jobv</i> = 'N', <i>v</i> is not referenced. If <i>jobq</i> = 'Q', <i>q</i> (<i>ldq</i> ,*) contains the orthogonal/unitary matrix <i>Q</i> . The second dimension of <i>q</i> must be at least $\max(1, n)$. If <i>jobq</i> = 'N', <i>q</i> is not referenced.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *ggsvp* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (m, n) .
<i>b</i>	Holds the matrix <i>B</i> of size (p, n) .
<i>u</i>	Holds the matrix <i>U</i> of size (m, m) .
<i>v</i>	Holds the matrix <i>V</i> of size (p, m) .
<i>q</i>	Holds the matrix <i>Q</i> of size (n, n) .
<i>jobu</i>	Restored based on the presence of the argument <i>u</i> as follows: <i>jobu</i> = 'U', if <i>u</i> is present, <i>jobu</i> = 'N', if <i>u</i> is omitted.
<i>jobv</i>	Restored based on the presence of the argument <i>v</i> as follows: <i>jobz</i> = 'V', if <i>v</i> is present, <i>jobz</i> = 'N', if <i>v</i> is omitted.
<i>jobq</i>	Restored based on the presence of the argument <i>q</i> as follows: <i>jobz</i> = 'Q', if <i>q</i> is present, <i>jobz</i> = 'N', if <i>q</i> is omitted.

?tgsja

Computes the generalized SVD of two upper triangular or trapezoidal matrices.

Syntax

FORTRAN 77:

```
call stgsja(jobu, jobv, jobq, m, p, n, k, l, a, lda, b, ldb, tola, tolb, alpha,
beta, u, ldu, v, ldv, q, ldq, work, ncycle, info)

call dtgsja(jobu, jobv, jobq, m, p, n, k, l, a, lda, b, ldb, tola, tolb, alpha,
beta, u, ldu, v, ldv, q, ldq, work, ncycle, info)

call ctgsja(jobu, jobv, jobq, m, p, n, k, l, a, lda, b, ldb, tola, tolb, alpha,
beta, u, ldu, v, ldv, q, ldq, work, ncycle, info)

call ztgsja(jobu, jobv, jobq, m, p, n, k, l, a, lda, b, ldb, tola, tolb, alpha,
beta, u, ldu, v, ldv, q, ldq, work, ncycle, info)
```

Fortran 95:

```
call tgsja(a, b, tola, tolb, k, l [,u] [,v] [,q] [,jobu] [,jobv] [,jobq]
[,alpha] [,beta] [,ncycle] [,info])
```

Description

This routine computes the generalized singular value decomposition (GSVD) of two real/complex upper triangular (or trapezoidal) matrices A and B . On entry, it is assumed that matrices A and B have the following forms, which may be obtained by the preprocessing subroutine [?ggsvp](#) from a general m -by- n matrix A and p -by- n matrix B :

$$A = \begin{matrix} & n-k-l & k & l \\ & k & & \\ & l & & \\ m-k-l & & & \end{matrix} \begin{pmatrix} 0 & A_{12} & A_{13} \\ 0 & 0 & A_{23} \\ 0 & 0 & 0 \end{pmatrix}, \quad \text{if } m-k-l \geq 0$$

$$= \begin{matrix} n-k-1 & k & 1 \\ m-k \end{matrix} \begin{pmatrix} 0 & A_{12} & A_{13} \\ 0 & 0 & A_{23} \end{pmatrix}, \quad \text{if } m-k-1 < 0$$

$$B = \begin{matrix} n-k-1 & k & 1 \\ p-1 \end{matrix} \begin{pmatrix} 0 & 0 & B_{13} \\ 0 & 0 & 0 \end{pmatrix}$$

where the k -by- k matrix A_{12} and 1 -by- 1 matrix B_{13} are nonsingular upper triangular; A_{23} is 1 -by- 1 upper triangular if $m-k-1 \geq 0$, otherwise A_{23} is $(m-k)$ -by- 1 upper trapezoidal.

On exit,

$$U^H A Q = D_1^* (0 \ R), \quad V^H B Q = D_2^* (0 \ R),$$

where U , V and Q are orthogonal/unitary matrices, R is a nonsingular upper triangular matrix, and D_1 and D_2 are "diagonal" matrices, which are of the following structures:

If $m-k-1 \geq 0$,

$$D_1 = \begin{matrix} & k & 1 \\ & k & 1 \\ m-k-1 \end{matrix} \begin{pmatrix} 0 & 0 \\ 0 & C \\ 0 & 0 \end{pmatrix}$$

$$D_1 = \begin{matrix} & k & l \\ & l & \\ p-l & \begin{pmatrix} 0 & S \\ 0 & 0 \end{pmatrix} \end{matrix}$$

$$(0 \ R) = \begin{matrix} & n-k-l & k & l \\ k & \begin{pmatrix} 0 & R_{11} & R_{12} \\ 0 & 0 & R_{22} \end{pmatrix} \\ l & \end{matrix}$$

where

$C = \text{diag}(\alpha(k+1), \dots, \alpha(k+l))$

$S = \text{diag}(\beta(k+1), \dots, \beta(k+l))$

$C^2 + S^2 = I$

R is stored in $a(1:k+l, n-k-l+1:n)$ on exit.

If $m-k-l < 0$,

$$D_1 = \begin{matrix} & k & m-l & k+l-m \\ & k & \\ m-k & \begin{pmatrix} I & 0 & 0 \\ 0 & C & 0 \end{pmatrix} \end{matrix}$$

$$D_2 = \begin{matrix} & k & m-k & k+l-m \\ m-k & \begin{pmatrix} 0 & S & 0 \end{pmatrix} \\ k+1-m & \begin{pmatrix} 0 & 0 & I \end{pmatrix} \\ p-l & \begin{pmatrix} 0 & 0 & 0 \end{pmatrix} \end{matrix}$$

$$(0 \ R) = \begin{matrix} & n-k-l & k & m-k & k+l-m \\ k & \begin{pmatrix} 0 & R_{11} & R_{12} & R_{13} \end{pmatrix} \\ m-k & \begin{pmatrix} 0 & 0 & R_{22} & R_{23} \end{pmatrix} \\ k+l-m & \begin{pmatrix} 0 & 0 & 0 & R_{33} \end{pmatrix} \end{matrix}$$

where

$C = \text{diag}(\alpha(K+1), \dots, \alpha(m)),$

$S = \text{diag}(\beta(K+1), \dots, \beta(m)),$

$C^2 + S^2 = I$

$$\begin{pmatrix} R_{11} & R_{12} & R_{13} \\ 0 & R_{22} & R_{23} \end{pmatrix}$$

On exit, $\begin{pmatrix} R_{11} & R_{12} & R_{13} \\ 0 & R_{22} & R_{23} \end{pmatrix}$ is stored in $a(1:m, n-k-l+1:n)$ and R_{33} is stored

in $b(m-k+1:l, n+m-k-l+1:n)$.

The computation of the orthogonal/unitary transformation matrices U , V or Q is optional. These matrices may either be formed explicitly, or they *may* be postmultiplied into input matrices U_1 , V_1 , or Q_1 .

Input Parameters

jobu CHARACTER*1. Must be 'U', 'I', or 'N'.

If $jobu = 'U'$, u must contain an orthogonal/unitary matrix U_1 on entry.
 If $jobu = 'I'$, u is initialized to the unit matrix.
 If $jobu = 'N'$, u is not computed.

$jobv$ CHARACTER*1. Must be 'V', 'I', or 'N'.
 If $jobv = 'V'$, v must contain an orthogonal/unitary matrix V_1 on entry.
 If $jobv = 'I'$, v is initialized to the unit matrix.
 If $jobv = 'N'$, v is not computed.

$jobq$ CHARACTER*1. Must be 'Q', 'I', or 'N'.
 If $jobq = 'Q'$, q must contain an orthogonal/unitary matrix Q_1 on entry.
 If $jobq = 'I'$, q is initialized to the unit matrix.
 If $jobq = 'N'$, q is not computed.

m INTEGER. The number of rows of the matrix A ($m \geq 0$).

p INTEGER. The number of rows of the matrix B ($p \geq 0$).

n INTEGER. The number of columns of the matrices A and B ($n \geq 0$).

k, l INTEGER. Specify the subblocks in the input matrices A and B , whose GSVD is computed.

$a, b, u, v, q, work$ REAL for stgsja
 DOUBLE PRECISION for dtgsja
 COMPLEX for ctgsja
 DOUBLE COMPLEX for ztgsja.

Arrays:
 $a(lda,*)$ contains the m -by- n matrix A .
 The second dimension of a must be at least $\max(1, n)$.
 $b(l db,*)$ contains the p -by- n matrix B .
 The second dimension of b must be at least $\max(1, n)$.
 If $jobu = 'U'$, $u(ldu,*)$ must contain a matrix U_1 (usually the orthogonal/unitary matrix returned by ?ggsvp).
 The second dimension of u must be at least $\max(1, m)$.
 If $jobv = 'V'$, $v(ldv,*)$ must contain a matrix V_1 (usually the orthogonal/unitary matrix returned by ?ggsvp).
 The second dimension of v must be at least $\max(1, p)$.

If $jobq = 'Q'$, $q(ldq,*)$ must contain a matrix Q_1 (usually the orthogonal/unitary matrix returned by ?ggsvp).
The second dimension of q must be at least $\max(1, n)$.
 $work(*)$ is a workspace array.
The dimension of $work$ must be at least $\max(1, 2n)$.

lda INTEGER. The first dimension of a ; at least $\max(1, m)$.

ldb INTEGER. The first dimension of b ; at least $\max(1, p)$.

ldu INTEGER. The first dimension of the array u .
 $ldu \geq \max(1, m)$ if $jobu = 'U'$; $ldu \geq 1$ otherwise.

ldv INTEGER. The first dimension of the array v .
 $ldv \geq \max(1, p)$ if $jobv = 'V'$; $ldv \geq 1$ otherwise.

ldq INTEGER. The first dimension of the array q .
 $ldq \geq \max(1, n)$ if $jobq = 'Q'$; $ldq \geq 1$ otherwise.

*tola, tol**b* REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
tola and *tolb* are the convergence criteria for the Jacobi-Kogbetliantz iteration procedure. Generally, they are the same as used in ?ggsvp:
 $tola = \max(m, n) * |A| * \text{MACHEPS}$,
 $tolb = \max(p, n) * |B| * \text{MACHEPS}$.

Output Parameters

a On exit, $a(n-k+1:n, 1:\min(k+1, m))$ contains the triangular matrix R or part of R .

b On exit, if necessary, $b(m-k+1: l, n+m-k-l+1: n)$ contains a part of R .

alpha, beta REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
Arrays, DIMENSION at least $\max(1, n)$. Contain the generalized singular value pairs of A and B :
 $alpha(1:k) = 1$,
 $beta(1:k) = 0$,
and if $m-k-l \geq 0$,
 $alpha(k+1:k+l) = \text{diag}(C)$,

```

      beta(k+1:k+1) = diag(S),
      or if  $m-k-1 < 0$ ,
      alpha(k+1:m) = C, alpha(m+1:k+1)=0
      beta(K+1:M) = S,
      beta(m+1:k+1) = 1.
      Furthermore, if  $k+1 < n$ ,
      alpha(k+1+1:n) = 0 and
      beta(k+1+1:n) = 0.

u      If jobu = 'I', u contains the orthogonal/unitary matrix U.
      If jobu = 'U', u contains the product  $U_1 * U$ .
      If jobu = 'N', u is not referenced.

v      If jobv = 'I', v contains the orthogonal/unitary matrix U.
      If jobv = 'V', v contains the product  $V_1 * V$ .
      If jobv = 'N', v is not referenced.

q      If jobq = 'I', q contains the orthogonal/unitary matrix U.
      If jobq = 'Q', q contains the product  $Q_1 * Q$ .
      If jobq = 'N', q is not referenced.

ncycle INTEGER. The number of cycles required for convergence.
info    INTEGER.
      If info = 0, the execution is successful.
      If info = -i, the i-th parameter had an illegal value.
      If info = 1, the procedure does not converge after MAXIT
      cycles.

```

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tgsgja` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (m, n) .
<i>b</i>	Holds the matrix <i>B</i> of size (p, n) .
<i>u</i>	Holds the matrix <i>U</i> of size (m, m) .
<i>v</i>	Holds the matrix <i>V</i> of size (p, p) .
<i>q</i>	Holds the matrix <i>Q</i> of size (n, n) .
<i>alpha</i>	Holds the vector of length (n) .

<i>beta</i>	Holds the vector of length (<i>n</i>).
<i>jobu</i>	If omitted, this argument is restored based on the presence of argument <i>u</i> as follows: <i>jobu</i> = 'U', if <i>u</i> is present, <i>jobu</i> = 'N', if <i>u</i> is omitted. If present, <i>jobu</i> must be equal to 'I' or 'U' and the argument <i>u</i> must also be present. Note that there will be an error condition if <i>jobu</i> is present and <i>u</i> omitted.
<i>jobv</i>	If omitted, this argument is restored based on the presence of argument <i>v</i> as follows: <i>jobv</i> = 'V', if <i>v</i> is present, <i>jobv</i> = 'N', if <i>v</i> is omitted. If present, <i>jobv</i> must be equal to 'I' or 'V' and the argument <i>v</i> must also be present. Note that there will be an error condition if <i>jobv</i> is present and <i>v</i> omitted.
<i>jobq</i>	If omitted, this argument is restored based on the presence of argument <i>q</i> as follows: <i>jobq</i> = 'Q', if <i>q</i> is present, <i>jobq</i> = 'N', if <i>q</i> is omitted. If present, <i>jobq</i> must be equal to 'I' or 'Q' and the argument <i>q</i> must also be present. Note that there will be an error condition if <i>jobq</i> is present and <i>q</i> omitted.

Driver Routines

Each of the LAPACK driver routines solves a complete problem. To arrive at the solution, driver routines typically call a sequence of appropriate [computational routines](#).

Driver routines are described in the following sections :

[Linear Least Squares \(LLS\) Problems](#)

[Generalized LLS Problems](#)

[Symmetric Eigenproblems](#)

[Nonsymmetric Eigenproblems](#)

- Singular Value Decomposition
- Generalized Symmetric Definite Eigenproblems
- Generalized Nonsymmetric Eigenproblems

Linear Least Squares (LLS) Problems

This section describes LAPACK driver routines used for solving linear least squares problems. Table 4-8 lists all such routines for Fortran 77 interface. Respective routine names in Fortran 95 interface are without the first symbol (see [Routine Naming Conventions](#)).

Table 4-8 Driver Routines for Solving LLS Problems

Routine Name	Operation performed
?gels	Uses QR or LQ factorization to solve a overdetermined or underdetermined linear system with full rank matrix.
?gelsy	Computes the minimum-norm solution to a linear least squares problem using a complete orthogonal factorization of A.
?gelss	Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A.
?gelsd	Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A and a divide and conquer method.

?gels

Uses QR or LQ factorization to solve a overdetermined or underdetermined linear system with full rank matrix.

Syntax

FORTRAN 77:

```
call sgels(trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info)
call dgels(trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info)
call cgels(trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info)
call zgels(trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info)
```

Fortran 95:

```
call gels(a, b [,trans] [,info])
```

Description

This routine solves overdetermined or underdetermined real/ complex linear systems involving an m -by- n matrix A , or its transpose/ conjugate-transpose, using a QR or LQ factorization of A . It is assumed that A has full rank.

The following options are provided:

1. If $trans = 'N'$ and $m \geq n$: find the least squares solution of an overdetermined system, that is, solve the least squares problem

```
minimize ||b - A*x||2
```

2. If $trans = 'N'$ and $m < n$: find the minimum norm solution of an underdetermined system $A*X = B$.

3. If $trans = 'T'$ or $'C'$ and $m \geq n$: find the minimum norm solution of an undetermined system $A_H^*X = B$.

4. If $trans = 'T'$ or $'C'$ and $m < n$: find the least squares solution of an overdetermined system, that is, solve the least squares problem

```
minimize ||b - AH*x||2
```

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the m -by- $nrhs$ right hand side matrix B and the n -by- nrh solution matrix X .

Input Parameters

<i>trans</i>	CHARACTER*1. Must be 'N', 'T', or 'C'. If $trans = 'N'$, the linear system involves matrix A ; If $trans = 'T'$, the linear system involves the transposed matrix A^T (for real flavors only); If $trans = 'C'$, the linear system involves the conjugate-transposed matrix A^H (for complex flavors only).
<i>m</i>	INTEGER. The number of rows of the matrix A ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns of the matrix A ($n \geq 0$).

<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns in <i>B</i> ($nrhs \geq 0$).
<i>a</i> , <i>b</i> , <i>work</i>	REAL for <i>sgels</i> DOUBLE PRECISION for <i>dgels</i> COMPLEX for <i>cgels</i> DOUBLE COMPLEX for <i>zgels</i> . Arrays: <i>a</i> (<i>lda</i> ,*) contains the <i>m</i> -by- <i>n</i> matrix <i>A</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>b</i> (<i>ldb</i> ,*) contains the matrix <i>B</i> of right hand side vectors, stored columnwise; <i>B</i> is <i>m</i> -by- <i>nrhs</i> if <i>trans</i> = 'N' , or <i>n</i> -by- <i>nrhs</i> if <i>trans</i> = 'T' or 'C'. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; at least $\max(1, m)$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; must be at least $\max(1, m, n)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array; must be at least $\min(m, n) + \max(1, m, n, nrhs)$. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla . See <i>Application Notes</i> for the suggested value of <i>lwork</i> .

Output Parameters

<i>a</i>	On exit, overwritten by the factorization data as follows: if $m \geq n$, array <i>a</i> contains the details of the <i>QR</i> factorization of the matrix <i>A</i> as returned by ?geqrf ; if $m < n$, array <i>a</i> contains the details of the <i>LQ</i> factorization of the matrix <i>A</i> as returned by ?gelqf .
<i>b</i>	If <i>info</i> = 0, <i>b</i> overwritten by the solution vectors, stored columnwise:

if *trans* = 'N' and $m \geq n$, rows 1 to n of *b* contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of modulus of elements $n+1$ to m in that column;
 if *trans* = 'N' and $m < n$, rows 1 to n of *b* contain the minimum norm solution vectors;
 if *trans* = 'T' or 'C' and $m \geq n$, rows 1 to m of *b* contain the minimum norm solution vectors; if *trans* = 'T' or 'C' and $m < n$, rows 1 to m of *b* contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of modulus of elements $m+1$ to n in that column.

work(1) If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.
 If *info* = *i*, the *i*-th diagonal element of the triangular factor of *A* is zero, so that *A* does not have full rank; the least squares solution could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *gels* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (m, n) .
<i>b</i>	Holds the matrix of size $\max(m, n)$ -by- <i>nrhs</i> . If <i>trans</i> = 'N', then, on entry, the size of <i>b</i> is m -by- <i>nrhs</i> , If <i>trans</i> = 'T', then, on entry, the size of <i>b</i> is n -by- <i>nrhs</i> ,
<i>trans</i>	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using $lwork = \min(m, n) + \max(1, m, n, nrhs) * blocksize$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?gelsy

Computes the minimum-norm solution to a linear least squares problem using a complete orthogonal factorization of A.

Syntax

FORTRAN 77:

```
call sgelsy(m, n, nrhs, a, lda, b, ldb, jpvt, rcond, rank, work, lwork, info)
call dgelsy(m, n, nrhs, a, lda, b, ldb, jpvt, rcond, rank, work, lwork, info)
call cgelsy(m, n, nrhs, a, lda, b, ldb, jpvt, rcond, rank, work, lwork, rwork,
info)
call zgelsy(m, n, nrhs, a, lda, b, ldb, jpvt, rcond, rank, work, lwork, rwork,
info)
```

Fortran 95:

```
call gelsy(a, b [,rank] [,jpvt] [,rcond] [,info])
```

Description

This routine computes the minimum-norm solution to a real/complex linear least squares problem:

$$\text{minimize } ||b - A*x||_2$$

using a complete orthogonal factorization of A . A is an m -by- n matrix which may be rank-deficient. Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the m -by- $nrhs$ right hand side matrix B and the n -by- $nrhs$ solution matrix X .

The routine first computes a QR factorization with column pivoting:

$$AP = Q \begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{pmatrix}$$

with R_{11} defined as the largest leading submatrix whose estimated condition number is less than $1/rcond$. The order of R_{11} , $rank$, is the effective rank of A . Then, R_{22} is considered to be negligible, and R_{12} is annihilated by orthogonal/unitary transformations from the right, arriving at the complete orthogonal factorization:

$$AP = Q \begin{pmatrix} T_{11} & 0 \\ 0 & 0 \end{pmatrix} Z$$

The minimum-norm solution is then

$$X = PZ^H \begin{pmatrix} T_{11}^{-1} & Q_1^H b \\ 0 & 0 \end{pmatrix}$$

where Q_1 consists of the first $rank$ columns of Q . This routine is basically identical to the original `qelsx` except three differences:

- The call to the subroutine `?geqpf` has been substituted by the call to the subroutine `?geqp3`. This subroutine is a BLAS-3 version of the QR factorization with column pivoting.
- Matrix B (the right hand side) is updated with BLAS-3.
- The permutation of matrix B (the right hand side) is faster and more simple.

Input Parameters

m	INTEGER. The number of rows of the matrix A ($m \geq 0$).
n	INTEGER. The number of columns of the matrix A ($n \geq 0$).
$nrhs$	INTEGER. The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).
$a, b, work$	REAL for <code>sgelsy</code> DOUBLE PRECISION for <code>dgelsy</code> COMPLEX for <code>cgelsy</code> DOUBLE COMPLEX for <code>zgelsy</code> . Arrays: $a(lda,*)$ contains the m -by- n matrix A . The second dimension of a must be at least $\max(1, n)$. $b(l db,*)$ contains the m -by- $nrhs$ right hand side matrix B . The second dimension of b must be at least $\max(1, nrhs)$. $work$ is a workspace array, its dimension $\max(1, lwork)$.
lda	INTEGER. The first dimension of a ; at least $\max(1, m)$.
ldb	INTEGER. The first dimension of b ; must be at least $\max(1, m, n)$.
$jpvt$	INTEGER. Array, DIMENSION at least $\max(1, n)$. On entry, if $jpvt(i) \neq 0$, the i -th column of A is permuted to the front of AP , otherwise the i -th column of A is a free column.
$rcond$	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors.

rcond is used to determine the effective rank of A , which is defined as the order of the largest leading triangular submatrix R_{11} in the QR factorization with pivoting of A , whose estimated condition number $< 1/rcond$.

lwork

INTEGER. The size of the *work* array.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#). See *Application Notes* for the suggested value of *lwork*.

rwork

REAL for [cgelsy](#) DOUBLE PRECISION for [zgelsy](#).

Workspace array, DIMENSION at least $\max(1, 2n)$. Used in complex flavors only.

Output Parameters

a

On exit, overwritten by the details of the complete orthogonal factorization of A .

b

Overwritten by the n -by-*nrhs* solution matrix X .

jpvt

On exit, if *jpvt*(*i*) = *k*, then the *i*-th column of AP was the *k*-th column of A .

rank

INTEGER. The effective rank of A , that is, the order of the submatrix R_{11} . This is the same as the order of the submatrix T_{11} in the complete orthogonal factorization of A .

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine [gelsy](#) interface are the following:

a

Holds the matrix A of size (m, n) .

<i>b</i>	Holds the matrix of size $\max(m, n)$ -by- <i>nrhs</i> . On entry, contains the <i>m</i> -by- <i>nrhs</i> right hand side matrix <i>B</i> , On exit, overwritten by the <i>n</i> -by- <i>nrhs</i> solution matrix <i>X</i> .
<i>jpvt</i>	Holds the vector of length (<i>n</i>). Default value for this element is <i>jpvt</i> (<i>i</i>) = 0.
<i>rcond</i>	Default value for this element is <i>rcond</i> = 100*EPSILON(1.0_WP).

Application Notes

For real flavors:

The unblocked strategy requires that:

$$lwork \geq \max(mn+3n+1, 2*mn + nrhs),$$

where $mn = \min(m, n)$.

The block algorithm requires that:

$$lwork \geq \max(mn+2n+nb*(n+1), 2*mn+nb*nrhs),$$

where *nb* is an upper bound on the blocksize returned by [ilaenv](#) for the routines [sgeqp3/dgeqp3](#), [stzrzf/dtzrzf](#), [stzrqf/dtzrqf](#), [sormqr/dormqr](#), and [sormrz/dormrz](#).

For complex flavors:

The unblocked strategy requires that:

$$lwork \geq mn + \max(2*mn, n+1, mn + nrhs),$$

where $mn = \min(m, n)$.

The block algorithm requires that:

$$lwork < mn + \max(2*mn, nb*(n+1), mn+mn*nb, mn+ nb*nrhs),$$

where *nb* is an upper bound on the blocksize returned by [ilaenv](#) for the routines [cgeqp3/zgeqp3](#), [ctzrzf/ztzrzf](#), [ctzrqf/ztzrqf](#), [cunmqr/zunmqr](#), and [cunmrz/zunmrz](#).

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?gelss

Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A .

Syntax

FORTRAN 77:

```
call sgelss(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, info)
call dgelss(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, info)
call cgelss(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, rwork,
info)
call zgelss(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, rwork,
info)
```

Fortran 95:

```
call gelss(a, b [,rank] [,s] [,rcond] [,info])
```

Description

This routine computes the minimum norm solution to a real linear least squares problem:

minimize $\|b - A \cdot x\|_2$

using the singular value decomposition (SVD) of A . A is an m -by- n matrix which may be rank-deficient. Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the m -by- $nrhs$ right hand side matrix B and the n -by- $nrhs$ solution matrix X . The effective rank of A is determined by treating as zero those singular values which are less than `rcond` times the largest singular value.

Input Parameters

<i>m</i>	INTEGER. The number of rows of the matrix <i>A</i> ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns of the matrix <i>A</i> ($n \geq 0$).
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns in <i>B</i> ($nrhs \geq 0$).
<i>a</i> , <i>b</i> , <i>work</i>	REAL for <i>sgelss</i> DOUBLE PRECISION for <i>dgelss</i> COMPLEX for <i>cgelss</i> DOUBLE COMPLEX for <i>zgelss</i> . Arrays: <i>a</i> (<i>lda</i> ,*) contains the <i>m</i> -by- <i>n</i> matrix <i>A</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>b</i> (<i>ldb</i> ,*) contains the <i>m</i> -by- <i>nrhs</i> right hand side matrix <i>B</i> . The second dimension of <i>b</i> must be at least $\max(1, nrhs)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; at least $\max(1, m)$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; must be at least $\max(1, m, n)$.
<i>rcond</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. <i>rcond</i> is used to determine the effective rank of <i>A</i> . Singular values $s(i) \leq rcond * s(1)$ are treated as zero. If <i>rcond</i> < 0, machine precision is used instead.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array; $lwork \geq 1$. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <i>xerbla</i> . See <i>Application Notes</i> for the suggested value of <i>lwork</i> .
<i>rwork</i>	REAL for <i>cgelss</i> DOUBLE PRECISION for <i>zgelss</i> .

Workspace array used in complex flavors only. `DIMENSION` at least $\max(1, 5 * \min(m, n))$.

Output Parameters

<i>a</i>	On exit, the first $\min(m, n)$ rows of <i>A</i> are overwritten with its right singular vectors, stored row-wise.
<i>b</i>	Overwritten by the <i>n</i> -by- <i>nrhs</i> solution matrix <i>X</i> . If $m \geq n$ and <i>rank</i> = <i>n</i> , the residual sum-of-squares for the solution in the <i>i</i> -th column is given by the sum of squares of modulus of elements <i>n</i> +1: <i>m</i> in that column.
<i>s</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, <code>DIMENSION</code> at least $\max(1, \min(m, n))$. The singular values of <i>A</i> in decreasing order. The condition number of <i>A</i> in the 2-norm is $k_2(A) = s(1) / s(\min(m, n))$.
<i>rank</i>	INTEGER. The effective rank of <i>A</i> , that is, the number of singular values which are greater than <i>rcond</i> * <i>s</i> (1).
<i>work</i> (1)	If <i>info</i> = 0, on exit, <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , then the algorithm for computing the SVD failed to converge; <i>i</i> indicates the number of off-diagonal elements of an intermediate bidiagonal form which did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gelss` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (m, n) .

<i>b</i>	Holds the matrix of size $\max(m, n)$ -by- <i>nrhs</i> . On entry, contains the m -by- <i>nrhs</i> right hand side matrix <i>B</i> , On exit, overwritten by the n -by- <i>nrhs</i> solution matrix <i>X</i> .
<i>s</i>	Holds the vector of length $\min(m, n)$.
<i>rcond</i>	Default value for this element is <i>rcond</i> = 100*EPSILON(1.0_WP).

Application Notes

For real flavors:

$$lwork \geq 3 * \min(m, n) + \max(2 * \min(m, n), \max(m, n), nrhs)$$

For complex flavors:

$$lwork \geq 2 * \min(m, n) + \max(m, n, nrhs)$$

For good performance, *lwork* should generally be larger.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?gelsd

Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A and a divide and conquer method.

Syntax

FORTRAN 77:

```
call sgelsd(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, iwork,
info)

call dgelsd(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, iwork,
info)

call cgelsd(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, rwork,
iwork, info)

call zgelsd(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, rwork,
iwork, info)
```

Fortran 95:

```
call gelsd(a, b [,rank] [,s] [,rcond] [,info])
```

Description

This routine computes the minimum-norm solution to a real linear least squares problem:

minimize $\|b - A^*x\|_2$

using the singular value decomposition (SVD) of A . A is an m -by- n matrix which may be rank-deficient.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the m -by- $nrhs$ right hand side matrix B and the n -by- $nrhs$ solution matrix X .

The problem is solved in three steps:

1. Reduce the coefficient matrix A to bidiagonal form with Householder transformations, reducing the original problem into a "bidiagonal least squares problem" (BLS).
2. Solve the BLS using a divide and conquer approach.
3. Apply back all the Householder transformations to solve the original least squares problem.

The effective rank of A is determined by treating as zero those singular values which are less than $rcond$ times the largest singular value.

The routine uses auxiliary routines `?lals0` and `?lalsa`.

Input Parameters

<i>m</i>	INTEGER. The number of rows of the matrix A ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns of the matrix A ($n \geq 0$).
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).
<i>a, b, work</i>	REAL for <code>sgelsd</code> DOUBLE PRECISION for <code>dgelsd</code> COMPLEX for <code>cgelsd</code> DOUBLE COMPLEX for <code>zgelsd</code>. Arrays: <i>a</i>(<i>lda</i>,*) contains the m-by-n matrix A. The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>b</i>(<i>ldb</i>,*) contains the m-by-<i>nrhs</i> right hand side matrix B. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; at least $\max(1, m)$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; must be at least $\max(1, m, n)$.
<i>rcond</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. <i>rcond</i> is used to determine the effective rank of A. Singular values $s(i) \leq rcond * s(1)$ are treated as zero. If <i>rcond</i> ≤ 0, machine precision is used instead.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array; $lwork \geq 1$. If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the array <i>work</i> and the minimum sizes of the arrays <i>rwork</i> and <i>iwork</i>, and

returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* is issued by [xerbla](#).

See *Application Notes* for the suggested value of *lwork*.

iwork INTEGER. Workspace array. See *Application Notes* for the suggested dimension of *iwork*.

rwork REAL for `cgelsd`
DOUBLE PRECISION for `zgelsd`.
Workspace array, used in complex flavors only. See *Application Notes* for the suggested dimension of *rwork*.

Output Parameters

a On exit, *A* has been overwritten.

b Overwritten by the *n*-by-*nrhs* solution matrix *X*.

If $m \geq n$ and $rank = n$, the residual sum-of-squares for the solution in the *i*-th column is given by the sum of squares of modulus of elements $n+1:m$ in that column.

s REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Array, DIMENSION at least $\max(1, \min(m, n))$. The singular values of *A* in decreasing order. The condition number of *A* in the 2-norm is
 $k^2(A) = s(1) / s(\min(m, n))$.

rank INTEGER. The effective rank of *A*, that is, the number of singular values which are greater than $rcond * s(1)$.

work(1) If *info* = 0, on exit, *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

rwork(1) If *info* = 0, on exit, *rwork*(1) returns the minimum size of the workspace array *iwork* required for optimum performance.

iwork(1) If *info* = 0, on exit, *iwork*(1) returns the minimum size of the workspace array *iwork* required for optimum performance.

info INTEGER.

If $info = 0$, the execution is successful.
 If $info = -i$, the i -th parameter had an illegal value.
 If $info = i$, then the algorithm for computing the SVD failed to converge; i indicates the number of off-diagonal elements of an intermediate bidiagonal form that did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gelsd` interface are the following:

a	Holds the matrix A of size (m, n) .
b	Holds the matrix of size $\max(m, n)$ -by- $nrhs$. On entry, contains the m -by- $nrhs$ right hand side matrix B , On exit, overwritten by the n -by- $nrhs$ solution matrix X .
s	Holds the vector of length $\min(m, n)$.
$rcond$	Default value for this element is $rcond = 100 * \text{EPSILON}(1.0\_WP)$.

Application Notes

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

The exact minimum amount of workspace needed depends on m , n and $nrhs$. The size $lwork$ of the workspace array `work` must be as given below.

For real flavors:

If $m \geq n$,

$$lwork \geq 12n + 2n * smlsiz + 8n * nlvl + n * nrhs + (smlsiz + 1)^2;$$

If $m < n$,

$$lwork \geq 12m + 2m * smlsiz + 8m * nlvl + m * nrhs + (smlsiz + 1)^2;$$

For complex flavors:

If $m \geq n$,

$lwork < 2n + n*nrhs;$

If $m < n$,

$lwork \geq 2m + m*nrhs;$

where *smlsiz* is returned by *ilaenv* and is equal to the maximum size of the subproblems at the bottom of the computation tree (usually about 25), and

$nlvl = \text{INT}(\log_2(\min(m, n)/(smlsiz+1))) + 1.$

For good performance, *lwork* should generally be larger.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The dimension of the workspace array *iwork* must be at least

$3*\min(m, n)*nlvl + 11*\min(m, n).$

The dimension of the workspace array *iwork* (for complex flavors) must be at least $\max(1, lrwork).$

$lrwork \geq 10n + 2n*smlsiz + 8n*nlvl + 3*smlsiz*nrhs + (smlsiz+1)^2$ if $m \geq n$, and

$lrwork \geq 10m + 2m*smlsiz + 8m*nlvl + 3*smlsiz*nrhs + (smlsiz+1)^2$ if $m < n$.

Generalized LLS Problems

This section describes LAPACK driver routines used for solving generalized linear least squares problems. [Table 4-9](#) lists all such routines for Fortran 77 interface. Respective routine names in Fortran 95 interface are without the first symbol (see [Routine Naming Conventions](#)).

Table 4-9 Driver Routines for Solving Generalized LLS Problems

Routine Name	Operation performed
?gglse	Solves the linear equality-constrained least squares problem using a generalized RQ factorization.
?ggglm	Solves a general Gauss-Markov linear model problem using a generalized QR factorization.

?gglse

Solves the linear equality-constrained least squares problem using a generalized RQ factorization.

Syntax

FORTRAN 77:

```
call sgglse(m, n, p, a, lda, b, ldb, c, d, x, work, lwork, info)
call dgglse(m, n, p, a, lda, b, ldb, c, d, x, work, lwork, info)
call cgglse(m, n, p, a, lda, b, ldb, c, d, x, work, lwork, info)
call zgglse(m, n, p, a, lda, b, ldb, c, d, x, work, lwork, info)
```

Fortran 95:

```
call gglse(a, b, c, d, x [,info])
```

Description

This routine solves the linear equality-constrained least squares (LSE) problem:

minimize $||c - A*x||^2$ subject to $B*x = d$

where A is an m -by- n matrix, B is a p -by- n matrix, c is a given m -vector, and d is a given p -vector. It is assumed that $p \leq n \leq m+p$, and

$$\text{rank}(B)=p \text{ and } \text{rank} \begin{pmatrix} A \\ B \end{pmatrix} = n .$$

These conditions ensure that the LSE problem has a unique solution, which is obtained using a generalized RQ factorization of the matrices (B, A) given by

$$B = (0 \ R)^T Q, \quad A = Z^T T^T Q$$

Input Parameters

m	INTEGER. The number of rows of the matrix A ($m \geq 0$).
n	INTEGER. The number of columns of the matrices A and B ($n \geq 0$).
p	INTEGER. The number of rows of the matrix B ($0 \leq p \leq n \leq m+p$).
$a, b, c, d, work$	REAL for sgglse DOUBLE PRECISION for dgglse COMPLEX for cgglse DOUBLE COMPLEX for zgglse. Arrays: $a(lda,*)$ contains the m-by-n matrix A. The second dimension of a must be at least $\max(1, n)$. $b(l db,*)$ contains the p-by-n matrix B. The second dimension of b must be at least $\max(1, n)$. $c(*)$, dimension at least $\max(1, m)$, contains the right hand side vector for the least squares part of the LSE problem. $d(*)$, dimension at least $\max(1, p)$, contains the right hand side vector for the constrained equation. $work$ is a workspace array, its dimension $\max(1, lwork)$.
lda	INTEGER. The first dimension of a ; at least $\max(1, m)$.
ldb	INTEGER. The first dimension of b ; at least $\max(1, p)$.
$lwork$	INTEGER. The size of the $work$ array; $lwork \geq \max(1, m+n+p)$. If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by xerbla. See <i>Application Notes</i> for the suggested value of $lwork$.

Output Parameters

<i>x</i>	REAL for <i>sggls</i> DOUBLE PRECISION for <i>dggls</i> COMPLEX for <i>cggls</i> DOUBLE COMPLEX for <i>zggls</i>. Array, DIMENSION at least $\max(1, n)$. On exit, contains the solution of the LSE problem.
<i>a</i>	On exit, the elements on and above the diagonal of the array contain the $\min(m, n)$-by-n upper trapezoidal matrix T.
<i>b</i>	On exit, the upper triangle of the subarray $b(1:p, n-p+1:n)$ contains the p-by-p upper triangular matrix R.
<i>d</i>	On exit, <i>d</i> is destroyed.
<i>c</i>	On exit, the residual sum-of-squares for the solution is given by the sum of squares of elements $n-p+1$ to m of vector <i>c</i>.
<i>work</i> (1)	If <i>info</i> = 0, on exit, <i>work</i>(1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value. If <i>info</i> = 1, the upper triangular factor R associated with B in the generalized RQ factorization of the pair (B, A) is singular, so that $\text{rank}(B) < P$; the least squares solution could not be computed. If <i>info</i> = 2, the $(n-p)$-by-$(n-p)$ part of the upper trapezoidal factor T associated with A in the generalized RQ factorization of the pair (B, A) is singular, so that

$$\text{rank} \begin{pmatrix} A \\ B \end{pmatrix} < n$$

; the least squares solution could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gglse` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (m, n) .
<i>b</i>	Holds the matrix <i>B</i> of size (p, n) .
<i>c</i>	Holds the vector of length (m) .
<i>d</i>	Holds the vector of length (p) .
<i>x</i>	Holds the vector of length (n) .

Application Notes

For optimum performance, use

$$lwork \geq p + \min(m, n) + \max(m, n) * nb,$$

where *nb* is an upper bound for the optimal blocksizes for `?geqrf`, `?gerqf`, `?ormqr`/`?unmqr` and `?ormrq`/`?unmrq`.

You may set *lwork* to -1. The routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?ggglm

Solves a general Gauss-Markov linear model problem using a generalized QR factorization.

Syntax

FORTRAN 77:

```
call sggglm(n, m, p, a, lda, b, ldb, d, x, y, work, lwork, info)
call dggglm(n, m, p, a, lda, b, ldb, d, x, y, work, lwork, info)
call cggglm(n, m, p, a, lda, b, ldb, d, x, y, work, lwork, info)
call zggglm(n, m, p, a, lda, b, ldb, d, x, y, work, lwork, info)
```

Fortran 95:

```
call ggglm(a, b, d, x, y [,info])
```

Description

This routine solves a general Gauss-Markov linear model (GLM) problem:

minimize_x ||y||₂ subject to $d = A^*x + B^*y$

where A is an n -by- m matrix, B is an n -by- p matrix, and d is a given n -vector. It is assumed that $m \leq n \leq m+p$, and $\text{rank}(A) = m$ and $\text{rank}(A \ B) = n$.

Under these assumptions, the constrained equation is always consistent, and there is a unique solution x and a minimal 2-norm solution y , which is obtained using a generalized QR factorization of the matrices $(A, \ B)$ given by

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix}; \quad B = Q^* T^* Z.$$

In particular, if matrix B is square nonsingular, then the problem GLM is equivalent to the following weighted linear least squares problem

minimize_x || $B^{-1}(d - A^*x)$ ||₂.

Input Parameters

<i>n</i>	INTEGER. The number of rows of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>m</i>	INTEGER. The number of columns in <i>A</i> ($m \geq 0$).
<i>p</i>	INTEGER. The number of columns in <i>B</i> ($p \geq n - m$).
<i>a, b, d, work</i>	REAL for sggglm DOUBLE PRECISION for dggglm COMPLEX for cggglm DOUBLE COMPLEX for zggglm. Arrays: <i>a(lda,*)</i> contains the <i>n</i> -by- <i>m</i> matrix <i>A</i> . The second dimension of <i>a</i> must be at least $\max(1, m)$. <i>b(l db,*)</i> contains the <i>n</i> -by- <i>p</i> matrix <i>B</i> . The second dimension of <i>b</i> must be at least $\max(1, p)$. <i>d(*)</i> , dimension at least $\max(1, n)$, contains the left hand side of the GLM equation. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; at least $\max(1, n)$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; at least $\max(1, n)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array; $lwork \geq \max(1, n+m+p)$. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla. See <i>Application Notes</i> for the suggested value of <i>lwork</i> .

Output Parameters

<i>x, y</i>	REAL for sggglm DOUBLE PRECISION for dggglm COMPLEX for cggglm DOUBLE COMPLEX for zggglm. Arrays <i>x</i> (*), <i>y</i> (*). DIMENSION at least $\max(1, m)$ for <i>x</i> and at least $\max(1, p)$ for <i>y</i> .

	On exit, x and y are the solutions of the GLM problem.
a	On exit, the upper triangular part of the array a contains the m -by- m upper triangular matrix R .
b	On exit, if $n \leq p$, the upper triangle of the subarray $b(1:n, p-n+1:p)$ contains the n -by- n upper triangular matrix T ; if $n > p$, the elements on and above the $(n-p)$ -th subdiagonal contain the n -by- p upper trapezoidal matrix T .
d	On exit, d is destroyed
$work(1)$	If $info = 0$, on exit, $work(1)$ contains the minimum value of $lwork$ required for optimum performance.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value. If $info = 1$, the upper triangular factor R associated with A in the generalized QR factorization of the pair (A, B) is singular, so that $rank(A) < m$; the least squares solution could not be computed. If $info = 2$, the bottom $(n-m)$ -by- $(n-m)$ part of the upper trapezoidal factor T associated with B in the generalized QR factorization of the pair (A, B) is singular, so that $rank(A, B) < n$; the least squares solution could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ggglm` interface are the following:

a	Holds the matrix A of size (n, m) .
b	Holds the matrix B of size (n, p) .
d	Holds the vector of length (n) .
x	Holds the vector of length (m) .
y	Holds the vector of length (p) .

Application Notes

For optimum performance, use

$$lwork \geq m + \min(n, p) + \max(n, p)*nb,$$

where *nb* is an upper bound for the optimal blocksizes for `?geqrf`, `?gerqf`, `?ormqr`/`?unmqr` and `?ormrq`/`?unmrq`.

You may set *lwork* to -1. The routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Symmetric Eigenproblems

This section describes LAPACK driver routines used for solving symmetric eigenvalue problems. See also computational routines [computational routines](#) that can be called to solve these problems. [Table 4-10](#) lists all such driver routines for Fortran 77 interface. Respective routine names in Fortran 95 interface are without the first symbol (see [Routine Naming Conventions](#)).

Table 4-10 Driver Routines for Solving Symmetric Eigenproblems

Routine Name	Operation performed
?syev / ?heev	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix.
?syevd / ?heevd	Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric / Hermitian matrix using divide and conquer algorithm.
?syevx / ?heevx	Computes selected eigenvalues and, optionally, eigenvectors of a symmetric / Hermitian matrix.
?syevr / ?heevr	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix using the Relatively Robust Representations.
?spev / ?hpev	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix in packed storage.

Routine Name	Operation performed
?spevd / ?hpevd	Uses divide and conquer algorithm to compute all eigenvalues and (optionally) all eigenvectors of a real symmetric / Hermitian matrix held in packed storage.
?spevx / ?hpevx	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix in packed storage.
?sbev / ?hbev	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian band matrix.
?sbevd / ?hbevd	Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric / Hermitian band matrix using divide and conquer algorithm.
?sbevz / ?hbevz	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian band matrix.
?stev	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix.
?stevd	Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric tridiagonal matrix using divide and conquer algorithm.
?stevx	Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.
?stevr	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix using the Relatively Robust Representations.

[?syev](#)

Computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix.

Syntax

FORTRAN 77:

```
call ssyev(jobz, uplo, n, a, lda, w, work, lwork, info)
call dsyev(jobz, uplo, n, a, lda, w, work, lwork, info)
```

Fortran 95:

```
call syev(a, w [,jobz] [,uplo] [,info])
```

Description

This routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A .

Note that for most cases of real symmetric eigenvalue problems the default choice should be [?syevr](#) function as its underlying algorithm is faster and uses less workspace.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of A . If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of A .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>a, work</i>	REAL for <code>ssyev</code> DOUBLE PRECISION for <code>dsyev</code> Arrays: <i>a</i> (<i>lda</i> ,*) is an array containing either upper or lower triangular part of the symmetric matrix A , as specified by <i>uplo</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The first dimension of the array <i>a</i> . Must be at least $\max(1, n)$.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . Constraint: $lwork \geq \max(1, 3n-1)$. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla .

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

<i>a</i>	On exit, if <i>jobz</i> = 'V', then if <i>info</i> = 0, array <i>a</i> contains the orthonormal eigenvectors of the matrix <i>A</i> . If <i>jobz</i> = 'N', then on exit the lower triangle (if <i>uplo</i> = 'L') or the upper triangle (if <i>uplo</i> = 'U') of <i>A</i> , including the diagonal, is overwritten.
<i>w</i>	REAL for ssyev DOUBLE PRECISION for dsyev Array, DIMENSION at least max(1, <i>n</i>). If <i>info</i> = 0, contains the eigenvalues of the matrix <i>A</i> in ascending order.
<i>work</i> (1)	On exit, if <i>lwork</i> > 0, then <i>work</i> (1) returns the required minimal size of <i>lwork</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , then the algorithm failed to converge; <i>i</i> indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *syev* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>w</i>	Holds the vector of length (<i>n</i>).
<i>job</i>	Must be 'N' or 'V'. The default value is 'N'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For optimum performance set $lwork \geq (nb+2) * n$, where *nb* is the blocksize for ?sytrd returned by ilaenv.

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible `lwork` sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

If it is not clear how much workspace to supply, use a generous value of `lwork` for the first run, or set `lwork = -1`.

If `lwork` has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array on exit. Use this value (`work(1)`) for subsequent runs.

If `lwork = -1`, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array `work`. This operation is called a workspace query.

Note that if `lwork` is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

cheev

Computes all eigenvalues and, optionally, eigenvectors of a Hermitian matrix.

Syntax

FORTRAN 77:

```
call cheev(jobz, uplo, n, a, lda, w, work, lwork, rwork, info)
call zheev(jobz, uplo, n, a, lda, w, work, lwork, rwork, info)
```

Fortran 95:

```
call heev(a, w [,jobz] [,uplo] [,info])
```

Description

This routine computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A .

Note that for most cases of complex Hermitian eigenvalue problems the default choice should be [?heevr](#) function as its underlying algorithm is faster and uses less workspace.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of A . If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of A .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>a, work</i>	COMPLEX for cheev DOUBLE COMPLEX for zheev Arrays: <i>a</i> (<i>lda</i> ,*) is an array containing either upper or lower triangular part of the Hermitian matrix A , as specified by <i>uplo</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The first dimension of the array <i>a</i> . Must be at least $\max(1, n)$.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . C onstraint: $lwork \geq \max(1, 2n-1)$. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla .

rwork See *Application Notes* for the suggested value of *lwork*.
 REAL for cheev
 DOUBLE PRECISION for zheev.
 Workspace array, DIMENSION at least $\max(1, 3n-2)$.

Output Parameters

a On exit, if *jobz* = 'V', then if *info* = 0, array *a* contains the orthonormal eigenvectors of the matrix *A*.
 If *jobz* = 'N', then on exit the lower triangle (if *uplo* = 'L') or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.

w REAL for cheev
 DOUBLE PRECISION for zheev
 Array, DIMENSION at least $\max(1, n)$.
 If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order.

work(1) On exit, if *lwork* > 0, then *work(1)* returns the required minimal size of *lwork*.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.
 If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *heev* interface are the following:

a Holds the matrix *A* of size (*n*, *n*).
w Holds the vector of length (*n*).
job Must be 'N' or 'V'. The default value is 'N'.
uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For optimum performance use

$$lwork \geq (nb+1) * n,$$

where nb is the blocksize for `?hetrd` returned by `ilaenv`.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?syevd

Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric matrix using divide and conquer algorithm.

Syntax

FORTRAN 77:

```
call ssyevd(jobz, uplo, n, a, lda, w, work, lwork, iwork, liwork, info)
call dsyevd(jobz, uplo, n, a, lda, w, work, lwork, iwork, liwork, info)
```

Fortran 95:

```
call syevd(a, w [,jobz] [,uplo] [,info])
```

Description

This routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric matrix A . In other words, it can compute the spectral factorization of A as: $A = Z \Lambda Z^T$.

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i , and Z is the orthogonal matrix whose columns are the eigenvectors z_i . Thus,

$$A * z_i = \lambda_i * z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the *QL* or *QR* algorithm.

Note that for most cases of real symmetric eigenvalue problems the default choice should be `?syevr` function as its underlying algorithm is faster and uses less workspace. `?syevd` requires more workspace but is faster in some cases, especially for large matrices.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of <i>A</i>. If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of <i>A</i>.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).
<i>a</i>	REAL for <code>ssyevd</code> DOUBLE PRECISION for <code>dsyevd</code> Array, DIMENSION (<i>lda</i>, *). <i>a</i>(<i>lda</i>,*) is an array containing either upper or lower triangular part of the symmetric matrix <i>A</i>, as specified by <i>uplo</i>. The second dimension of <i>a</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The first dimension of the array <i>a</i>. Must be at least $\max(1, n)$.
<i>work</i>	REAL for <code>ssyevd</code> DOUBLE PRECISION for <code>dsyevd</code>. Workspace array, DIMENSION at least <i>lwork</i>.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i>. Constraints:

if $n \leq 1$, then $lwork \geq 1$;
 if $jobz = 'N'$ and $n > 1$, then $lwork \geq 2*n + 1$;
 if $jobz = 'V'$ and $n > 1$, then $lwork \geq 2*n^2 + 6*n + 1$.
 If $lwork = -1$, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

iwork

INTEGER.

Workspace array, its dimension $\max(1, liwork)$.*liwork*

INTEGER.

The dimension of the array *iwork*.

Constraints:

if $n \leq 1$, then $liwork \geq 1$;
 if $jobz = 'N'$ and $n > 1$, then $liwork \geq 1$;
 if $jobz = 'V'$ and $n > 1$, then $liwork \geq 5*n + 3$.
 If $liwork = -1$, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

w

REAL for ssyevd

DOUBLE PRECISION for dsyevd

Array, DIMENSION at least $\max(1, n)$.

If $info = 0$, contains the eigenvalues of the matrix *A* in ascending order. See also *info*.

a

If $jobz = 'V'$, then on exit this array is overwritten by the orthogonal matrix *Z* which contains the eigenvectors of *A*.

work(1)

On exit, if $lwork > 0$, then *work*(1) returns the required minimal size of *lwork*.

<i>iwork</i> (1)	On exit, if <i>liwork</i> > 0, then <i>iwork</i> (1) returns the required minimal size of <i>liwork</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>i</i>, then the algorithm failed to converge; <i>i</i> indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero. If <i>info</i> = <i>i</i>, and <i>jobz</i> = 'V', then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns <i>info</i>/(<i>n</i>+1) through <i>mod</i>(<i>info</i>,<i>n</i>+1). If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *syevd* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>w</i>	Holds the vector of length (<i>n</i>).
<i>jobz</i>	Must be 'N' or 'V'. The default value is 'N'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T+E$ such that $\|E\|_2 = O(\epsilon) * \|T\|_2$, where ϵ is the machine precision.

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run, or set *lwork* = -1 (*liwork* = -1).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If $lwork = -1$ ($liwork = -1$), then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work, iwork$). This operation is called a workspace query.

Note that if $lwork$ ($liwork$) is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex analogue of this routine is [?heevd](#)

?heevd

Computes all eigenvalues and (optionally) all eigenvectors of a complex Hermitian matrix using divide and conquer algorithm.

Syntax

FORTTRAN 77:

```
call cheevd(jobz, uplo, n, a, lda, w, work, lwork, rwork, lrwork, iwork,
liwork, info)

call zheevd(jobz, uplo, n, a, lda, w, work, lwork, rwork, lrwork, iwork,
liwork, info)
```

Fortran 95:

```
call heevd(a, w [,job] [,uplo] [,info])
```

Description

This routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian matrix A . In other words, it can compute the spectral factorization of A as: $A = Z \Lambda Z^H$.

Here Λ is a real diagonal matrix whose diagonal elements are the eigenvalues λ_i , and Z is the (complex) unitary matrix whose columns are the eigenvectors z_i . Thus,

$$A z_i = \lambda_i z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

Note that for most cases of complex Hermetian eigenvalue problems the default choice should be `?heevr` function as its underlying algorithm is faster and uses less workspace. `?heevd` requires more workspace but is faster in some cases, especially for large matrices.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of <i>A</i>. If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of <i>A</i>.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).
<i>a</i>	COMPLEX for <code>cheevd</code> DOUBLE COMPLEX for <code>zheevd</code> Array, DIMENSION (<i>lda</i>, *). <i>a</i>(<i>lda</i>,*) is an array containing either upper or lower triangular part of the Hermitian matrix <i>A</i>, as specified by <i>uplo</i>. The second dimension of <i>a</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The first dimension of the array <i>a</i> . Must be at least $\max(1, n)$.
<i>work</i>	COMPLEX for <code>cheevd</code> DOUBLE COMPLEX for <code>zheevd</code>. Workspace array, DIMENSION $\max(1, lwork)$.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i>. Constraints: if $n \leq 1$, then $lwork \geq 1$; if <i>jobz</i> = 'N' and $n > 1$, then $lwork \geq n+1$; if <i>jobz</i> = 'V' and $n > 1$, then $lwork \geq n^2+2*n$. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i>, <i>rwork</i> and <i>iwork</i> arrays, returns these values as the first entries

of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

rwork REAL for cheevd
DOUBLE PRECISION for zheevd
Workspace array, DIMENSION at least *lrwork*.

lrwork INTEGER.
The dimension of the array *rwork*. Constraints:
if $n \leq 1$, then $lrwork \geq 1$;
if $job = 'N'$ and $n > 1$, then $lrwork \geq n$;
if $job = 'V'$ and $n > 1$, then $lrwork \geq 2*n^2 + 5*n + 1$.
If $lrwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

iwork INTEGER. Workspace array, its dimension $\max(1, liwork)$.

liwork INTEGER.
The dimension of the array *iwork*. Constraints: if $n \leq 1$, then $liwork \geq 1$;
if $jobz = 'N'$ and $n > 1$, then $liwork \geq 1$;
if $jobz = 'V'$ and $n > 1$, then $liwork \geq 5*n+3$.
If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

w REAL for cheevd
DOUBLE PRECISION for zheevd
Array, DIMENSION at least $\max(1, n)$.

	If <i>info</i> = 0, contains the eigenvalues of the matrix <i>A</i> in ascending order. See also <i>info</i> .
<i>a</i>	If <i>jobz</i> = 'V', then on exit this array is overwritten by the unitary matrix <i>Z</i> which contains the eigenvectors of <i>A</i> .
<i>work</i> (1)	On exit, if <i>lwork</i> > 0, then the real part of <i>work</i> (1) returns the required minimal size of <i>lwork</i> .
<i>rwork</i> (1)	On exit, if <i>lrwork</i> > 0, then <i>rwork</i> (1) returns the required minimal size of <i>lrwork</i> .
<i>iwork</i> (1)	On exit, if <i>liwork</i> > 0, then <i>iwork</i> (1) returns the required minimal size of <i>liwork</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>i</i> , and <i>jobz</i> = 'N', then the algorithm failed to converge; <i>i</i> off-diagonal elements of an intermediate tridiagonal form did not converge to zero; if <i>info</i> = <i>i</i> , and <i>jobz</i> = 'V', then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns <i>info</i> /(<i>n</i> +1) through mod(<i>info</i> , <i>n</i> +1). If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `heevd` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>w</i>	Holds the vector of length (<i>n</i>).
<i>jobz</i>	Must be 'N' or 'V'. The default value is 'N'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $A + E$ such that $\|E\|_2 = O(\epsilon) * \|A\|_2$, where ϵ is the machine precision.

If you are in doubt how much workspace to supply, use a generous value of `lwork` (`liwork` or `lrwork`) for the first run or set `lwork = -1` (`liwork = -1`, `lrwork = -1`).

If you choose the first option and set any of admissible `lwork` (`liwork` or `lrwork`) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (`work`, `iwork`, `rwork`) on exit. Use this value (`work(1)`, `iwork(1)`, `rwork(1)`) for subsequent runs.

If you set `lwork = -1` (`liwork = -1`, `lrwork = -1`), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`, `iwork`, `rwork`). This operation is called a workspace query.

Note that if you set `lwork` (`liwork`, `lrwork`) to less than the minimal required value and not `-1`, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real analogue of this routine is [?syevd](#). See also [?hpevd](#) for matrices held in packed storage, and [?hbevd](#) for banded matrices.

?syevx

Computes selected eigenvalues and, optionally, eigenvectors of a symmetric matrix.

Syntax

FORTRAN 77:

```
call ssyevx(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z,
ldz, work, lwork, iwork, ifail, info)
```

```
call dsyevx(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z,
ldz, work, lwork, iwork, ifail, info)
```

Fortran 95:

```
call syevx(a, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol]
[,info])
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Note that for most cases of real symmetric eigenvalue problems the default choice should be `?syevr` function as its underlying algorithm is faster and uses less workspace. `?syevx` is faster for a few selected eigenvalues.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be 'A', 'V', or 'I'. If <i>range</i> = 'A', all eigenvalues will be found. If <i>range</i> = 'V', all eigenvalues in the half-open interval (<i>vl</i>, <i>vu</i>] will be found. If <i>range</i> = 'I', the eigenvalues with indices <i>il</i> through <i>iu</i> will be found.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of <i>A</i>. If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of <i>A</i>.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).
<i>a</i> , <i>work</i>	REAL for <code>ssyevx</code> DOUBLE PRECISION for <code>dsyevx</code>. Arrays: <i>a</i>(<i>lda</i>,*) is an array containing either upper or lower triangular part of the symmetric matrix <i>A</i>, as specified by <i>uplo</i>. The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The first dimension of the array <i>a</i>. Must be at least $\max(1, n)$.
<i>vl</i> , <i>vu</i>	REAL for <code>ssyevx</code> DOUBLE PRECISION for <code>dsyevx</code>. If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues; $vl \leq vu$. Not referenced if <i>range</i> = 'A' or 'I'.
<i>il</i> , <i>iu</i>	INTEGER.

If $range = 'I'$, the indices of the smallest and largest eigenvalues to be returned.

Constraints: $1 \leq il \leq iu \leq n$, if $n > 0$;

$il = 1$ and $iu = 0$, if $n = 0$.

Not referenced if $range = 'A'$ or $'V'$.

abstol

REAL for `ssyevx`

DOUBLE PRECISION for `dsyevx`.

The absolute error tolerance for the eigenvalues. See *Application Notes* for more information.

ldz

INTEGER. The first dimension of the output array *z*; $ldz \geq 1$.

If $jobz = 'V'$, then $ldz \geq \max(1, n)$.

lwork

INTEGER.

The dimension of the array *work*.

If $n \leq 1$ then $lwork \geq 1$, otherwise $lwork = 8 * n$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by `xerbla`.

See *Application Notes* for the suggested value of *lwork*.

iwork

INTEGER. Workspace array, DIMENSION at least $\max(1, 5n)$.

Output Parameters

a

On exit, the lower triangle (if $uplo = 'L'$) or the upper triangle (if $uplo = 'U'$) of *A*, including the diagonal, is overwritten.

m

INTEGER. The total number of eigenvalues found;

$0 \leq m \leq n$.

If $range = 'A'$, $m = n$, and if $range = 'I'$, $m = iu - il + 1$.

w

REAL for `ssyevx`

DOUBLE PRECISION for `dsyevx`

Array, DIMENSION at least $\max(1, n)$. The first *m* elements contain the selected eigenvalues of the matrix *A* in ascending order.

<i>z</i>	REAL for <i>ssyevx</i> DOUBLE PRECISION for <i>dsyevx</i>. Array <i>z(ldz,*)</i> contains eigenvectors. The second dimension of <i>z</i> must be at least $\max(1, m)$. If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix <i>A</i> corresponding to the selected eigenvalues, with the <i>i</i>-th column of <i>z</i> holding the eigenvector associated with <i>w(i)</i>. If an eigenvector fails to converge, then that column of <i>z</i> contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in <i>ifail</i>. If <i>jobz</i> = 'N', then <i>z</i> is not referenced. Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array <i>z</i>; if <i>range</i> = 'V', the exact value of <i>m</i> is not known in advance and an upper bound must be used.
<i>work(1)</i>	On exit, if <i>lwork</i> > 0, then <i>work(1)</i> returns the required minimal size of <i>lwork</i>.
<i>ifail</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> elements of <i>ifail</i> are zero; if <i>info</i> > 0, then <i>ifail</i> contains the indices of the eigenvectors that failed to converge. If <i>jobz</i> = 'V', then <i>ifail</i> is not referenced.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value. If <i>info</i> = <i>i</i>, then <i>i</i> eigenvectors failed to converge; their indices are stored in the array <i>ifail</i>.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *syevx* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (n, n) .

<i>w</i>	Holds the vector of length (<i>n</i>).
<i>a</i>	Holds the matrix <i>A</i> of size (<i>m</i> , <i>n</i>).
<i>ifail</i>	Holds the vector of length (<i>n</i>).
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>vl</i>	Default value for this element is <i>vl</i> = -HUGE(<i>vl</i>).
<i>vu</i>	Default value for this element is <i>vu</i> = HUGE(<i>vl</i>).
<i>il</i>	Default value for this argument is <i>il</i> = 1.
<i>iu</i>	Default value for this argument is <i>iu</i> = <i>n</i> .
<i>abstol</i>	Default value for this element is <i>abstol</i> = 0.0_WP.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted Note that there will be an error condition if <i>ifail</i> is present and <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: <i>range</i> = 'V', if one of or both <i>vl</i> and <i>vu</i> are present, <i>range</i> = 'I', if one of or both <i>il</i> and <i>iu</i> are present, <i>range</i> = 'A', if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present, Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

For optimum performance use $lwork \geq (nb+3)*n$, where *nb* is the maximum of the blocksize for ?sytrd and ?ormtr returned by ilaenv.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If *lwork* has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array *work*. This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon \max(|a|, |b|)$, where ϵ is the machine precision.

If $abstol$ is less than or equal to zero, then $\epsilon * |T|$ is used as tolerance, where $|T|$ is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form. Eigenvalues are computed most accurately when $abstol$ is set to twice the underflow threshold $2 * \text{slamch}('S')$, not zero.

If this routine returns with $info > 0$, indicating that some eigenvectors did not converge, try setting $abstol$ to $2 * \text{slamch}('S')$.

?heevx

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix.

Syntax

FORTRAN 77:

```
call cheevx(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z,
ldz, work, lwork, rwork, iwork, ifail, info)

call zheevx(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z,
ldz, work, lwork, rwork, iwork, ifail, info)
```

Fortran 95:

```
call heevx(a, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol]
[,info])
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Note that for most cases of complex Hermetian eigenvalue problems the default choice should be [?heevr](#) function as its underlying algorithm is faster and uses less workspace. [?heevx](#) is faster for a few selected eigenvalues.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed.

If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

range CHARACTER*1. Must be 'A', 'V', or 'I'.
 If *range* = 'A', all eigenvalues will be found.
 If *range* = 'V', all eigenvalues in the half-open interval (*vl*, *vu*] will be found.
 If *range* = 'I', the eigenvalues with indices *il* through *iu* will be found.

uplo CHARACTER*1. Must be 'U' or 'L'.
 If *uplo* = 'U', *a* stores the upper triangular part of *A*.
 If *uplo* = 'L', *a* stores the lower triangular part of *A*.

n INTEGER. The order of the matrix *A* ($n \geq 0$).

a, *work* COMPLEX for cheevx
 DOUBLE COMPLEX for zheevx.
 Arrays:
a(*lda*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A*, as specified by *uplo*.
 The second dimension of *a* must be at least $\max(1, n)$.
work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The first dimension of the array *a*. Must be at least $\max(1, n)$.

vl, *vu* REAL for cheevx
 DOUBLE PRECISION for zheevx.
 If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues; $vl \leq vu$. Not referenced if *range* = 'A' or 'I'.

il, *iu* INTEGER.
 If *range* = 'I', the indices of the smallest and largest eigenvalues to be returned. Constraints:
 $1 \leq il \leq iu \leq n$, if $n > 0$; $il = 1$ and $iu = 0$, if $n = 0$. Not referenced if *range* = 'A' or 'V'.

abstol REAL for cheevx
 DOUBLE PRECISION for zheevx. The absolute error tolerance for the eigenvalues. See *Application Notes* for more information.

<i>ldz</i>	INTEGER. The first dimension of the output array <i>z</i>; $ldz \geq 1$. If <i>jobz</i> = 'V', then $ldz \geq \max(1, n)$.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i>. $lwork \geq 1$ if $n \leq 1$; otherwise at least $2*n$. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla. See <i>Application Notes</i> for the suggested value of <i>lwork</i>.
<i>rwork</i>	REAL for cheevx DOUBLE PRECISION for zheevx. Workspace array, DIMENSION at least $\max(1, 7n)$.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, 5n)$.

Output Parameters

<i>a</i>	On exit, the lower triangle (if <i>uplo</i> = 'L') or the upper triangle (if <i>uplo</i> = 'U') of <i>A</i>, including the diagonal, is overwritten.
<i>m</i>	INTEGER. The total number of eigenvalues found; $0 \leq m \leq n$. If <i>range</i> = 'A', $m = n$, and if <i>range</i> = 'I', $m = iu-il+1$.
<i>w</i>	REAL for cheevx DOUBLE PRECISION for zheevx Array, DIMENSION at least $\max(1, n)$. The first <i>m</i> elements contain the selected eigenvalues of the matrix <i>A</i> in ascending order.
<i>z</i>	COMPLEX for cheevx DOUBLE COMPLEX for zheevx. Array <i>z</i>(<i>ldz</i>,*) contains eigenvectors. The second dimension of <i>z</i> must be at least $\max(1, m)$.

If $jobz = 'V'$, then if $info = 0$, the first m columns of z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i -th column of z holding the eigenvector associated with $w(i)$. If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in $ifail$.

If $jobz = 'N'$, then z is not referenced. Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array z ; if $range = 'V'$, the exact value of m is not known in advance and an upper bound must be used.

$work(1)$ On exit, if $lwork > 0$, then $work(1)$ returns the required minimal size of $lwork$.

$ifail$ INTEGER.
Array, DIMENSION at least $\max(1, n)$.
If $jobz = 'V'$, then if $info = 0$, the first m elements of $ifail$ are zero; if $info > 0$, then $ifail$ contains the indices of the eigenvectors that failed to converge.
If $jobz = 'V'$, then $ifail$ is not referenced.

$info$ INTEGER.
If $info = 0$, the execution is successful.
If $info = -i$, the i -th parameter had an illegal value.
If $info = i$, then i eigenvectors failed to converge; their indices are stored in the array $ifail$.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `heevx` interface are the following:

a	Holds the matrix A of size (n, n) .
w	Holds the vector of length (n) .
z	Holds the matrix Z of size (n, n) .
$ifail$	Holds the vector of length (n) .
$uplo$	Must be 'U' or 'L'. The default value is 'U'.

<i>vl</i>	Default value for this element is $vl = -HUGE(vl)$.
<i>vu</i>	Default value for this element is $vu = HUGE(vl)$.
<i>il</i>	Default value for this argument is $il = 1$.
<i>iu</i>	Default value for this argument is $iu = n$.
<i>abstol</i>	Default value for this element is $abstol = 0.0\_WP$.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted Note that there will be an error condition if <i>ifail</i> is present and <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: <i>range</i> = 'V', if one of or both <i>vl</i> and <i>vu</i> are present, <i>range</i> = 'I', if one of or both <i>il</i> and <i>iu</i> are present, <i>range</i> = 'A', if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present, Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

For optimum performance use $lwork \geq (nb+1)*n$, where *nb* is the maximum of the blocksize for ?hetrd and ?unmtr returned by [ilaenv](#).

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to $abstol + \epsilon * \max(|a|, |b|)$, where ϵ is the machine precision.

If `abstol` is less than or equal to zero, then $\epsilon * |T|$ will be used in its place, where $|T|$ is the 1-norm of the tridiagonal matrix obtained by reducing `A` to tridiagonal form. Eigenvalues will be computed most accurately when `abstol` is set to twice the underflow threshold `2*slamch('S')`, not zero.

If this routine returns with `info > 0`, indicating that some eigenvectors did not converge, try setting `abstol` to `2*slamch('S')`.

?syevr

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix using the Relatively Robust Representations.

Syntax

FORTRAN 77:

```
call ssyevr(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z,
ldz, isuppz, work, lwork, iwork, liwork, info)

call dsyevr(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z,
ldz, isuppz, work, lwork, iwork, liwork, info)
```

Fortran 95:

```
call syevr(a, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,isuppz] [,abstol]
[,info])
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix `A`. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

The routine first reduces the matrix `A` to tridiagonal form `T` with a call to [?sytrd](#). Then, whenever possible, [?syevr](#) calls [?stemr](#) to compute the eigenspectrum using Relatively Robust Representations. [?stemr](#) computes eigenvalues by the *dqds* algorithm, while orthogonal eigenvectors are computed from various “good” $L^*D^*L^T$ representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For the each unreduced block of `T`:

- a.** Compute $T - \sigma^* I = L^* D^* L^T$, so that L and D define all the wanted eigenvalues to high relative accuracy. This means that small relative changes in the entries of D and L cause only small relative changes in the eigenvalues and eigenvectors. The standard (unfactored) representation of the tridiagonal matrix T does not have this property in general.
- b.** Compute the eigenvalues to suitable accuracy. If the eigenvectors are desired, the algorithm attains full accuracy of the computed eigenvalues only right before the corresponding vectors have to be computed, see Steps c) and d).
- c.** For each cluster of close eigenvalues, select a new shift close to the cluster, find a new factorization, and refine the shifted eigenvalues to suitable accuracy.
- d.** For each eigenvalue with a large enough relative separation, compute the corresponding eigenvector by forming a rank revealing twisted factorization. Go back to Step c) for any clusters that remain.

The desired accuracy of the output can be specified by the input parameter *abstol*.

The routine `?syevr` calls `?stemr` when the full spectrum is requested on machines that conform to the IEEE-754 floating point standard. `?syevr` calls `?stebz` and `?stein` on non-IEEE machines and when partial spectrum requests are made.

Note that `?syevr` is preferable for most cases of real symmetric eigenvalue problems as its underlying algorithm is fast and uses less workspace.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues $\lambda(i)$ in the half-open interval: $vl < \lambda(i) \leq vu.$ If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i>. For <i>range</i> = 'V' or 'I' and $iu - il < n - 1$, <code>sstebz/dstebz</code> and <code>sstein/dstein</code> are called.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', *a* stores the upper triangular part of *A*.
 If *uplo* = 'L', *a* stores the lower triangular part of *A*.

n INTEGER. The order of the matrix *A* ($n \geq 0$).

a, *work* REAL for ssyevr
 DOUBLE PRECISION for dsyevr.

Arrays:
a(*lda*,*) is an array containing either upper or lower triangular part of the symmetric matrix *A*, as specified by *uplo*.
 The second dimension of *a* must be at least $\max(1, n)$.
work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The first dimension of the array *a*. Must be at least $\max(1, n)$.

vl, *vu* REAL for ssyevr
 DOUBLE PRECISION for dsyevr.
 If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.
 Constraint: $vl < vu$.
 If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu* INTEGER.
 If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.
 Constraint:
 $1 \leq il \leq iu \leq n$, if $n > 0$;
 $il=1$ and $iu=0$, if $n = 0$.
 If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol REAL for ssyevr
 DOUBLE PRECISION for dsyevr. The absolute error tolerance to which each eigenvalue/eigenvector is required.
 If *jobz* = 'V', the eigenvalues and eigenvectors output have residual norms bounded by *abstol*, and the dot products between different eigenvectors are bounded by *abstol*.

If $abstol < n * \epsilon * |T|$, then $n * \epsilon * |T|$ is used instead, where ϵ is the machine precision, and $|T|$ is the 1-norm of the matrix T . The eigenvalues are computed to an accuracy of $\epsilon * |T|$ irrespective of $abstol$.
If high relative accuracy is important, set $abstol$ to `?lamch('S')`.

ldz INTEGER. The leading dimension of the output array *z*.
Constraints:

$ldz \geq 1$ if $jobz = 'N'$;
 $ldz < \max(1, n)$ if $jobz = 'V'$.

lwork INTEGER.
The dimension of the array *work*.

Constraint: $lwork \geq \max(1, 26n)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#).
See *Application Notes* for the suggested value of *lwork*.

iwork INTEGER. Workspace array, its dimension $\max(1, liwork)$.

liwork INTEGER.
The dimension of the array *iwork*, $lwork \geq \max(1, 10n)$.
If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *iwork* array, returns this value as the first entry of the *iwork* array, and no error message related to *liwork* is issued by [xerbla](#).

Output Parameters

a On exit, the lower triangle (if $uplo = 'L'$) or the upper triangle (if $uplo = 'U'$) of A , including the diagonal, is overwritten.

m INTEGER. The total number of eigenvalues found, $0 \leq m \leq n$.
If $range = 'A'$, $m = n$, and if $range = 'I'$, $m = iu - il + 1$.

w, z REAL for `ssyevr`
DOUBLE PRECISION for `dsyevr`.

Arrays:

$w(*)$, DIMENSION at least $\max(1, n)$, contains the selected eigenvalues in ascending order, stored in $w(1)$ to $w(m)$;
 $z(ldz, *)$, the second dimension of z must be at least $\max(1, m)$.

If $jobz = 'V'$, then if $info = 0$, the first m columns of z contain the orthonormal eigenvectors of the matrix T corresponding to the selected eigenvalues, with the i -th column of z holding the eigenvector associated with $w(i)$.

If $jobz = 'N'$, then z is not referenced. Note that you must ensure that at least $\max(1, m)$ columns are supplied in the array z ; if $range = 'V'$, the exact value of m is not known in advance and an upper bound must be used.

isuppz

INTEGER.

Array, DIMENSION at least $2 * \max(1, m)$.

The support of the eigenvectors in z , i.e., the indices indicating the nonzero elements in z . The i -th eigenvector is nonzero only in elements $isuppz(2i-1)$ through $isuppz(2i)$. Referenced only if eigenvectors are needed ($jobz = 'V'$) and all eigenvalues are needed, that is, $range = 'A'$ or $range = 'I'$ and $il = 1$ and $iu = n$.

work(1)

On exit, if $info = 0$, then *work(1)* returns the required minimal size of *lwork*.

iwork(1)

On exit, if $info = 0$, then *iwork(1)* returns the required minimal size of *liwork*.

info

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, an internal error has occurred.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `syevr` interface are the following:

a Holds the matrix *A* of size (n, n) .

<i>w</i>	Holds the vector of length (<i>n</i>).
<i>z</i>	Holds the matrix <i>Z</i> of size (<i>n</i> , <i>n</i>), where the values <i>n</i> and <i>m</i> are significant.
<i>isuppz</i>	Holds the vector of length ($2*m$), where the values ($2*m$) are significant.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>vl</i>	Default value for this element is $vl = -HUGE(vl)$.
<i>vu</i>	Default value for this element is $vu = HUGE(vl)$.
<i>il</i>	Default value for this argument is $il = 1$.
<i>iu</i>	Default value for this argument is $iu = n$.
<i>abstol</i>	Default value for this element is $abstol = 0.0\_WP$.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted Note that there will be an error condition if <i>isuppz</i> is present and <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: <i>range</i> = 'V', if one of or both <i>vl</i> and <i>vu</i> are present, <i>range</i> = 'I', if one of or both <i>il</i> and <i>iu</i> are present, <i>range</i> = 'A', if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present, Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

For optimum performance use $lwork \geq (nb+6)*n$, where *nb* is the maximum of the blocksize for ?sytrd and ?ormtr returned by ilaenv.

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set $lwork = -1$ ($liwork = -1$).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If $lwork = -1$ ($liwork = -1$), then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if `lwork` (`liwork`) is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Normal execution of `?stegr` may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the IEEE standard default manner.

?heevr

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix using the Relatively Robust Representations.

Syntax

FORTRAN 77:

```
call cheevr(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z,
ldz, isuppz, work, lwork, rwork, lrwork, iwork, liwork, info)

call zheevr(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z,
ldz, isuppz, work, lwork, rwork, lrwork, iwork, liwork, info)
```

Fortran 95:

```
call heevr(a, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,isuppz] [,abstol]
[,info])
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

The routine first reduces the matrix A to tridiagonal form T with a call to `?hetrd`. Then, whenever possible, `?heevr` calls `?stegr` to compute the eigenspectrum using Relatively Robust Representations. `?stegr` computes eigenvalues by the *dqds* algorithm, while orthogonal eigenvectors are computed from various “good” L^*D*L^T representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For each unreduced block (submatrix) of T :

- a.** Compute $T - \sigma^* I = L^* D^* L^T$, so that L and D define all the wanted eigenvalues to high relative accuracy. This means that small relative changes in the entries of D and L cause only small relative changes in the eigenvalues and eigenvectors. The standard (unfactored) representation of the tridiagonal matrix T does not have this property in general.
- b.** Compute the eigenvalues to suitable accuracy. If the eigenvectors are desired, the algorithm attains full accuracy of the computed eigenvalues only right before the corresponding vectors have to be computed, see Steps c) and d).
- c.** For each cluster of close eigenvalues, select a new shift close to the cluster, find a new factorization, and refine the shifted eigenvalues to suitable accuracy.
- d.** For each eigenvalue with a large enough relative separation, compute the corresponding eigenvector by forming a rank revealing twisted factorization. Go back to Step c) for any clusters that remain.

The desired accuracy of the output can be specified by the input parameter *abstol*.

The routine `?heevr` calls `?stemr` when the full spectrum is requested on machines which conform to the IEEE-754 floating point standard, or `?stebz` and `?stein` on non-IEEE machines and when partial spectrum requests are made.

Note that the routine `?heevr` is preferable for most cases of complex Hermitian eigenvalue problems as its underlying algorithm is fast and uses less workspace.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues $\lambda(i)$ in the half-open interval: $v_l < \lambda(i) \leq v_u$. If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i>. For <i>range</i> = 'V' or 'I', <code>sstebz/dstebz</code> and <code>cstein/zstein</code> are called.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', *a* stores the upper triangular part of *A*.
 If *uplo* = 'L', *a* stores the lower triangular part of *A*.

n
 INTEGER. The order of the matrix *A* ($n \geq 0$).

a, work
 COMPLEX for *cheevr*
 DOUBLE COMPLEX for *zheevr*.
 Arrays:
a(lda,)* is an array containing either upper or lower triangular part of the Hermitian matrix *A*, as specified by *uplo*.
 The second dimension of *a* must be at least $\max(1, n)$.
work is a workspace array, its dimension $\max(1, lwork)$.

lda
 INTEGER. The first dimension of the array *a*.
 Must be at least $\max(1, n)$.

vl, vu
 REAL for *cheevr*
 DOUBLE PRECISION for *zheevr*.
 If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.
 Constraint: $vl < vu$.
 If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, iu
 INTEGER.
 If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.
 Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$.
 If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol
 REAL for *cheevr*
 DOUBLE PRECISION for *zheevr*.
 The absolute error tolerance to which each eigenvalue/eigenvector is required.
 If *jobz* = 'V', the eigenvalues and eigenvectors output have residual norms bounded by *abstol*, and the dot products between different eigenvectors are bounded by *abstol*.

If $abstol < n * \epsilon * |T|$, then $n * \epsilon * |T|$ will be used in its place, where ϵ is the machine precision, and $|T|$ is the 1-norm of the matrix T . The eigenvalues are computed to an accuracy of $\epsilon * |T|$ irrespective of $abstol$. If high relative accuracy is important, set $abstol$ to `?lamch('S')`.

ldz INTEGER. The leading dimension of the output array *z*.
Constraints:

$ldz \geq 1$ if *jobz* = 'N';
 $ldz \geq \max(1, n)$ if *jobz* = 'V'.

lwork INTEGER.
The dimension of the array *work*.

Constraint: $lwork \geq \max(1, 2n)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for the suggested value of *lwork*.

rwork REAL for cheevr
DOUBLE PRECISION for zheevr.
Workspace array, DIMENSION $\max(1, lwork)$.

lrwork INTEGER.
The dimension of the array *rwork*;
 $lwork \geq \max(1, 24n)$.

If $lrwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#).

iwork INTEGER. Workspace array, its dimension $\max(1, liwork)$.

liwork INTEGER.
The dimension of the array *iwork*,
 $lwork \geq \max(1, 10n)$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#).

Output Parameters

<i>a</i>	On exit, the lower triangle (if <i>uplo</i> = 'L') or the upper triangle (if <i>uplo</i> = 'U') of <i>A</i> , including the diagonal, is overwritten.
<i>m</i>	INTEGER. The total number of eigenvalues found, $0 \leq m \leq n$. If <i>range</i> = 'A', $m = n$, and if <i>range</i> = 'I', $m = iu-il+1$.
<i>w</i>	REAL for <i>cheevr</i> DOUBLE PRECISION for <i>zheevr</i> . Array, DIMENSION at least $\max(1, n)$, contains the selected eigenvalues in ascending order, stored in <i>w</i> (1) to <i>w</i> (<i>m</i>).
<i>z</i>	COMPLEX for <i>cheevr</i> DOUBLE COMPLEX for <i>zheevr</i> . Array <i>z</i> (<i>ldz</i> , *); the second dimension of <i>z</i> must be at least $\max(1, m)$. If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix <i>T</i> corresponding to the selected eigenvalues, with the <i>i</i> -th column of <i>z</i> holding the eigenvector associated with <i>w</i> (<i>i</i>). If <i>jobz</i> = 'N', then <i>z</i> is not referenced. Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array <i>z</i> ; if <i>range</i> = 'V', the exact value of <i>m</i> is not known in advance and an upper bound must be used.
<i>isuppz</i>	INTEGER. Array, DIMENSION at least $2 * \max(1, m)$. The support of the eigenvectors in <i>z</i> , i.e., the indices indicating the nonzero elements in <i>z</i> . The <i>i</i> -th eigenvector is nonzero only in elements <i>isuppz</i> (2 <i>i</i> -1) through

	<i>isuppz</i> (2 <i>i</i>). Referenced only if eigenvectors are needed (<i>jobz</i> = 'V') and all eigenvalues are needed, that is, <i>range</i> = 'A' or <i>range</i> = 'I' and <i>il</i> = 1 and <i>iu</i> = <i>n</i> .
<i>work</i> (1)	On exit, if <i>info</i> = 0, then <i>work</i> (1) returns the required minimal size of <i>lwork</i> .
<i>rwork</i> (1)	On exit, if <i>info</i> = 0, then <i>rwork</i> (1) returns the required minimal size of <i>lrwork</i> .
<i>iwork</i> (1)	On exit, if <i>info</i> = 0, then <i>iwork</i> (1) returns the required minimal size of <i>liwork</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , an internal error has occurred.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `heevr` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>w</i>	Holds the vector of length (<i>n</i>).
<i>z</i>	Holds the matrix <i>Z</i> of size (<i>n</i> , <i>n</i>), where the values <i>n</i> and <i>m</i> are significant.
<i>isuppz</i>	Holds the vector of length (2* <i>n</i>), where the values (2* <i>m</i>) are significant.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>vl</i>	Default value for this element is <i>vl</i> = -HUGE(<i>vl</i>).
<i>vu</i>	Default value for this element is <i>vu</i> = HUGE(<i>vl</i>).
<i>il</i>	Default value for this argument is <i>il</i> = 1.
<i>iu</i>	Default value for this argument is <i>iu</i> = <i>n</i> .
<i>abstol</i>	Default value for this element is <i>abstol</i> = 0.0_WP.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted. Note that there will be an error condition if <i>isuppz</i> is present and <i>z</i> is omitted.

range

Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows:
range = 'V', if one of or both *vl* and *vu* are present, *range* = 'I',
 if one of or both *il* and *iu* are present, *range* = 'A', if none of *vl*,
vu, *il*, *iu* is present, Note that there will be an error condition if one
 of or both *vl* and *vu* are present and at the same time one of or both
il and *iu* are present.

Application Notes

For optimum performance use $lwork \geq (nb+1)*n$, where *nb* is the maximum of the blocksize for ?hetrd and ?unmtr returned by [ilaenv](#).

If you are in doubt how much workspace to supply, use a generous value of *lwork* (or *lrwork*, or *liwork*) for the first run or set *lwork* = -1 (*lrwork* = -1, *liwork* = -1).

If you choose the first option and set any of admissible *lwork* (or *lrwork*, *liwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *rwork*, *iwork*) on exit. Use this value (*work*(1), *rwork*(1), *iwork*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *rwork*, *iwork*). This operation is called a workspace query.

Note that if you set *lwork* (*lrwork*, *liwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Normal execution of ?stegr may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the IEEE standard default manner.

?spev

Computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix in packed storage.

Syntax

FORTRAN 77:

```
call sspev(jobz, uplo, n, ap, w, z, ldz, work, info)
call dspev(jobz, uplo, n, ap, w, z, ldz, work, info)
```

Fortran 95:

```
call spev(a, w [,uplo] [,z] [,info])
```

Description

This routine computes all the eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A* in packed storage.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>job</i> = 'N', then only eigenvalues are computed. If <i>job</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ap</i> stores the packed upper triangular part of <i>A</i> . If <i>uplo</i> = 'L', <i>ap</i> stores the packed lower triangular part of <i>A</i> .
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).
<i>ap, work</i>	REAL for <i>sspev</i> DOUBLE PRECISION for <i>dspev</i> Arrays: <i>ap</i> (*) contains the packed upper or lower triangle of symmetric matrix <i>A</i> , as specified by <i>uplo</i> . The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$.

work (*) is a workspace array, DIMENSION at least $\max(1, 3n)$.
ldz INTEGER. The leading dimension of the output array *z*.
 Constraints:
 if *jobz* = 'N', then $ldz \geq 1$;
 if *jobz* = 'V', then $ldz \geq \max(1, n)$.

Output Parameters

w, z REAL for *sspev*
 DOUBLE PRECISION for *dspev*
 Arrays:
w(*), DIMENSION at least $\max(1, n)$.
 If *info* = 0, *w* contains the eigenvalues of the matrix *A* in ascending order.
z(*ldz*, *). The second dimension of *z* must be at least $\max(1, n)$.
 If *jobz* = 'V', then if *info* = 0, *z* contains the orthonormal eigenvectors of the matrix *A*, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).
 If *jobz* = 'N', then *z* is not referenced.

ap On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of *A*.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.
 If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *spev* interface are the following:

<i>a</i>	Stands for argument <i>a_p</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(n * (n + 1) / 2)$.
<i>w</i>	Holds the vector of length (n) .
<i>z</i>	Holds the matrix <i>Z</i> of size (n, n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted.

?hpev

Computes all eigenvalues and, optionally, eigenvectors of a Hermitian matrix in packed storage.

Syntax

FORTRAN 77:

```
call chpev(jobz, uplo, n, ap, w, z, ldz, work, rwork, info)
call zhpev(jobz, uplo, n, ap, w, z, ldz, work, rwork, info)
```

Fortran 95:

```
call hpev(a, w [,uplo] [,z] [,info])
```

Description

This routine computes all the eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix *A* in packed storage.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>job</i> = 'N', then only eigenvalues are computed. If <i>job</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>a_p</i> stores the packed upper triangular part of <i>A</i> .

If `uplo = 'L'`, `ap` stores the packed lower triangular part of A .

`n` INTEGER. The order of the matrix A ($n \geq 0$).

`ap, work` COMPLEX for `chpev`
DOUBLE COMPLEX for `zhpev`.

Arrays:
`ap(*)` contains the packed upper or lower triangle of Hermitian matrix A , as specified by `uplo`.
The dimension of `ap` must be at least $\max(1, n*(n+1)/2)$.

`work` (*) is a workspace array, DIMENSION at least $\max(1, 2n-1)$.

`ldz` INTEGER. The leading dimension of the output array z .

Constraints:
if `jobz = 'N'`, then $ldz \geq 1$;
if `jobz = 'V'`, then $ldz \geq \max(1, n)$.

`rwork` REAL for `chpev`
DOUBLE PRECISION for `zhpev`.
Workspace array, DIMENSION at least $\max(1, 3n-2)$.

Output Parameters

`w` REAL for `chpev`
DOUBLE PRECISION for `zhpev`.
Array, DIMENSION at least $\max(1, n)$.
If `info = 0`, `w` contains the eigenvalues of the matrix A in ascending order.

`z` COMPLEX for `chpev`
DOUBLE COMPLEX for `zhpev`.
Array `z(ldz,*)`.
The second dimension of `z` must be at least $\max(1, n)$.
If `jobz = 'V'`, then if `info = 0`, `z` contains the orthonormal eigenvectors of the matrix A , with the i -th column of `z` holding the eigenvector associated with $w(i)$.
If `jobz = 'N'`, then `z` is not referenced.

<i>ap</i>	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of A.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value. If <i>info</i> = <i>i</i>, then the algorithm failed to converge; <i>i</i> indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hpev` interface are the following:

<i>a</i>	Stands for argument <i>ap</i> in Fortran 77 interface. Holds the array A of size $(n * (n + 1) / 2)$.
<i>w</i>	Holds the vector of length (<i>n</i>).
<i>z</i>	Holds the matrix Z of size (<i>n</i> , <i>n</i>).
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted.

?spevd

Uses divide and conquer algorithm to compute all eigenvalues and (optionally) all eigenvectors of a real symmetric matrix held in packed storage.

Syntax

FORTRAN 77:

```
call sspevd(jobz, uplo, n, ap, w, z, ldz, work, lwork, iwork, liwork, info)
call dspevd(jobz, uplo, n, ap, w, z, ldz, work, lwork, iwork, liwork, info)
```

Fortran 95:

```
call spevd(a, w [,uplo] [,z] [,info])
```

Description

This routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric matrix A (held in packed storage). In other words, it can compute the spectral factorization of A as:

$$A = Z^* \Lambda^* Z^T.$$

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i , and Z is the orthogonal matrix whose columns are the eigenvectors z_i . Thus,

$$A^* z_i = \lambda_i^* z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ap</i> stores the packed upper triangular part of A . If <i>uplo</i> = 'L', <i>ap</i> stores the packed lower triangular part of A .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>ap, work</i>	REAL for <i>sspevd</i> DOUBLE PRECISION for <i>dspevd</i> Arrays: <i>ap</i> (*) contains the packed upper or lower triangle of symmetric matrix A , as specified by <i>uplo</i> . The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$ <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.

<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i>. Constraints: if <i>jobz</i> = 'N', then $ldz \geq 1$; if <i>jobz</i> = 'V', then $ldz \geq \max(1, n)$.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i>. Constraints: if $n \leq 1$, then $lwork \geq 1$; if <i>jobz</i> = 'N' and $n > 1$, then $lwork \geq 2*n$; if <i>jobz</i> = 'V' and $n > 1$, then $lwork \geq n^2 + 6*n + 1$. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the required sizes of the <i>work</i> and <i>iwork</i> arrays, returns these values as the first entries of the <i>work</i> and <i>iwork</i> arrays, and no error message related to <i>lwork</i> or <i>liwork</i> is issued by xerbla. See <i>Application Notes</i> for details.
<i>iwork</i>	INTEGER. Workspace array, its dimension $\max(1, liwork)$.
<i>liwork</i>	INTEGER. The dimension of the array <i>iwork</i>. Constraints: if $n \leq 1$, then $liwork \geq 1$; if <i>jobz</i> = 'N' and $n > 1$, then $liwork \geq 1$; if <i>jobz</i> = 'V' and $n > 1$, then $liwork \geq 5*n+3$. If <i>liwork</i> = -1, then a workspace query is assumed; the routine only calculates the required sizes of the <i>work</i> and <i>iwork</i> arrays, returns these values as the first entries of the <i>work</i> and <i>iwork</i> arrays, and no error message related to <i>lwork</i> or <i>liwork</i> is issued by xerbla. See <i>Application Notes</i> for details.

Output Parameters

<i>w, z</i>	REAL for <code>sspevd</code> DOUBLE PRECISION for <code>dspevd</code>

Arrays:

$w(*)$, DIMENSION at least $\max(1, n)$.

If $info = 0$, contains the eigenvalues of the matrix A in ascending order. See also $info$.

$z(ldz,*)$.

The second dimension of z must be: at least 1 if $jobz = 'N'$; at least $\max(1, n)$ if $jobz = 'V'$.

If $jobz = 'V'$, then this array is overwritten by the orthogonal matrix Z which contains the eigenvectors of A .

If $jobz = 'N'$, then z is not referenced.

ap

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of A .

$work(1)$

On exit, if $info = 0$, then $work(1)$ returns the required $lwork$.

$iwork(1)$

On exit, if $info = 0$, then $iwork(1)$ returns the required $liwork$.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = i$, then the algorithm failed to converge; i indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `spevd` interface are the following:

a	Stands for argument ap in Fortran 77 interface. Holds the array A of size $(n*(n+1)/2)$.
w	Holds the vector of length (n) .
z	Holds the matrix Z of size (n, n) .
$uplo$	Must be 'U' or 'L'. The default value is 'U'.

jobz Restored based on the presence of the argument *z* as follows:
jobz = 'V', if *z* is present,
jobz = 'N', if *z* is omitted.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T+E$ such that $\|E\|_2 = O(\epsilon) * \|T\|_2$, where ϵ is the machine precision.

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = -1 (*liwork* = -1).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If *lwork* = -1 (*liwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *lwork* (*liwork*) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex analogue of this routine is [?hpevd](#).

See also [?syevd](#) for matrices held in full storage, and [?sbevd](#) for banded matrices.

?hpevd

Uses divide and conquer algorithm to compute all eigenvalues and (optionally) all eigenvectors of a complex Hermitian matrix held in packed storage.

Syntax

FORTRAN 77:

```
call chpevd(jobz, uplo, n, ap, w, z, ldz, work, lwork, rwork, lrwork, iwork,
liwork, info)

call zhpevd(jobz, uplo, n, ap, w, z, ldz, work, lwork, rwork, lrwork, iwork,
liwork, info)
```

Fortran 95:

```
call hpevd(a, w [,uplo] [,z] [,info])
```

Description

This routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian matrix A (held in packed storage). In other words, it can compute the spectral factorization of A as: $A = Z^* \Lambda Z^H$.

Here Λ is a real diagonal matrix whose diagonal elements are the eigenvalues λ_i , and Z is the (complex) unitary matrix whose columns are the eigenvectors z_i . Thus,

$$A^* z_i = \lambda_i^* z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ap</i> stores the packed upper triangular part of A . If <i>uplo</i> = 'L', <i>ap</i> stores the packed lower triangular part of A .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>ap, work</i>	COMPLEX for <i>chpevd</i> DOUBLE COMPLEX for <i>zhpevd</i> Arrays: <i>ap</i> (*) contains the packed upper or lower triangle of Hermitian matrix A , as specified by <i>uplo</i> . The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i> . Constraints:

if $jobz = 'N'$, then $ldz \geq 1$;
 if $jobz = 'V'$, then $ldz \geq \max(1, n)$.

lwork INTEGER.
 The dimension of the array *work*.
 Constraints:
 if $n \leq 1$, then $lwork \geq 1$;
 if $jobz = 'N'$ and $n > 1$, then $lwork \geq n$;
 if $jobz = 'V'$ and $n > 1$, then $lwork \geq 2*n$.
 If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

rwork REAL for *chpevd*
 DOUBLE PRECISION for *zhpevd*
 Workspace array, its dimension $\max(1, lrwork)$.

lrwork INTEGER.
 The dimension of the array *rwork*. Constraints:
 if $n \leq 1$, then $lrwork \geq 1$;
 if $jobz = 'N'$ and $n > 1$, then $lrwork \geq n$;
 if $jobz = 'V'$ and $n > 1$, then $lrwork \geq 2*n^2 + 5*n + 1$.
 If $lrwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

iwork INTEGER. Workspace array, its dimension $\max(1, liwork)$.

liwork INTEGER.
 The dimension of the array *iwork*.
 Constraints:
 if $n \leq 1$, then $liwork \geq 1$;

if $jobz = 'N'$ and $n > 1$, then $liwork \geq 1$;
 if $jobz = 'V'$ and $n > 1$, then $liwork \geq 5*n+3$.
 If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$, $rwork$ and $iwork$ arrays, returns these values as the first entries of the $work$, $rwork$ and $iwork$ arrays, and no error message related to $lwork$ or $lrwork$ or $liwork$ is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

w	REAL for <code>chpevd</code> DOUBLE PRECISION for <code>zhpevd</code> Array, DIMENSION at least $\max(1, n)$. If $info = 0$, contains the eigenvalues of the matrix A in ascending order. See also $info$.
z	COMPLEX for <code>chpevd</code> DOUBLE COMPLEX for <code>zhpevd</code> Array, DIMENSION $(ldz,*)$. The second dimension of z must be: at least 1 if $jobz = 'N'$; at least $\max(1, n)$ if $jobz = 'V'$. If $jobz = 'V'$, then this array is overwritten by the unitary matrix Z which contains the eigenvectors of A. If $jobz = 'N'$, then z is not referenced.
ap	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of A.
$work(1)$	On exit, if $info = 0$, then $work(1)$ returns the required minimal size of $lwork$.
$rwork(1)$	On exit, if $info = 0$, then $rwork(1)$ returns the required minimal size of $lrwork$.
$iwork(1)$	On exit, if $info = 0$, then $iwork(1)$ returns the required minimal size of $liwork$.
$info$	INTEGER. If $info = 0$, the execution is successful.

If $info = i$, then the algorithm failed to converge; i indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.
 If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hpevd` interface are the following:

<i>a</i>	Stands for argument <i>a_p</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(n * (n + 1) / 2)$.
<i>w</i>	Holds the vector of length (n) .
<i>z</i>	Holds the matrix <i>Z</i> of size (n, n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: $jobz = 'V'$, if <i>z</i> is present, $jobz = 'N'$, if <i>z</i> is omitted.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T + E$ such that $\|E\|_2 = O(\epsilon) * \|T\|_2$, where ϵ is the machine precision.

If you are in doubt how much workspace to supply, use a generous value of *lwork* (*liwork* or *lrwork*) for the first run or set $lwork = -1$ ($liwork = -1$, $lrwork = -1$).

If you choose the first option and set any of admissible *lwork* (*liwork* or *lrwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*) on exit. Use this value (*work*(1), *iwork*(1), *rwork*(1)) for subsequent runs.

If you set $lwork = -1$ ($liwork = -1$, $lrwork = -1$), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*). This operation is called a workspace query.

Note that if you set `lwork` (`liwork`, `lrwork`) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real analogue of this routine is [?spevd](#).

See also [?heevd](#) for matrices held in full storage, and [?hbevd](#) for banded matrices.

?spevx

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix in packed storage.

Syntax

FORTRAN 77:

```
call sspevx(jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z, ldz,
work, iwork, ifail, info)
```

```
call dspevx(jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z, ldz,
work, iwork, ifail, info)
```

Fortran 95:

```
call spevx(a, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol]
[,info])
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A* in packed storage. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>job</i> = 'N', then only eigenvalues are computed. If <i>job</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues.

	If <i>range</i> = 'V', the routine computes eigenvalues <i>lambda(i)</i> in the half-open interval: $vl < \lambda(i) \leq vu$. If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i>.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ap</i> stores the packed upper triangular part of <i>A</i>. If <i>uplo</i> = 'L', <i>ap</i> stores the packed lower triangular part of <i>A</i>.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).
<i>ap, work</i>	REAL for <i>sspevx</i> DOUBLE PRECISION for <i>dspevx</i> Arrays: <i>ap</i>(*) contains the packed upper or lower triangle of the symmetric matrix <i>A</i>, as specified by <i>uplo</i>. The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$. <i>work</i>(*) is a workspace array, DIMENSION at least $\max(1, 8n)$.
<i>vl, vu</i>	REAL for <i>sspevx</i> DOUBLE PRECISION for <i>dspevx</i> If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: $vl < vu$. If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il, iu</i>	INTEGER. If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$. If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.
<i>abstol</i>	REAL for <i>sspevx</i> DOUBLE PRECISION for <i>dspevx</i> The absolute error tolerance to which each eigenvalue is required. See <i>Application notes</i> for details on error tolerance.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i>. Constraints:

if $jobz = 'N'$, then $ldz \geq 1$;
 if $jobz = 'V'$, then $ldz \geq \max(1, n)$.
iwork INTEGER. Workspace array, DIMENSION at least $\max(1, 5n)$.

Output Parameters

ap On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of A .

m INTEGER. The total number of eigenvalues found,
 $0 \leq m \leq n$. If $range = 'A'$, $m = n$, and if $range = 'I'$,
 $m = iu-il+1$.

w, z REAL for `sspevx`
 DOUBLE PRECISION for `dspevx`
Arrays:
w(*), DIMENSION at least $\max(1, n)$.
 If $info = 0$, contains the selected eigenvalues of the matrix A in ascending order.
z(ldz ,*).
 The second dimension of z must be at least $\max(1, m)$.
 If $jobz = 'V'$, then if $info = 0$, the first m columns of z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i -th column of z holding the eigenvector associated with $w(i)$.
 If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.
 If $jobz = 'N'$, then z is not referenced.
 Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array z ; if $range = 'V'$, the exact value of m is not known in advance and an upper bound must be used.

ifail INTEGER.
 Array, DIMENSION at least $\max(1, n)$.

If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices the eigenvectors that failed to converge.

If *jobz* = 'N', then *ifail* is not referenced.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *spevx* interface are the following:

<i>a</i>	Stands for argument <i>ap</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(n * (n+1) / 2)$.
<i>w</i>	Holds the vector of length (<i>n</i>).
<i>z</i>	Holds the matrix <i>Z</i> of size (<i>n</i> , <i>n</i>), where the values <i>n</i> and <i>m</i> are significant.
<i>ifail</i>	Holds the vector of length (<i>n</i>).
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>vl</i>	Default value for this element is <i>vl</i> = -HUGE(<i>vl</i>).
<i>vu</i>	Default value for this element is <i>vu</i> = HUGE(<i>vl</i>).
<i>il</i>	Default value for this argument is <i>il</i> = 1.
<i>iu</i>	Default value for this argument is <i>iu</i> = <i>n</i> .
<i>abstol</i>	Default value for this element is <i>abstol</i> = 0.0_WP.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted Note that there will be an error condition if <i>ifail</i> is present and <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: <i>range</i> = 'V', if one of or both <i>vl</i> and <i>vu</i> are present,

`range = 'I'`, if one of or both `il` and `iu` are present,
`range = 'A'`, if none of `vl`, `vu`, `il`, `iu` is present,
 Note that there will be an error condition if one of or both `vl` and `vu`
 are present and at the same time one of or both `il` and `iu` are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon \max(|a|, |b|)$, where ϵ is the machine precision.

If `abstol` is less than or equal to zero, then $\epsilon * ||T||_1$ will be used in its place, where T is the tridiagonal matrix obtained by reducing A to tridiagonal form. Eigenvalues will be computed most accurately when `abstol` is set to twice the underflow threshold $2 * ?lamch('S')$, not zero.

If this routine returns with `info > 0`, indicating that some eigenvectors did not converge, try setting `abstol` to $2 * ?lamch('S')$.

?hpevx

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix in packed storage.

Syntax

FORTRAN 77:

```
call chpevx(jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z, ldz,
work, rwork, iwork, ifail, info)

call zhpevx(jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z, ldz,
work, rwork, iwork, ifail, info)
```

Fortran 95:

```
call hpevx(a, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol]
[,info])
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A in packed storage. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues $\lambda(i)$ in the half-open interval: $vl < \lambda(i) \leq vu$. If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i>.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ap</i> stores the packed upper triangular part of <i>A</i>. If <i>uplo</i> = 'L', <i>ap</i> stores the packed lower triangular part of <i>A</i>.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).
<i>ap, work</i>	COMPLEX for <i>chpevx</i> DOUBLE COMPLEX for <i>zhpevx</i> Arrays: <i>ap</i>(*) contains the packed upper or lower triangle of the Hermitian matrix <i>A</i>, as specified by <i>uplo</i>. The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$. <i>work</i>(*) is a workspace array, DIMENSION at least $\max(1, 2n)$.
<i>vl, vu</i>	REAL for <i>chpevx</i> DOUBLE PRECISION for <i>zhpevx</i> If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: $vl < vu$. If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il, iu</i>	INTEGER. If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.

	Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$. If $range = 'A'$ or $'V'$, il and iu are not referenced.
<i>abstol</i>	REAL for <code>chpevx</code> DOUBLE PRECISION for <code>zhpevx</code> The absolute error tolerance to which each eigenvalue is required. See <i>Application notes</i> for details on error tolerance.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i>. Constraints: if $jobz = 'N'$, then $ldz \geq 1$; if $jobz = 'V'$, then $ldz \geq \max(1, n)$.
<i>rwork</i>	REAL for <code>chpevx</code> DOUBLE PRECISION for <code>zhpevx</code> Workspace array, DIMENSION at least $\max(1, 7n)$.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, 5n)$.

Output Parameters

<i>ap</i>	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of <i>A</i>.
<i>m</i>	INTEGER. The total number of eigenvalues found, $0 \leq m \leq n$. If $range = 'A'$, $m = n$, and if $range = 'I'$, $m = iu-il+1$.
<i>w</i>	REAL for <code>chpevx</code> DOUBLE PRECISION for <code>zhpevx</code> Array, DIMENSION at least $\max(1, n)$. If $info = 0$, contains the selected eigenvalues of the matrix <i>A</i> in ascending order.
<i>z</i>	COMPLEX for <code>chpevx</code> DOUBLE COMPLEX for <code>zhpevx</code> Array $z(ldz,*)$. The second dimension of <i>z</i> must be at least $\max(1, m)$.

If `jobz = 'V'`, then if `info = 0`, the first m columns of z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i -th column of z holding the eigenvector associated with $w(i)$. If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in `ifail`.

If `jobz = 'N'`, then z is not referenced.

Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array z ; if `range = 'V'`, the exact value of m is not known in advance and an upper bound must be used.

`ifail`

INTEGER.

Array, DIMENSION at least $\max(1, n)$.

If `jobz = 'V'`, then if `info = 0`, the first m elements of `ifail` are zero; if `info > 0`, the `ifail` contains the indices the eigenvectors that failed to converge.

If `jobz = 'N'`, then `ifail` is not referenced.

`info`

INTEGER.

If `info = 0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

If `info = i`, then i eigenvectors failed to converge; their indices are stored in the array `ifail`.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hpevx` interface are the following:

<code>a</code>	Stands for argument <code>ap</code> in Fortran 77 interface. Holds the array A of size $(n * (n+1) / 2)$.
<code>w</code>	Holds the vector of length (n) .
<code>z</code>	Holds the matrix Z of size (n, n) , where the values n and m are significant.
<code>ifail</code>	Holds the vector of length (n) .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.

<i>vl</i>	Default value for this element is $vl = -\text{HUGE}(vl)$.
<i>vu</i>	Default value for this element is $vu = \text{HUGE}(vl)$.
<i>il</i>	Default value for this argument is $il = 1$.
<i>iu</i>	Default value for this argument is $iu = n$.
<i>abstol</i>	Default value for this element is $abstol = 0.0\_WP$.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: $jobz = 'V'$, if <i>z</i> is present, $jobz = 'N'$, if <i>z</i> is omitted Note that there will be an error condition if <i>ifail</i> is present and <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: $range = 'V'$, if one of or both <i>vl</i> and <i>vu</i> are present, $range = 'I'$, if one of or both <i>il</i> and <i>iu</i> are present, $range = 'A'$, if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present, Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon \cdot \max(|a|, |b|)$, where ϵ is the machine precision.

If *abstol* is less than or equal to zero, then $\epsilon \cdot ||T||_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold $2 \cdot \text{lamch}('S')$, not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to $2 \cdot \text{lamch}('S')$.

?sbev

Computes all eigenvalues and, optionally, eigenvectors of a real symmetric band matrix.

Syntax

FORTRAN 77:

```
call ssbev(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, info)
call dsbev(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, info)
```

Fortran 95:

```
call sbbev(a, w [,uplo] [,z] [,info])
```

Description

This routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric band matrix *A*.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ab</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = 'L', <i>ab</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).
<i>kd</i>	INTEGER. The number of super- or sub-diagonals in <i>A</i> ($kd \geq 0$).
<i>ab, work</i>	REAL for ssbev DOUBLE PRECISION for dsbev. Arrays: <i>ab</i> (<i>ldab</i> ,*) is an array containing either upper or lower triangular part of the symmetric matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format. The second dimension of <i>ab</i> must be at least $\max(1, n)$.

work (*) is a workspace array.
 The dimension of *work* must be at least $\max(1, 3n-2)$.

ldab INTEGER. The leading dimension of *ab*; must be at least $kd + 1$.

ldz INTEGER. The leading dimension of the output array *z*.
 Constraints:
 if *jobz* = 'N', then $ldz \geq 1$;
 if *jobz* = 'V', then $ldz \geq \max(1, n)$.

Output Parameters

w, *z* REAL for ssbev
 DOUBLE PRECISION for dsbev
 Arrays:
w (*), DIMENSION at least $\max(1, n)$.
 If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order.
z(*ldz*,*).
 The second dimension of *z* must be at least $\max(1, n)$.
 If *jobz* = 'V', then if *info* = 0, *z* contains the orthonormal eigenvectors of the matrix *A*, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).
 If *jobz* = 'N', then *z* is not referenced.

ab On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.
 If *uplo* = 'U', the first superdiagonal and the diagonal of the tridiagonal matrix *T* are returned in rows *kd* and *kd*+1 of *ab*, and if *uplo* = 'L', the diagonal and first subdiagonal of *T* are returned in the first two rows of *ab*.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.
 If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sbevd` interface are the following:

<i>a</i>	Stands for argument <i>ab</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(kd+1, n)$.
<i>w</i>	Holds the vector of length (n) .
<i>z</i>	Holds the matrix <i>Z</i> of size (n, n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: $jobz = 'V'$, if <i>z</i> is present, $jobz = 'N'$, if <i>z</i> is omitted.

?hbevd

Computes all eigenvalues and, optionally, eigenvectors of a Hermitian band matrix.

Syntax

FORTRAN 77:

```
call chbevd(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, rwork, info)
call zhbevd(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, rwork, info)
```

Fortran 95:

```
call hbevd(a, w [,uplo] [,z] [,info])
```

Description

This routine computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix *A*.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed.

If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

uplo CHARACTER*1. Must be 'U' or 'L'.
 If *uplo* = 'U', *ab* stores the upper triangular part of *A*.
 If *uplo* = 'L', *ab* stores the lower triangular part of *A*.

n INTEGER. The order of the matrix *A* ($n \geq 0$).

kd INTEGER. The number of super- or sub-diagonals in *A* ($kd \geq 0$).

ab, work COMPLEX for chbev
 DOUBLE COMPLEX for zhbev.
Arrays:
ab (*ldab*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A* (as specified by *uplo*) in band storage format.
 The second dimension of *ab* must be at least $\max(1, n)$.
work (*) is a workspace array.
 The dimension of *work* must be at least $\max(1, n)$.

ldab INTEGER. The leading dimension of *ab*; must be at least *kd* + 1.

ldz INTEGER. The leading dimension of the output array *z*.
Constraints:
 if *jobz* = 'N', then $ldz \geq 1$;
 if *jobz* = 'V', then $ldz \geq \max(1, n)$.

rwork REAL for chbev
 DOUBLE PRECISION for zhbev
 Workspace array, DIMENSION at least $\max(1, 3n-2)$.

Output Parameters

w REAL for chbev
 DOUBLE PRECISION for zhbev
 Array, DIMENSION at least $\max(1, n)$.
 If *info* = 0, contains the eigenvalues in ascending order.

z COMPLEX for chbev
 DOUBLE COMPLEX for zhbev.
 Array *z*(*ldz*,*).

	The second dimension of z must be at least $\max(1, n)$. If $jobz = 'V'$, then if $info = 0$, z contains the orthonormal eigenvectors of the matrix A, with the i-th column of z holding the eigenvector associated with $w(i)$. If $jobz = 'N'$, then z is not referenced.
ab	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. If $uplo = 'U'$, the first superdiagonal and the diagonal of the tridiagonal matrix T are returned in rows kd and $kd+1$ of ab, and if $uplo = 'L'$, the diagonal and first subdiagonal of T are returned in the first two rows of ab.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the ith parameter had an illegal value. If $info = i$, then the algorithm failed to converge; i indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hbev` interface are the following:

a	Stands for argument ab in Fortran 77 interface. Holds the array A of size $(kd+1, n)$.
w	Holds the vector of length (n) .
z	Holds the matrix Z of size (n, n) .
$uplo$	Must be <code>'U'</code> or <code>'L'</code> . The default value is <code>'U'</code> .
$jobz$	Restored based on the presence of the argument z as follows: $jobz = 'V'$, if z is present, $jobz = 'N'$, if z is omitted.

?sbevd

Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric band matrix using divide and conquer algorithm.

Syntax

FORTRAN 77:

```
call ssbevd(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, lwork, iwork, liwork, info)
```

```
call dsbevd(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, lwork, iwork, liwork, info)
```

Fortran 95:

```
call sbevd(a, w [,uplo] [,z] [,info])
```

Description

This routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric band matrix A . In other words, it can compute the spectral factorization of A as:

$$A = Z \Lambda Z^T$$

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i , and Z is the orthogonal matrix whose columns are the eigenvectors z_i . Thus,

$$A z_i = \lambda_i z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ab</i> stores the upper triangular part of A .

	If <i>uplo</i> = 'L', <i>ab</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).
<i>kd</i>	INTEGER. The number of super- or sub-diagonals in <i>A</i> ($kd \geq 0$).
<i>ab, work</i>	REAL for <i>ssbevd</i> DOUBLE PRECISION for <i>dsbevd</i> . Arrays: <i>ab</i> (<i>ldab</i> ,*) is an array containing either upper or lower triangular part of the symmetric matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format. The second dimension of <i>ab</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>ldab</i>	INTEGER. The leading dimension of <i>ab</i> ; must be at least $kd+1$.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i> . Constraints: if <i>jobz</i> = 'N', then $ldz \geq 1$; if <i>jobz</i> = 'V', then $ldz \geq \max(1, n)$.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . Constraints: if $n \leq 1$, then $lwork \geq 1$; if <i>jobz</i> = 'N' and $n > 1$, then $lwork \geq 2n$; if <i>jobz</i> = 'V' and $n > 1$, then $lwork \geq 2*n^2 + 5*n + 1$. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> and <i>iwork</i> arrays, returns these values as the first entries of the <i>work</i> and <i>iwork</i> arrays, and no error message related to <i>lwork</i> or <i>liwork</i> is issued by xerbla . See <i>Application Notes</i> for details.
<i>iwork</i>	INTEGER. Workspace array, its dimension $\max(1, liwork)$.
<i>liwork</i>	INTEGER.

The dimension of the array *iwork*. Constraints: if $n \leq 1$, then $liwork < 1$; if $job = 'N'$ and $n > 1$, then $liwork < 1$; if $job = 'V'$ and $n > 1$, then $liwork < 5*n+3$. If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

<i>w</i> , <i>z</i>	REAL for ssbevd DOUBLE PRECISION for dsbevd Arrays: <i>w</i>(*), DIMENSION at least $\max(1, n)$. If <i>info</i> = 0, contains the eigenvalues of the matrix <i>A</i> in ascending order. See also <i>info</i>. <i>z</i>(<i>ldz</i>,*). The second dimension of <i>z</i> must be: at least 1 if $job = 'N'$; at least $\max(1, n)$ if $job = 'V'$. If $job = 'V'$, then this array is overwritten by the orthogonal matrix <i>Z</i> which contains the eigenvectors of <i>A</i>. The <i>i</i>-th column of <i>Z</i> contains the eigenvector which corresponds to the eigenvalue <i>w</i>(<i>i</i>). If $job = 'N'$, then <i>z</i> is not referenced.
<i>ab</i>	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.
<i>work</i> (1)	On exit, if <i>lwork</i> > 0, then <i>work</i> (1) returns the required minimal size of <i>lwork</i> .
<i>iwork</i> (1)	On exit, if <i>liwork</i> > 0, then <i>iwork</i> (1) returns the required minimal size of <i>liwork</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful.

If $info = i$, then the algorithm failed to converge; i indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.
 If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sbevd` interface are the following:

a	Stands for argument ab in Fortran 77 interface. Holds the array A of size $(kd+1, n)$.
w	Holds the vector of length (n) .
z	Holds the matrix Z of size (n, n) .
$uplo$	Must be 'U' or 'L'. The default value is 'U'.
$jobz$	Restored based on the presence of the argument z as follows: $jobz = 'V'$, if z is present, $jobz = 'N'$, if z is omitted.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T+E$ such that

$\|E\|_2 = O(\epsilon) * \|T\|_2$, where ϵ is the machine precision.

If it is not clear how much workspace to supply, use a generous value of $lwork$ (or $liwork$) for the first run or set $lwork = -1$ ($liwork = -1$).

If any of admissible $lwork$ (or $liwork$) has any of admissible sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array ($work, iwork$) on exit. Use this value ($work(1), iwork(1)$) for subsequent runs.

If $lwork = -1$ ($liwork = -1$), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work, iwork$). This operation is called a workspace query.

Note that if *work* (*liwork*) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex analogue of this routine is [?hbevd](#).

See also [?syevd](#) for matrices held in full storage, and [?spevd](#) for matrices held in packed storage.

?hbevd

Computes all eigenvalues and (optionally) all eigenvectors of a complex Hermitian band matrix using divide and conquer algorithm.

Syntax

FORTRAN 77:

```
call chbevd(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, lwork, rwork, lrwork,
iwork, liwork, info)

call zhbevd(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, lwork, rwork, lrwork,
iwork, liwork, info)
```

Fortran 95:

```
call hbevd(a, w [,uplo] [,z] [,info])
```

Description

This routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian band matrix *A*. In other words, it can compute the spectral factorization of *A* as: $A = Z \Lambda Z^H$.

Here Λ is a real diagonal matrix whose diagonal elements are the eigenvalues λ_i , and *Z* is the (complex) unitary matrix whose columns are the eigenvectors z_i . Thus,

$$A z_i = \lambda_i z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the *QL* or *QR* algorithm.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ab</i> stores the upper triangular part of <i>A</i>. If <i>uplo</i> = 'L', <i>ab</i> stores the lower triangular part of <i>A</i>.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).
<i>kd</i>	INTEGER. The number of super- or sub-diagonals in <i>A</i> ($kd \geq 0$).
<i>ab, work</i>	COMPLEX for <i>chbevd</i> DOUBLE COMPLEX for <i>zhbevd</i>. Arrays: <i>ab</i> (<i>ldab</i>,*) is an array containing either upper or lower triangular part of the Hermitian matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format. The second dimension of <i>ab</i> must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array, its dimension $\max(1, lwork)$.
<i>ldab</i>	INTEGER. The leading dimension of <i>ab</i>; must be at least $kd+1$.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i>. Constraints: if <i>jobz</i> = 'N', then $ldz \geq 1$; if <i>jobz</i> = 'V', then $ldz \geq \max(1, n)$.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i>. Constraints: if $n \leq 1$, then $lwork \geq 1$; if <i>jobz</i> = 'N' and $n > 1$, then $lwork \geq n$; if <i>jobz</i> = 'V' and $n > 1$, then $lwork \geq 2*n^2$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

rwork

REAL for chbevd
DOUBLE PRECISION for zhbevd
Workspace array, DIMENSION at least *lrwork*.

lrwork

INTEGER.
The dimension of the array *rwork*.
Constraints:

if $n \leq 1$, then $lrwork \geq 1$;
if $jobz = 'N'$ and $n > 1$, then $lrwork \geq n$;
if $jobz = 'V'$ and $n > 1$, then $lrwork \geq 2*n^2 + 5*n + 1$.

If $lrwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

iwork

INTEGER. Workspace array, DIMENSION $\max(1, liwork)$.

liwork

INTEGER.
The dimension of the array *iwork*.
Constraints:

if $jobz = 'N'$ or $n \leq 1$, then $liwork \geq 1$;
if $jobz = 'V'$ and $n > 1$, then $liwork \geq 5*n+3$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

<i>w</i>	REAL for <code>chbevd</code> DOUBLE PRECISION for <code>zhbevd</code> Array, DIMENSION at least $\max(1, n)$. If <i>info</i> = 0, contains the eigenvalues of the matrix <i>A</i> in ascending order. See also <i>info</i>.
<i>z</i>	COMPLEX for <code>chbevd</code> DOUBLE COMPLEX for <code>zhbevd</code> Array, DIMENSION (<i>ldz</i>, *). The second dimension of <i>z</i> must be: at least 1 if <i>jobz</i> = 'N'; at least $\max(1, n)$ if <i>jobz</i> = 'V'. If <i>jobz</i> = 'V', then this array is overwritten by the unitary matrix <i>Z</i> which contains the eigenvectors of <i>A</i>. The <i>i</i>-th column of <i>Z</i> contains the eigenvector which corresponds to the eigenvalue <i>w</i>(<i>i</i>). If <i>jobz</i> = 'N', then <i>z</i> is not referenced.
<i>ab</i>	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.
<i>work</i> (1)	On exit, if <i>lwork</i> > 0, then the real part of <i>work</i>(1) returns the required minimal size of <i>lwork</i>.
<i>rwork</i> (1)	On exit, if <i>lrwork</i> > 0, then <i>rwork</i>(1) returns the required minimal size of <i>lrwork</i>.
<i>iwork</i> (1)	On exit, if <i>liwork</i> > 0, then <i>iwork</i>(1) returns the required minimal size of <i>liwork</i>.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>i</i>, then the algorithm failed to converge; <i>i</i> indicates the number of elements of an intermediate tridiagonal form which did not converge to zero. If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hbevd` interface are the following:

<i>a</i>	Stands for argument <i>ab</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(kd+1, n)$.
<i>w</i>	Holds the vector of length (n) .
<i>z</i>	Holds the matrix <i>Z</i> of size (n, n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T + E$ such that $\|E\|_2 = O(\epsilon) \|T\|_2$, where ϵ is the machine precision.

If you are in doubt how much workspace to supply, use a generous value of *lwork* (*liwork* or *lrwork*) for the first run or set *lwork* = -1 (*liwork* = -1, *lrwork* = -1).

If you choose the first option and set any of admissible *lwork* (*liwork* or *lrwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*) on exit. Use this value (*work*(1), *iwork*(1), *rwork*(1)) for subsequent runs.

If you set *lwork* = -1 (*liwork* = -1, *lrwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*, *lrwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real analogue of this routine is [?sbevd](#).

See also [?heevd](#) for matrices held in full storage, and [?hpevd](#) for matrices held in packed storage.

?sbevx

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric band matrix.

Syntax

FORTRAN 77:

```
call ssbevx(jobz, range, uplo, n, kd, ab, ldab, q, ldq, vl, vu, il, iu, abstol,
m, w, z, ldz, work, iwork, ifail, info)

call dsbevx(jobz, range, uplo, n, kd, ab, ldab, q, ldq, vl, vu, il, iu, abstol,
m, w, z, ldz, work, iwork, ifail, info)
```

Fortran 95:

```
call sbevx(a, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,q]
[,abstol] [,info])
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric band matrix *A*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues $\lambda(i)$ in the half-open interval: $vl < \lambda(i) \leq vu$. If <i>range</i> = 'I', the routine computes eigenvalues in range <i>il</i> to <i>iu</i> .
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ab</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = 'L', <i>ab</i> stores the lower triangular part of <i>A</i> .

<i>n</i>	INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).
<i>kd</i>	INTEGER. The number of super- or sub-diagonals in <i>A</i> ($kd \geq 0$).
<i>ab, work</i>	REAL for <i>ssbev</i> DOUBLE PRECISION for <i>dsbev</i> . Arrays: <i>ab</i> (<i>ldab</i> ,*) is an array containing either upper or lower triangular part of the symmetric matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format. The second dimension of <i>ab</i> must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, 7n)$.
<i>ldab</i>	INTEGER. The leading dimension of <i>ab</i> ; must be at least $kd + 1$.
<i>vl, vu</i>	REAL for <i>ssbev</i> DOUBLE PRECISION for <i>dsbev</i> . If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: $vl < vu$. If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il, iu</i>	INTEGER. If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$. If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.
<i>abstol</i>	REAL for <i>chpev</i> DOUBLE PRECISION for <i>zhpev</i> The absolute error tolerance to which each eigenvalue is required. See <i>Application notes</i> for details on error tolerance.
<i>ldq, ldz</i>	INTEGER. The leading dimensions of the output arrays <i>q</i> and <i>z</i> , respectively. Constraints: $ldq \geq 1, ldz \geq 1$;

If $jobz = 'V'$, then $ldq \geq \max(1, n)$ and $ldz \geq \max(1, n)$.

iwork

INTEGER. Workspace array, DIMENSION at least $\max(1, 5n)$.

Output Parameters

q

REAL for ssbevz DOUBLE PRECISION for dsbevz.

Array, DIMENSION (ldz, n).

If $jobz = 'V'$, the n -by- n orthogonal matrix is used in the reduction to tridiagonal form.

If $jobz = 'N'$, the array *q* is not referenced.

m

INTEGER. The total number of eigenvalues found, $0 \leq m \leq n$.

If $range = 'A'$, $m = n$, and if $range = 'I'$, $m = iu-il+1$.

w, z

REAL for ssbevz

DOUBLE PRECISION for dsbevz

Arrays:

$w(*)$, DIMENSION at least $\max(1, n)$. The first m elements of *w* contain the selected eigenvalues of the matrix *A* in ascending order.

$z(ldz,*)$.

The second dimension of *z* must be at least $\max(1, m)$.

If $jobz = 'V'$, then if $info = 0$, the first m columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the i -th column of *z* holding the eigenvector associated with $w(i)$. If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

If $jobz = 'N'$, then *z* is not referenced.

Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array *z*; if $range = 'V'$, the exact value of m is not known in advance and an upper bound must be used.

ab

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.

If $uplo = 'U'$, the first superdiagonal and the diagonal of the tridiagonal matrix T are returned in rows kd and $kd+1$ of ab , and if $uplo = 'L'$, the diagonal and first subdiagonal of T are returned in the first two rows of ab .

ifail

INTEGER.

Array, DIMENSION at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, the first m elements of *ifail* are zero; if $info > 0$, the *ifail* contains the indices the eigenvectors that failed to converge.

If $jobz = 'N'$, then *ifail* is not referenced.

info

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, then i eigenvectors failed to converge; their indices are stored in the array *ifail*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sbevx` interface are the following:

<i>a</i>	Stands for argument <i>ab</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(kd+1, n)$.
<i>w</i>	Holds the vector of length (n) .
<i>z</i>	Holds the matrix <i>Z</i> of size (n, n) , where the values n and m are significant.
<i>ifail</i>	Holds the vector of length (n) .
<i>q</i>	Holds the matrix <i>Q</i> of size (n, n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>vl</i>	Default value for this element is $vl = -HUGE(vl)$.
<i>vu</i>	Default value for this element is $vu = HUGE(vl)$.
<i>il</i>	Default value for this argument is $il = 1$.
<i>iu</i>	Default value for this argument is $iu = n$.
<i>abstol</i>	Default value for this element is $abstol = 0.0\_WP$.

<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted Note that there will be an error condition if either <i>ifail</i> or <i>q</i> is present and <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i>, <i>vu</i>, <i>il</i>, <i>iu</i> as follows: <i>range</i> = 'V', if one of or both <i>vl</i> and <i>vu</i> are present, <i>range</i> = 'I', if one of or both <i>il</i> and <i>iu</i> are present, <i>range</i> = 'A', if none of <i>vl</i>, <i>vu</i>, <i>il</i>, <i>iu</i> is present, Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon \max(|a|, |b|)$, where ϵ is the machine precision.

If *abstol* is less than or equal to zero, then $\epsilon * ||T||_1$ is used as tolerance, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold $2 * \text{?lamch}('S')$, not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to $2 * \text{?lamch}('S')$.

?hbevx

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian band matrix.

Syntax

FORTRAN 77:

```
call chbevx(jobz, range, uplo, n, kd, ab, ldab, q, ldq, vl, vu, il, iu, abstol,
m, w, z, ldz, work, rwork, iwork, ifail, info)

call zhbevx(jobz, range, uplo, n, kd, ab, ldab, q, ldq, vl, vu, il, iu, abstol,
m, w, z, ldz, work, rwork, iwork, ifail, info)
```

Fortran 95:

```
call hbev(x, a, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,q]
[,abstol] [,info])
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix A . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

jobz CHARACTER*1. Must be 'N' or 'V'.
 If *jobz* = 'N', then only eigenvalues are computed.
 If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

range CHARACTER*1. Must be 'A' or 'V' or 'I'.
 If *range* = 'A', the routine computes all eigenvalues.
 If *range* = 'V', the routine computes eigenvalues $\lambda(i)$ in the half-open interval: $vl < \lambda(i) \leq vu$.
 If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.

uplo CHARACTER*1. Must be 'U' or 'L'.
 If *uplo* = 'U', *ab* stores the upper triangular part of A .
 If *uplo* = 'L', *ab* stores the lower triangular part of A .

n INTEGER. The order of the matrix A ($n \geq 0$).

kd INTEGER. The number of super- or sub-diagonals in A ($kd \geq 0$).

ab, work COMPLEX for chbev(x)
 DOUBLE COMPLEX for zhbev(x).
Arrays:
ab (*ldab*,*) is an array containing either upper or lower triangular part of the Hermitian matrix A (as specified by *uplo*) in band storage format.
 The second dimension of *ab* must be at least $\max(1, n)$.
work (*) is a workspace array.

	The dimension of <i>work</i> must be at least $\max(1, n)$.
<i>ldab</i>	INTEGER. The leading dimension of <i>ab</i> ; must be at least $kd + 1$.
<i>vl, vu</i>	REAL for <i>chbev</i> x DOUBLE PRECISION for <i>zhbev</i> x. If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: $vl < vu$. If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il, iu</i>	INTEGER. If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$. If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.
<i>abstol</i>	REAL for <i>chbev</i> x DOUBLE PRECISION for <i>zhbev</i> x. The absolute error tolerance to which each eigenvalue is required. See <i>Application notes</i> for details on error tolerance.
<i>ldq, ldz</i>	INTEGER. The leading dimensions of the output arrays <i>q</i> and <i>z</i> , respectively. Constraints: $ldq \geq 1, ldz \geq 1$; If <i>jobz</i> = 'V', then $ldq \geq \max(1, n)$ and $ldz \geq \max(1, n)$.
<i>rwork</i>	REAL for <i>chbev</i> x DOUBLE PRECISION for <i>zhbev</i> x Workspace array, DIMENSION at least $\max(1, 7n)$.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, 5n)$.

Output Parameters

<i>q</i>	COMPLEX for <i>chbev</i> x DOUBLE COMPLEX for <i>zhbev</i> x. Array, DIMENSION (ldz, n).

If *jobz* = 'V', the *n*-by-*n* unitary matrix is used in the reduction to tridiagonal form.
 If *jobz* = 'N', the array *q* is not referenced.

m INTEGER. The total number of eigenvalues found,
 $0 \leq m \leq n$.
 If *range* = 'A', *m* = *n*, and if *range* = 'I', *m* = *iu-il*+1.

w REAL for chbev
 DOUBLE PRECISION for zhbev
 Array, DIMENSION at least $\max(1, n)$. The first *m* elements contain the selected eigenvalues of the matrix *A* in ascending order.

z COMPLEX for chbev
 DOUBLE COMPLEX for zhbev.
 Array *z*(*ldz*,*).
 The second dimension of *z* must be at least $\max(1, m)$.
 If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).
 If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.
 If *jobz* = 'N', then *z* is not referenced.
 Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

ab On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.
 If *uplo* = 'U', the first superdiagonal and the diagonal of the tridiagonal matrix *T* are returned in rows *kd* and *kd*+1 of *ab*, and if *uplo* = 'L', the diagonal and first subdiagonal of *T* are returned in the first two rows of *ab*.

ifail INTEGER.
 Array, DIMENSION at least $\max(1, n)$.

If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge.
 If *jobz* = 'N', then *ifail* is not referenced.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *hbev*x interface are the following:

<i>a</i>	Stands for argument <i>ab</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(kd+1, n)$.
<i>w</i>	Holds the vector of length (n) .
<i>z</i>	Holds the matrix <i>Z</i> of size (n, n) , where the values <i>n</i> and <i>m</i> are significant.
<i>ifail</i>	Holds the vector of length (n) .
<i>q</i>	Holds the matrix <i>Q</i> of size (n, n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>vl</i>	Default value for this element is <i>vl</i> = -HUGE(<i>vl</i>).
<i>vu</i>	Default value for this element is <i>vu</i> = HUGE(<i>vl</i>).
<i>il</i>	Default value for this argument is <i>il</i> = 1.
<i>iu</i>	Default value for this argument is <i>iu</i> = <i>n</i> .
<i>abstol</i>	Default value for this element is <i>abstol</i> = 0.0_WP.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted Note that there will be an error condition if either <i>ifail</i> or <i>q</i> is present and <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows:

`range = 'V'`, if one of or both `vl` and `vu` are present,
`range = 'I'`, if one of or both `il` and `iu` are present,
`range = 'A'`, if none of `vl`, `vu`, `il`, `iu` is present,
 Note that there will be an error condition if one of or both `vl` and `vu`
 are present and at the same time one of or both `il` and `iu` are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon * \max(|a|, |b|)$, where ϵ is the machine precision.

If `abstol` is less than or equal to zero, then $\epsilon * ||T||_1$ will be used in its place, where T is the tridiagonal matrix obtained by reducing A to tridiagonal form. Eigenvalues will be computed most accurately when `abstol` is set to twice the underflow threshold $2 * \text{lamch}('S')$, not zero.

If this routine returns with `info > 0`, indicating that some eigenvectors did not converge, try setting `abstol` to $2 * \text{lamch}('S')$.

?stev

Computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix.

Syntax

FORTRAN 77:

```
call sstev(jobz, n, d, e, z, ldz, work, info)
call dstev(jobz, n, d, e, z, ldz, work, info)
```

Fortran 95:

```
call stev(d, e [,z] [,info])
```

Description

This routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix A .

Input Parameters

`jobz` CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then only eigenvalues are computed.
 If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

n INTEGER. The order of the matrix *A* ($n \geq 0$).

d, e, work REAL for *sstev*
 DOUBLE PRECISION for *dsteve*.
Arrays:
d(*) contains the *n* diagonal elements of the tridiagonal matrix *A*.
 The dimension of *d* must be at least $\max(1, n)$.
e(*) contains the *n*-1 subdiagonal elements of the tridiagonal matrix *A*.
 The dimension of *e* must be at least $\max(1, n-1)$. The *n*-th element of this array is used as workspace.
work(*) is a workspace array.
 The dimension of *work* must be at least $\max(1, 2n-2)$.
 If *jobz* = 'N', *work* is not referenced.

ldz INTEGER. The leading dimension of the output array *z*; $ldz \geq 1$. If *jobz* = 'V' then $ldz \geq \max(1, n)$.

Output Parameters

d On exit, if *info* = 0, contains the eigenvalues of the matrix *A* in ascending order.

z REAL for *sstev*
 DOUBLE PRECISION for *dsteve*
Array, DIMENSION (*ldz*, *).
 The second dimension of *z* must be at least $\max(1, n)$.
 If *jobz* = 'V', then if *info* = 0, *z* contains the orthonormal eigenvectors of the matrix *A*, with the *i*-th column of *z* holding the eigenvector associated with the eigenvalue returned in *d*(*i*).
 If *job* = 'N', then *z* is not referenced.

e On exit, this array is overwritten with intermediate results.

info INTEGER.
 If *info* = 0, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, then the algorithm failed to converge;

i elements of e did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `stev` interface are the following:

d	Holds the vector of length (n) .
e	Holds the vector of length (n) .
z	Holds the matrix Z of size (n, n) .
$jobz$	Restored based on the presence of the argument z as follows: $jobz = 'V'$, if z is present, $jobz = 'N'$, if z is omitted.

?stevd

Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric tridiagonal matrix using divide and conquer algorithm.

Syntax

FORTRAN 77:

```
call sstevd(jobz, n, d, e, z, ldz, work, lwork, iwork, liwork, info)
```

```
call dstevd(jobz, n, d, e, z, ldz, work, lwork, iwork, liwork, info)
```

Fortran 95:

```
call stevd(d, e [,z] [,info])
```

Description

This routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric tridiagonal matrix T . In other words, the routine can compute the spectral factorization of T

as: $T = Z \Lambda Z^T$.

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i , and Z is the orthogonal matrix whose columns are the eigenvectors z_i . Thus,

$$T^* z_i = \lambda_i * z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the *QL* or *QR* algorithm.

There is no complex analogue of this routine.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>n</i>	INTEGER. The order of the matrix <i>T</i> ($n \geq 0$).
<i>d, e, work</i>	REAL for <i>sstevd</i> DOUBLE PRECISION for <i>dstevd</i>. Arrays: <i>d</i>(*) contains the <i>n</i> diagonal elements of the tridiagonal matrix <i>T</i>. The dimension of <i>d</i> must be at least $\max(1, n)$. <i>e</i>(*) contains the <i>n</i>-1 off-diagonal elements of <i>T</i>. The dimension of <i>e</i> must be at least $\max(1, n-1)$. The <i>n</i>-th element of this array is used as workspace. <i>work</i>(*) is a workspace array. The dimension of <i>work</i> must be at least <i>lwork</i>.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i>. Constraints: $ldz \geq 1$ if <i>job</i> = 'N'; $ldz < \max(1, n)$ if <i>job</i> = 'V'.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i>. Constraints: if <i>jobz</i> = 'N' or $n \leq 1$, then $lwork \geq 1$; if <i>jobz</i> = 'V' and $n > 1$, then $lwork \geq n^2 + 4*n + 1$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the required sizes of the $work$ and $iwork$ arrays, returns these values as the first entries of the $work$ and $iwork$ arrays, and no error message related to $lwork$ or $liwork$ is issued by [xerbla](#). See *Application Notes* for details.

$iwork$

INTEGER. Workspace array, its dimension $\max(1, liwork)$.

$liwork$

INTEGER.

The dimension of the array $iwork$.

Constraints:

if $jobz = 'N'$ or $n \leq 1$, then $liwork \geq 1$;

if $jobz = 'V'$ and $n > 1$, then $liwork \geq 5*n+3$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the required sizes of the $work$ and $iwork$ arrays, returns these values as the first entries of the $work$ and $iwork$ arrays, and no error message related to $lwork$ or $liwork$ is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

d

On exit, if $info = 0$, contains the eigenvalues of the matrix T in ascending order.

See also $info$.

z

REAL for `ssteve`

DOUBLE PRECISION for `dsteve`

Array, DIMENSION ($ldz, *$).

The second dimension of z must be:

at least 1 if $jobz = 'N'$;

at least $\max(1, n)$ if $jobz = 'V'$.

If $jobz = 'V'$, then this array is overwritten by the orthogonal matrix Z which contains the eigenvectors of T .

If $jobz = 'N'$, then z is not referenced.

e

On exit, this array is overwritten with intermediate results.

$work(1)$

On exit, if $lwork > 0$, then $work(1)$ returns the required minimal size of $lwork$.

<i>iwork</i> (1)	On exit, if <i>liwork</i> > 0, then <i>iwork</i> (1) returns the required minimal size of <i>liwork</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>i</i>, then the algorithm failed to converge; <i>i</i> indicates the number of elements of an intermediate tridiagonal form which did not converge to zero. If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `stevd` interface are the following:

<i>d</i>	Holds the vector of length (<i>n</i>).
<i>e</i>	Holds the vector of length (<i>n</i>).
<i>z</i>	Holds the matrix <i>z</i> of size (<i>n</i> , <i>n</i>).
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T+E$ such that $\|E\|_2 = O(\epsilon) * \|T\|_2$, where ϵ is the machine precision.

If λ_i is an exact eigenvalue, and m_i is the corresponding computed value, then

$$|\mu_i - \lambda_i| \leq c(n) * \epsilon * \|T\|_2$$

where $c(n)$ is a modestly increasing function of n .

If z_i is the corresponding exact eigenvector, and w_i is the corresponding computed vector, then the angle $\theta(z_i, w_i)$ between them is bounded as follows:

$$\theta(z_i, w_i) \leq c(n) * \epsilon * \|T\|_2 / \min_{i \neq j} |\lambda_i - \lambda_j|.$$

Thus the accuracy of a computed eigenvector depends on the gap between its eigenvalue and all the other eigenvalues.

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run, or set *lwork* = -1 (*liwork* = -1).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If *lwork* = -1 (*liwork* = -1), then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *lwork* (*liwork*) is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?stevx

Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

Syntax

FORTRAN 77:

```
call sstevx(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, work,
iwork, ifail, info)
```

```
call dstevx(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, work,
iwork, ifail, info)
```

Fortran 95:

```
call stevx(d, e, w [, z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol]
[,info])
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix *A*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues $\lambda(i)$ in the half-open interval: $vl < \lambda(i) \leq vu$. If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i>.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).
<i>d, e, work</i>	REAL for <i>sstevx</i> DOUBLE PRECISION for <i>dstevx</i>. Arrays: <i>d</i>(*) contains the <i>n</i> diagonal elements of the tridiagonal matrix <i>A</i>. The dimension of <i>d</i> must be at least $\max(1, n)$. <i>e</i>(*) contains the <i>n</i>-1 subdiagonal elements of <i>A</i>. The dimension of <i>e</i> must be at least $\max(1, n-1)$. The <i>n</i>-th element of this array is used as workspace. <i>work</i>(*) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, 5n)$.
<i>vl, vu</i>	REAL for <i>sstevx</i> DOUBLE PRECISION for <i>dstevx</i>. If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: $vl < vu$. If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il, iu</i>	INTEGER. If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$. If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.

abstol REAL for `sstevx`
DOUBLE PRECISION for `dstevx`. The absolute error tolerance to which each eigenvalue is required. See *Application notes* for details on error tolerance.

ldz INTEGER. The leading dimensions of the output array *z*;
 $ldz \geq 1$. If *jobz* = 'V', then $ldz \geq \max(1, n)$.

iwork INTEGER. Workspace array, DIMENSION at least $\max(1, 5n)$.

Output Parameters

m INTEGER. The total number of eigenvalues found,
 $0 \leq m \leq n$.
If *range* = 'A', $m = n$, and if *range* = 'I', $m = iu - il + 1$.

w, *z* REAL for `sstevx`
DOUBLE PRECISION for `dstevx`.
Arrays:
w(*), DIMENSION at least $\max(1, n)$.
The first *m* elements of *w* contain the selected eigenvalues of the matrix *A* in ascending order.
z(*ldz*,*).
The second dimension of *z* must be at least $\max(1, m)$.
If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).
If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.
If *jobz* = 'N', then *z* is not referenced.
Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

d, *e* On exit, these arrays may be multiplied by a constant factor chosen to avoid overflow or underflow in computing the eigenvalues.

ifail INTEGER.

Array, *DIMENSION* at least $\max(1, n)$.
 If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge.
 If *jobz* = 'N', then *ifail* is not referenced.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.
 If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *stevx* interface are the following:

<i>d</i>	Holds the vector of length (<i>n</i>).
<i>e</i>	Holds the vector of length (<i>n</i>).
<i>w</i>	Holds the vector of length (<i>n</i>).
<i>z</i>	Holds the matrix <i>z</i> of size (<i>n</i> , <i>n</i>), where the values <i>n</i> and <i>m</i> are significant.
<i>ifail</i>	Holds the vector of length (<i>n</i>).
<i>vl</i>	Default value for this element is <i>vl</i> = -HUGE(<i>vl</i>).
<i>vu</i>	Default value for this element is <i>vu</i> = HUGE(<i>vl</i>).
<i>il</i>	Default value for this argument is <i>il</i> = 1.
<i>iu</i>	Default value for this argument is <i>iu</i> = <i>n</i> .
<i>abstol</i>	Default value for this element is <i>abstol</i> = 0.0_WP.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted Note that there will be an error condition if <i>ifail</i> is present and <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: <i>range</i> = 'V', if one of or both <i>vl</i> and <i>vu</i> are present,

$range = 'I'$, if one of or both il and iu are present,
 $range = 'A'$, if none of vl, vu, il, iu is present,
 Note that there will be an error condition if one of or both vl and vu
 are present and at the same time one of or both il and iu are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon \max(|a|, |b|)$, where ϵ is the machine precision.

If $abstol$ is less than or equal to zero, then $\epsilon * ||A||^1$ is used instead. Eigenvalues are computed most accurately when $abstol$ is set to twice the underflow threshold $2 * \lambda_{\text{lamch}}('S')$, not zero.

If this routine returns with $info > 0$, indicating that some eigenvectors did not converge, set $abstol$ to $2 * \lambda_{\text{lamch}}('S')$.

?stevr

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix using the Relatively Robust Representations.

Syntax

FORTRAN 77:

```
call sstevr(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz,
work, lwork, iwork, liwork, info)

call dstevr(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz,
work, lwork, iwork, liwork, info)
```

Fortran 95:

```
call stevr(d, e, w [, z] [,vl] [,vu] [,il] [,iu] [,m] [,isuppz] [,abstol]
[,info])
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix T . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Whenever possible, the routine calls `?stemr` to compute the eigenspectrum using Relatively Robust Representations. `?stegr` computes eigenvalues by the *dqds* algorithm, while orthogonal eigenvectors are computed from various “good” $L^*D^*L^T$ representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For the *i*-th unreduced block of *T*:

- a.** Compute $T - \sigma_i = L_i^*D_i^*L_i^T$, such that $L_i^*D_i^*L_i^T$ is a relatively robust representation.
- b.** Compute the eigenvalues, λ_j , of $L_i^*D_i^*L_i^T$ to high relative accuracy by the *dqds* algorithm.
- c.** If there is a cluster of close eigenvalues, “choose” σ_i close to the cluster, and go to Step (a).
- d.** Given the approximate eigenvalue λ_j of $L_i^*D_i^*L_i^T$, compute the corresponding eigenvector by forming a rank-revealing twisted factorization.

The desired accuracy of the output can be specified by the input parameter *abstol*.

The routine `?stevr` calls `?stemr` when the full spectrum is requested on machines which conform to the IEEE-754 floating point standard. `?stevr` calls `?stebz` and `?stein` on non-IEEE machines and when partial spectrum requests are made.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues $\lambda(i)$ in the half-open interval: $vl < \lambda(i) \leq vu.$ If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> . For <i>range</i> = 'V' or 'I' and $iu - il < n - 1$, <code>sstebz/dstebz</code> and <code>sstein/dstein</code> are called.
<i>n</i>	INTEGER. The order of the matrix <i>T</i> ($n \geq 0$).
<i>d, e, work</i>	REAL for <code>sstevr</code>

DOUBLE PRECISION for `dstevr`.

Arrays:
`d(*)` contains the n diagonal elements of the tridiagonal matrix T .
The dimension of d must be at least $\max(1, n)$.
`e(*)` contains the $n-1$ subdiagonal elements of A .
The dimension of e must be at least $\max(1, n-1)$. The n -th element of this array is used as workspace.
`work` is a workspace array, its dimension $\max(1, lwork)$.

`vl, vu` REAL for `sstevr`
DOUBLE PRECISION for `dstevr`.
If `range = 'V'`, the lower and upper bounds of the interval to be searched for eigenvalues.
Constraint: `vl < vu`.
If `range = 'A'` or `'I'`, `vl` and `vu` are not referenced.

`il, iu` INTEGER.
If `range = 'I'`, the indices in ascending order of the smallest and largest eigenvalues to be returned.
Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$.
If `range = 'A'` or `'V'`, `il` and `iu` are not referenced.

`abstol` REAL for `ssyevr`
DOUBLE PRECISION for `dsyevr`.
The absolute error tolerance to which each eigenvalue/eigenvector is required.
If `jobz = 'V'`, the eigenvalues and eigenvectors output have residual norms bounded by `abstol`, and the dot products between different eigenvectors are bounded by `abstol`. If `abstol < n * eps * |T|`, then $n * eps * |T|$ will be used in its place, where `eps` is the machine precision, and $|T|$ is the 1-norm of the matrix T . The eigenvalues are computed to an accuracy of $eps * |T|$ irrespective of `abstol`.
If high relative accuracy is important, set `abstol` to `?lamch('S')`.

`ldz` INTEGER. The leading dimension of the output array z .
Constraints:

ldz ≥ 1 if *jobz* = 'N';
ldz $\geq \max(1, n)$ if *jobz* = 'V'.
lwork INTEGER.
The dimension of the array *work*. Constraint:
lwork $\geq \max(1, 20*n)$.
If *lwork* = -1, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.
iwork INTEGER.
Workspace array, its dimension $\max(1, liwork)$.
liwork INTEGER.
The dimension of the array *iwork*,
liwork $\geq \max(1, 10*n)$.
If *liwork* = -1, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

m INTEGER. The total number of eigenvalues found,
 $0 \leq m \leq n$. If *range* = 'A', *m* = *n*, and if *range* = 'I',
m = *iu-il*+1.
w, z REAL for *sstevr*
DOUBLE PRECISION for *dstevr*.
Arrays:
w(*), DIMENSION at least $\max(1, n)$.
The first *m* elements of *w* contain the selected eigenvalues of the matrix *T* in ascending order.
z(*ldz*,*).
The second dimension of *z* must be at least $\max(1, m)$.

	If <code>jobz = 'V'</code>, then if <code>info = 0</code>, the first m columns of z contain the orthonormal eigenvectors of the matrix T corresponding to the selected eigenvalues, with the i-th column of z holding the eigenvector associated with $w(i)$. If <code>jobz = 'N'</code>, then z is not referenced. Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array z; if <code>range = 'V'</code>, the exact value of m is not known in advance and an upper bound must be used.
<code>d, e</code>	On exit, these arrays may be multiplied by a constant factor chosen to avoid overflow or underflow in computing the eigenvalues.
<code>isuppz</code>	INTEGER. Array, DIMENSION at least $2 * \max(1, m)$. The support of the eigenvectors in z, i.e., the indices indicating the nonzero elements in z. The i-th eigenvector is nonzero only in elements <code>isuppz(2i-1)</code> through <code>isuppz(2i)</code>. Implemented only for <code>range = 'A'</code> or <code>'I'</code> and <code>iu-il = n-1</code>.
<code>work(1)</code>	On exit, if <code>info = 0</code> , then <code>work(1)</code> returns the required minimal size of <code>lwork</code> .
<code>iwork(1)</code>	On exit, if <code>info = 0</code> , then <code>iwork(1)</code> returns the required minimal size of <code>liwork</code> .
<code>info</code>	INTEGER. If <code>info = 0</code>, the execution is successful. If <code>info = -i</code>, the i-th parameter had an illegal value. If <code>info = i</code>, an internal error has occurred.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `stevr` interface are the following:

<code>d</code>	Holds the vector of length (n) .
<code>e</code>	Holds the vector of length (n) .

<i>w</i>	Holds the vector of length (<i>n</i>).
<i>z</i>	Holds the matrix <i>Z</i> of size (<i>n</i> , <i>n</i>), where the values <i>n</i> and <i>m</i> are significant.
<i>isuppz</i>	Holds the vector of length ($2*n$), where the values ($2*m$) are significant.
<i>vl</i>	Default value for this element is $vl = -HUGE(vl)$.
<i>vu</i>	Default value for this element is $vu = HUGE(vl)$.
<i>il</i>	Default value for this argument is $il = 1$.
<i>iu</i>	Default value for this argument is $iu = n$.
<i>abstol</i>	Default value for this element is $abstol = 0.0\_WP$.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: $jobz = 'V'$, if <i>z</i> is present, $jobz = 'N'$, if <i>z</i> is omitted Note that there will be an error condition if <i>ifail</i> is present and <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: $range = 'V'$, if one of or both <i>vl</i> and <i>vu</i> are present, $range = 'I'$, if one of or both <i>il</i> and <i>iu</i> are present, $range = 'A'$, if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present, Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

Normal execution of the routine `?stegr` may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the IEEE standard default manner.

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run, or set $lwork = -1$ ($liwork = -1$).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If $lwork = -1$ ($liwork = -1$), then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if `lwork` (`liwork`) is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Nonsymmetric Eigenproblems

This section describes LAPACK driver routines used for solving nonsymmetric eigenproblems. See also [computational routines](#) that can be called to solve these problems.

[Table 4-11](#) lists all such driver routines for Fortran 77 interface. Respective routine names in Fortran 95 interface are without the first symbol (see [Routine Naming Conventions](#)).

Table 4-11 Driver Routines for Solving Nonsymmetric Eigenproblems

Routine Name	Operation performed
<code>?gees</code>	Computes the eigenvalues and Schur factorization of a general matrix, and orders the factorization so that selected eigenvalues are at the top left of the Schur form.
<code>?geesx</code>	Computes the eigenvalues and Schur factorization of a general matrix, orders the factorization and computes reciprocal condition numbers.
<code>?geev</code>	Computes the eigenvalues and left and right eigenvectors of a general matrix.
<code>?geevx</code>	Computes the eigenvalues and left and right eigenvectors of a general matrix, with preliminary matrix balancing, and computes reciprocal condition numbers for the eigenvalues and right eigenvectors.

?gees

Computes the eigenvalues and Schur factorization of a general matrix, and orders the factorization so that selected eigenvalues are at the top left of the Schur form.

Syntax

FORTRAN 77:

```
call sgees(jobvs, sort, select, n, a, lda, sdim, wr, wi, vs, ldvs, work,
lwork, bwork, info)

call dgees(jobvs, sort, select, n, a, lda, sdim, wr, wi, vs, ldvs, work,
lwork, bwork, info)

call cgees(jobvs, sort, select, n, a, lda, sdim, w, vs, ldvs, work, lwork,
rwork, bwork, info)

call zgees(jobvs, sort, select, n, a, lda, sdim, w, vs, ldvs, work, lwork,
rwork, bwork, info)
```

Fortran 95:

```
call gees(a, wr, wi [,vs] [,select] [,sdim] [,info])

call gees(a, w [,vs] [,select] [,sdim] [,info])
```

Description

This routine computes for an n -by- n real/complex nonsymmetric matrix A , the eigenvalues, the real Schur form T , and, optionally, the matrix of Schur vectors Z . This gives the Schur factorization $A = Z^* T^* Z^H$.

Optionally, it also orders the eigenvalues on the diagonal of the real-Schur/Schur form so that selected eigenvalues are at the top left. The leading columns of Z then form an orthonormal basis for the invariant subspace corresponding to the selected eigenvalues.

A real matrix is in real-Schur form if it is upper quasi-triangular with 1-by-1 and 2-by-2 blocks. 2-by-2 blocks will be standardized in the form

$$\begin{pmatrix} a & b \\ c & a \end{pmatrix}$$

where $b^*c < 0$. The eigenvalues of such a block are $a \pm \sqrt{bc}$

A complex matrix is in Schur form if it is upper triangular.

Input Parameters

jobvs CHARACTER*1. Must be 'N' or 'V'.
 If *jobvs* = 'N', then Schur vectors are not computed.
 If *jobvs* = 'V', then Schur vectors are computed.

sort CHARACTER*1. Must be 'N' or 'S'. Specifies whether or not to order the eigenvalues on the diagonal of the Schur form.
 If *sort* = 'N', then eigenvalues are not ordered.
 If *sort* = 'S', eigenvalues are ordered (see *select*).

select LOGICAL FUNCTION of two REAL arguments for real flavors.
 LOGICAL FUNCTION of one COMPLEX argument for complex flavors.
select must be declared EXTERNAL in the calling subroutine.
 If *sort* = 'S', *select* is used to select eigenvalues to sort to the top left of the Schur form.
 If *sort* = 'N', *select* is not referenced.

For real flavors:
 An eigenvalue $w_r(j) + \text{sqrt}(-1) * w_i(j)$ is selected if *select*($w_r(j)$, $w_i(j)$) is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.
 Note that a selected complex eigenvalue may no longer satisfy *select*($w_r(j)$, $w_i(j)$) = .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case *info* may be set to $n+2$ (see *info* below).

For complex flavors:
 An eigenvalue $w(j)$ is selected if *select*($w(j)$) is true.

<i>n</i>	INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).
<i>a, work</i>	REAL for sgees DOUBLE PRECISION for dgees COMPLEX for cgees DOUBLE COMPLEX for zgees. Arrays: <i>a(lda,*)</i> is an array containing the <i>n</i> -by- <i>n</i> matrix <i>A</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The first dimension of the array <i>a</i> . Must be at least $\max(1, n)$.
<i>ldvs</i>	INTEGER. The leading dimension of the output array <i>vs</i> . Constraints: $ldvs \geq 1$; $ldvs \geq \max(1, n)$ if <i>jobvs</i> = 'V'.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . Constraint: $lwork \geq \max(1, 3n)$ for real flavors; $lwork \geq \max(1, 2n)$ for complex flavors. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla .
<i>rwork</i>	REAL for cgees DOUBLE PRECISION for zgees Workspace array, DIMENSION at least $\max(1, n)$. Used in complex flavors only.
<i>bwork</i>	LOGICAL. Workspace array, DIMENSION at least $\max(1, n)$. Not referenced if <i>sort</i> = 'N'.

Output Parameters

<i>a</i>	On exit, this array is overwritten by the real-Schur/Schur form <i>T</i> .

sdim INTEGER.
 If *sort* = 'N', *sdim* = 0.
 If *sort* = 'S', *sdim* is equal to the number of eigenvalues (after sorting) for which *select* is true.
 Note that for real flavors complex conjugate pairs for which *select* is true for either eigenvalue count as 2.

wr, wi REAL for sgees
 DOUBLE PRECISION for dgees
 Arrays, DIMENSION at least max(1, *n*) each. Contain the real and imaginary parts, respectively, of the computed eigenvalues, in the same order that they appear on the diagonal of the output real-Schur form *T*. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

w COMPLEX for cgees
 DOUBLE COMPLEX for zgees.
 Array, DIMENSION at least max(1, *n*). Contains the computed eigenvalues. The eigenvalues are stored in the same order as they appear on the diagonal of the output Schur form *T*.

vs REAL for sgees
 DOUBLE PRECISION for dgees
 COMPLEX for cgees
 DOUBLE COMPLEX for zgees.
 Array *vs(ldvs,*)*; the second dimension of *vs* must be at least max(1, *n*).
 If *jobvs* = 'V', *vs* contains the orthogonal/unitary matrix *Z* of Schur vectors.
 If *jobvs* = 'N', *vs* is not referenced.

work(1) On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*th parameter had an illegal value.
 If *info* = *i*, and
 $i \leq n$:

the QR algorithm failed to compute all the eigenvalues; elements $1:ilo-1$ and $i+1:n$ of wr and wi (for real flavors) or w (for complex flavors) contain those eigenvalues which have converged; if $jobvs = 'V'$, vs contains the matrix which reduces A to its partially converged Schur form;
 $i = n+1$:
the eigenvalues could not be reordered because some eigenvalues were too close to separate (the problem is very ill-conditioned);
 $i = n+2$:
after reordering, round-off changed values of some complex eigenvalues so that leading eigenvalues in the Schur form no longer satisfy $select = .TRUE..$ This could also be caused by underflow due to scaling.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gees` interface are the following:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>wr</i>	Holds the vector of length (n) . Used in real flavors only.
<i>wi</i>	Holds the vector of length (n) . Used in real flavors only.
<i>w</i>	Holds the vector of length (n) . Used in complex flavors only.
<i>vs</i>	Holds the matrix VS of size (n, n) .
<i>jobvs</i>	Restored based on the presence of the argument <i>vs</i> as follows: $jobvs = 'V'$, if <i>vs</i> is present, $jobvs = 'N'$, if <i>vs</i> is omitted.
<i>sort</i>	Restored based on the presence of the argument <i>select</i> as follows: $sort = 'S'$, if <i>select</i> is present, $sort = 'N'$, if <i>select</i> is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?geesx

Computes the eigenvalues and Schur factorization of a general matrix, orders the factorization and computes reciprocal condition numbers.

Syntax

FORTRAN 77:

```
call sgeesx(jobvs, sort, select, sense, n, a, lda, sdim, wr, wi, vs, ldvs,
rconde, rcondv, work, lwork, iwork, liwork, bwork, info)

call dgeesx(jobvs, sort, select, sense, n, a, lda, sdim, wr, wi, vs, ldvs,
rconde, rcondv, work, lwork, iwork, liwork, bwork, info)

call cgeesx(jobvs, sort, select, sense, n, a, lda, sdim, w, vs, ldvs, rconde,
rcondv, work, lwork, rwork, bwork, info)

call zgeesx(jobvs, sort, select, sense, n, a, lda, sdim, w, vs, ldvs, rconde,
rcondv, work, lwork, rwork, bwork, info)
```

Fortran 95:

```
call geesx(a, wr, wi [,vs] [,select] [,sdim] [,rconde] [,rconde] [,info])
call geesx(a, w [,vs] [,select] [,sdim] [,rconde] [,rconde] [,info])
```

Description

This routine computes for an n -by- n real/complex nonsymmetric matrix A , the eigenvalues, the real-Schur/Schur form T , and, optionally, the matrix of Schur vectors Z . This gives the Schur factorization $A = Z^* T^* Z^H$.

Optionally, it also orders the eigenvalues on the diagonal of the real-Schur/Schur form so that selected eigenvalues are at the top left; computes a reciprocal condition number for the average of the selected eigenvalues (*rconde*); and computes a reciprocal condition number for the right invariant subspace corresponding to the selected eigenvalues (*rcondv*). The leading columns of Z form an orthonormal basis for this invariant subspace.

For further explanation of the reciprocal condition numbers *rconde* and *rcondv*, see [LUG], Section 4.10 (where these quantities are called *s* and *sep* respectively).

A real matrix is in real-Schur form if it is upper quasi-triangular with 1-by-1 and 2-by-2 blocks. 2-by-2 blocks will be standardized in the form

$$\begin{pmatrix} a & b \\ c & a \end{pmatrix}$$

where $b^*c < 0$. The eigenvalues of such a block are $a \pm \sqrt{bc}$

A complex matrix is in Schur form if it is upper triangular.

Input Parameters

<i>jobvs</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobvs</i> = 'N', then Schur vectors are not computed. If <i>jobvs</i> = 'V', then Schur vectors are computed.
<i>sort</i>	CHARACTER*1. Must be 'N' or 'S'. Specifies whether or not to order the eigenvalues on the diagonal of the Schur form. If <i>sort</i> = 'N', then eigenvalues are not ordered. If <i>sort</i> = 'S', eigenvalues are ordered (see <i>select</i>).
<i>select</i>	LOGICAL FUNCTION of two REAL arguments for real flavors.

LOGICAL FUNCTION of one COMPLEX argument for complex flavors.

select must be declared EXTERNAL in the calling subroutine.

If *sort* = 'S', *select* is used to select eigenvalues to sort to the top left of the Schur form.

If *sort* = 'N', *select* is not referenced.

For real flavors:

An eigenvalue $w_r(j) + \sqrt{-1} * w_i(j)$ is selected if *select*(*w_r*(*j*), *w_i*(*j*)) is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.

Note that a selected complex eigenvalue may no longer satisfy *select*(*w_r*(*j*), *w_i*(*j*)) = .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case *info* may be set to *n*+2 (see *info* below).

For complex flavors:

An eigenvalue *w*(*j*) is selected if *select*(*w*(*j*)) is true.

sense

CHARACTER*1. Must be 'N', 'E', 'V', or 'B'. Determines which reciprocal condition number are computed.

If *sense* = 'N', none are computed;

If *sense* = 'E', computed for average of selected eigenvalues only;

If *sense* = 'V', computed for selected right invariant subspace only;

If *sense* = 'B', computed for both.

If *sense* is 'E', 'V', or 'B', then *sort* must equal 'S'.

n

INTEGER. The order of the matrix *A* (*n* ≥ 0).

a, *work*

REAL for *sgeesx*

DOUBLE PRECISION for *dgeesx*

COMPLEX for *cgeesx*

DOUBLE COMPLEX for *zgeesx*.

Arrays:

a(*lda*,*) is an array containing the *n*-by-*n* matrix *A*.

The second dimension of *a* must be at least max(1, *n*).

work is a workspace array, its dimension max(1, *lwork*).

<i>lda</i>	INTEGER. The first dimension of the array <i>a</i> . Must be at least $\max(1, n)$.
<i>ldvs</i>	INTEGER. The leading dimension of the output array <i>vs</i> . Constraints: $ldvs \geq 1$; $ldvs \geq \max(1, n)$ if <i>jobvs</i> = 'V'.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . Constraint: $lwork \geq \max(1, 3n)$ for real flavors; $lwork \geq \max(1, 2n)$ for complex flavors. Also, if <i>sense</i> = 'E', 'V', or 'B', then $lwork \geq n+2*sdim*(n-sdim)$ for real flavors; $lwork \geq 2*sdim*(n-sdim)$ for complex flavors; where <i>sdim</i> is the number of selected eigenvalues computed by this routine. Note that $2*sdim*(n-sdim) \leq n*n/2$. Note also that an error is only returned if $lwork < \max(1, 2*n)$, but if <i>sense</i> = 'E', or 'V', or 'B' this may not be large enough. For good performance, <i>lwork</i> must generally be larger. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates upper bound on the optimal size of the array <i>work</i> , returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION (<i>liwork</i>). Used in real flavors only. Not referenced if <i>sense</i> = 'N' or 'E'.
<i>liwork</i>	INTEGER. The dimension of the array <i>iwork</i> . Used in real flavors only. Constraint: $liwork \geq 1$; if <i>sense</i> = 'V' or 'B', $liwork \geq sdim*(n-sdim)$.
<i>rwork</i>	REAL for cgeesx DOUBLE PRECISION for zgeesx

bwork Workspace array, `DIMENSION` at least $\max(1, n)$. Used in complex flavors only.

LOGICAL. Workspace array, `DIMENSION` at least $\max(1, n)$.
Not referenced if `sort = 'N'`.

Output Parameters

a On exit, this array is overwritten by the real-Schur/Schur form T .

sdim INTEGER.
If `sort = 'N'`, `sdim` = 0.
If `sort = 'S'`, `sdim` is equal to the number of eigenvalues (after sorting) for which `select` is true.
Note that for real flavors complex conjugate pairs for which `select` is true for either eigenvalue count as 2.

wr, wi REAL for `sgeesx`
DOUBLE PRECISION for `dgeesx`
Arrays, `DIMENSION` at least $\max(1, n)$ each. Contain the real and imaginary parts, respectively, of the computed eigenvalues, in the same order that they appear on the diagonal of the output real-Schur form T . Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

w COMPLEX for `cgeesx`
DOUBLE COMPLEX for `zgeesx`.
Array, `DIMENSION` at least $\max(1, n)$. Contains the computed eigenvalues. The eigenvalues are stored in the same order as they appear on the diagonal of the output Schur form T .

vs REAL for `sgeesx`
DOUBLE PRECISION for `dgeesx`
COMPLEX for `cgeesx`
DOUBLE COMPLEX for `zgeesx`.
Array `vs(ldvs,*)`; the second dimension of `vs` must be at least $\max(1, n)$.
If `jobvs = 'V'`, `vs` contains the orthogonal/unitary matrix Z of Schur vectors.
If `jobvs = 'N'`, `vs` is not referenced.

<i>rconde, rcondv</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. If <i>sense</i> = 'E' or 'B', <i>rconde</i> contains the reciprocal condition number for the average of the selected eigenvalues. If <i>sense</i> = 'N' or 'V', <i>rconde</i> is not referenced. If <i>sense</i> = 'V' or 'B', <i>rcondv</i> contains the reciprocal condition number for the selected right invariant subspace. If <i>sense</i> = 'N' or 'E', <i>rcondv</i> is not referenced.
<i>work(1)</i>	On exit, if <i>info</i> = 0, then <i>work(1)</i> returns the required minimal size of <i>lwork</i>.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = -<i>i</i>, the <i>i</i>th parameter had an illegal value. If <i>info</i> = <i>i</i>, and <i>i</i> ≤ <i>n</i>: the <i>QR</i> algorithm failed to compute all the eigenvalues; elements 1:<i>ilo</i>-1 and <i>i</i>+1:<i>n</i> of <i>wr</i> and <i>wi</i> (for real flavors) or <i>w</i> (for complex flavors) contain those eigenvalues which have converged; if <i>jobvs</i> = 'V', <i>vs</i> contains the transformation which reduces <i>A</i> to its partially converged Schur form; <i>i</i> = <i>n</i>+1: the eigenvalues could not be reordered because some eigenvalues were too close to separate (the problem is very ill-conditioned); <i>i</i> = <i>n</i>+2: after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the Schur form no longer satisfy <i>select</i> = .TRUE.. This could also be caused by underflow due to scaling.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `geesx` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>wr</i>	Holds the vector of length (<i>n</i>). Used in real flavors only.
<i>wi</i>	Holds the vector of length (<i>n</i>). Used in real flavors only.
<i>w</i>	Holds the vector of length (<i>n</i>). Used in complex flavors only.
<i>vs</i>	Holds the matrix <i>VS</i> of size (<i>n</i> , <i>n</i>).
<i>jobvs</i>	Restored based on the presence of the argument <i>vs</i> as follows: <i>jobvs</i> = 'V', if <i>vs</i> is present, <i>jobvs</i> = 'N', if <i>vs</i> is omitted.
<i>sort</i>	Restored based on the presence of the argument <i>select</i> as follows: <i>sort</i> = 'S', if <i>select</i> is present, <i>sort</i> = 'N', if <i>select</i> is omitted.
<i>sense</i>	Restored based on the presence of arguments <i>rconde</i> and <i>rcondv</i> as follows: <i>sense</i> = 'B', if both <i>rconde</i> and <i>rcondv</i> are present, <i>sense</i> = 'E', if <i>rconde</i> is present and <i>rcondv</i> omitted, <i>sense</i> = 'V', if <i>rconde</i> is omitted and <i>rcondv</i> present, <i>sense</i> = 'N', if both <i>rconde</i> and <i>rcondv</i> are omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = -1 (*liwork* = -1).

If you choose the first option and set any of admissible *lwork* (or *liwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?geev

Computes the eigenvalues and left and right eigenvectors of a general matrix.

Syntax

FORTRAN 77:

```
call sgeev(jobvl, jobvr, n, a, lda, wr, wi, vl, ldvl, vr, ldvr, work, lwork,
info)

call dgeev(jobvl, jobvr, n, a, lda, wr, wi, vl, ldvl, vr, ldvr, work, lwork,
info)

call cgeev(jobvl, jobvr, n, a, lda, w, vl, ldvl, vr, ldvr, work, lwork, rwork,
info)

call zgeev(jobvl, jobvr, n, a, lda, w, vl, ldvl, vr, ldvr, work, lwork, rwork,
info)
```

Fortran 95:

```
call geev(a, wr, wi [,vl] [,vr] [,info])
call geev(a, w [,vl] [,vr] [,info])
```

Description

This routine computes for an n -by- n real/complex nonsymmetric matrix A , the eigenvalues and, optionally, the left and/or right eigenvectors. The right eigenvector $v(j)$ of A satisfies

$$A \cdot v(j) = \lambda(j) \cdot v(j)$$

where $\lambda(j)$ is its eigenvalue.

The left eigenvector $u(j)$ of A satisfies

$$u(j)^H \cdot A = \lambda(j) \cdot u(j)^H$$

where $u(j)^H$ denotes the conjugate transpose of $u(j)$. The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

Input Parameters

jobvl CHARACTER*1. Must be 'N' or 'V'.

If $jobvl = 'N'$, then left eigenvectors of A are not computed.
 If $jobvl = 'V'$, then left eigenvectors of A are computed.

jobvr CHARACTER*1. Must be 'N' or 'V'.
 If $jobvr = 'N'$, then right eigenvectors of A are not computed.
 If $jobvr = 'V'$, then right eigenvectors of A are computed.

n INTEGER. The order of the matrix A ($n \geq 0$).

a, work REAL for sgeev
 DOUBLE PRECISION for dgeev
 COMPLEX for cgeev
 DOUBLE COMPLEX for zgeev.

Arrays:
a(*lda*,*) is an array containing the n -by- n matrix A .
 The second dimension of *a* must be at least $\max(1, n)$.
work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The first dimension of the array *a*. Must be at least $\max(1, n)$.

ldvl, ldvr INTEGER. The leading dimensions of the output arrays *vl* and *vr*, respectively.

Constraints:
 $ldvl \geq 1; ldvr \geq 1$.
 If $jobvl = 'V'$, $ldvl \geq \max(1, n)$;
 If $jobvr = 'V'$, $ldvr \geq \max(1, n)$.

lwork INTEGER.
 The dimension of the array *work*.
Constraint:
 $lwork \geq \max(1, 3n)$, and if $jobvl = 'V'$ or $jobvr = 'V'$, $lwork < \max(1, 4n)$ (for real flavors);
 $lwork < \max(1, 2n)$ (for complex flavors).
 For good performance, *lwork* must generally be larger.
 If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

rwork REAL for cgeev
DOUBLE PRECISION for zgeev
Workspace array, DIMENSION at least $\max(1, 2n)$. Used in complex flavors only.

Output Parameters

a On exit, this array is overwritten by intermediate results.

wr, wi REAL for sgeev
DOUBLE PRECISION for dgeev
Arrays, DIMENSION at least $\max(1, n)$ each.
Contain the real and imaginary parts, respectively, of the computed eigenvalues. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

w COMPLEX for cgeev
DOUBLE COMPLEX for zgeev.
Array, DIMENSION at least $\max(1, n)$.
Contains the computed eigenvalues.

vl, vr REAL for sgeev
DOUBLE PRECISION for dgeev
COMPLEX for cgeev
DOUBLE COMPLEX for zgeev.
Arrays:
 $vl(ldvl,*)$; the second dimension of *vl* must be at least $\max(1, n)$.
If *jobvl* = 'V', the left eigenvectors $u(j)$ are stored one after another in the columns of *vl*, in the same order as their eigenvalues.
If *jobvl* = 'N', *vl* is not referenced.
For real flavors:
If the *j*-th eigenvalue is real, then $u(j) = vl(:, j)$, the *j*-th column of *vl*.
If the *j*-th and (*j*+1)-st eigenvalues form a complex conjugate pair, then $u(j) = vl(:, j) + i*vl(:, j+1)$ and $u(j+1) = vl(:, j) - i*vl(:, j+1)$, where $i = \sqrt{-1}$.
For complex flavors:

$u(j) = vl(:, j)$, the j -th column of vl .
 $vr(ldvr, *)$; the second dimension of vr must be at least $\max(1, n)$.
 If $jobvr = 'V'$, the right eigenvectors $v(j)$ are stored one after another in the columns of vr , in the same order as their eigenvalues.
 If $jobvr = 'N'$, vr is not referenced.
For real flavors:
 If the j -th eigenvalue is real, then $v(j) = vr(:, j)$, the j -th column of vr .
 If the j -th and $(j+1)$ -st eigenvalues form a complex conjugate pair, then $v(j) = vr(:, j) + i*vr(:, j+1)$ and $v(j+1) = vr(:, j) - i*vr(:, j+1)$, where $i = \text{sqrt}(-1)$.
For complex flavors:
 $v(j) = vr(:, j)$, the j -th column of vr .
 On exit, if $info = 0$, then $work(1)$ returns the required minimal size of $lwork$.
 INTEGER.
 If $info = 0$, the execution is successful.
 If $info = -i$, the i th parameter had an illegal value.
 If $info = i$, the QR algorithm failed to compute all the eigenvalues, and no eigenvectors have been computed; elements $i+1:n$ of wr and wi (for real flavors) or w (for complex flavors) contain those eigenvalues which have converged.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `geev` interface are the following:

a	Holds the matrix A of size (n, n) .
wr	Holds the vector of length (n) . Used in real flavors only.
wi	Holds the vector of length (n) . Used in real flavors only.
w	Holds the vector of length (n) . Used in complex flavors only.

<i>vl</i>	Holds the matrix <i>VL</i> of size (n, n) .
<i>vr</i>	Holds the matrix <i>VR</i> of size (n, n) .
<i>jobvl</i>	Restored based on the presence of the argument <i>vl</i> as follows: <i>jobvl</i> = 'V', if <i>vl</i> is present, <i>jobvl</i> = 'N', if <i>vl</i> is omitted.
<i>jobvr</i>	Restored based on the presence of the argument <i>vr</i> as follows: <i>jobvr</i> = 'V', if <i>vr</i> is present, <i>jobvr</i> = 'N', if <i>vr</i> is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?geevx

Computes the eigenvalues and left and right eigenvectors of a general matrix, with preliminary matrix balancing, and computes reciprocal condition numbers for the eigenvalues and right eigenvectors.

Syntax

FORTRAN 77:

```
call sgeevx(balanc, jobvl, jobvr, sense, n, a, lda, wr, wi, vl, ldvl, vr,
ldvr, ilo, ihi, scale, abnrm, rconde, rcondv, work, lwork, iwork, info)

call dgeevx(balanc, jobvl, jobvr, sense, n, a, lda, wr, wi, vl, ldvl, vr,
ldvr, ilo, ihi, scale, abnrm, rconde, rcondv, work, lwork, iwork, info)

call cgeevx(balanc, jobvl, jobvr, sense, n, a, lda, w, vl, ldvl, vr, ldvr,
ilo, ihi, scale, abnrm, rconde, rcondv, work, lwork, rwork, info)

call zgeevx(balanc, jobvl, jobvr, sense, n, a, lda, w, vl, ldvl, vr, ldvr,
ilo, ihi, scale, abnrm, rconde, rcondv, work, lwork, rwork, info)
```

Fortran 95:

```
call geevx(a, wr, wi [,vl] [,vr] [,balanc] [,ilo] [,ihi] [,scale] [,abnrm] [,
rconde] [,rcondv] [,info])

call geevx(a, w [,vl] [,vr] [,balanc] [,ilo] [,ihi] [,scale] [,abnrm] [,rconde]
[, rcondv] [,info])
```

Description

This routine computes for an n -by- n real/complex nonsymmetric matrix A , the eigenvalues and, optionally, the left and/or right eigenvectors.

Optionally also, it computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (ilo , ihi , $scale$, and $abnrm$), reciprocal condition numbers for the eigenvalues ($rconde$), and reciprocal condition numbers for the right eigenvectors ($rcondv$).

The right eigenvector $v(j)$ of A satisfies

$$A \cdot v(j) = \lambda(j) \cdot v(j)$$

where $\lambda(j)$ is its eigenvalue.

The left eigenvector $u(j)$ of A satisfies

$$u(j)^H A = \lambda(j) * u(j)^H$$

where $u(j)^H$ denotes the conjugate transpose of $u(j)$. The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

Balancing a matrix means permuting the rows and columns to make it more nearly upper triangular, and applying a diagonal similarity transformation $D * A * \text{inv}(D)$, where D is a diagonal matrix, to make its rows and columns closer in norm and the condition numbers of its eigenvalues and eigenvectors smaller. The computed reciprocal condition numbers correspond to the balanced matrix. Permuting rows and columns will not change the condition numbers in exact arithmetic) but diagonal scaling will. For further explanation of balancing, see [LUG], Section 4.10.

Input Parameters

<i>balanc</i>	CHARACTER*1. Must be 'N', 'P', 'S', or 'B'. Indicates how the input matrix should be diagonally scaled and/or permuted to improve the conditioning of its eigenvalues. If <i>balanc</i> = 'N', do not diagonally scale or permute; If <i>balanc</i> = 'P', perform permutations to make the matrix more nearly upper triangular. Do not diagonally scale; If <i>balanc</i> = 'S', diagonally scale the matrix, i.e. replace A by $D * A * \text{inv}(D)$, where D is a diagonal matrix chosen to make the rows and columns of A more equal in norm. Do not permute; If <i>balanc</i> = 'B', both diagonally scale and permute A. Computed reciprocal condition numbers will be for the matrix after balancing and/or permuting. Permuting does not change condition numbers (in exact arithmetic), but balancing does.
<i>jobvl</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobvl</i> = 'N', left eigenvectors of A are not computed; If <i>jobvl</i> = 'V', left eigenvectors of A are computed. If <i>sense</i> = 'E' or 'B', then <i>jobvl</i> must be 'V'.
<i>jobvr</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobvr</i> = 'N', right eigenvectors of A are not computed; If <i>jobvr</i> = 'V', right eigenvectors of A are computed. If <i>sense</i> = 'E' or 'B', then <i>jobvr</i> must be 'V'.

sense CHARACTER*1. Must be 'N', 'E', 'V', or 'B'. Determines which reciprocal condition number are computed.
 If *sense* = 'N', none are computed;
 If *sense* = 'E', computed for eigenvalues only;
 If *sense* = 'V', computed for right eigenvectors only;
 If *sense* = 'B', computed for eigenvalues and right eigenvectors.
 If *sense* is 'E' or 'B', both left and right eigenvectors must also be computed (*jobvl* = 'V' and *jobvr* = 'V').

n INTEGER. The order of the matrix *A* ($n \geq 0$).

a, work REAL for *sgeevx*
 DOUBLE PRECISION for *dgeevx*
 COMPLEX for *cgeevx*
 DOUBLE COMPLEX for *zgeevx*.
 Arrays:
a(*lda*,*) is an array containing the *n*-by-*n* matrix *A*.
 The second dimension of *a* must be at least $\max(1, n)$.
work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The first dimension of the array *a*. Must be at least $\max(1, n)$.

ldvl, ldvr INTEGER. The leading dimensions of the output arrays *vl* and *vr*, respectively.
 Constraints:
 $ldvl \geq 1$; $ldvr \geq 1$.
 If *jobvl* = 'V', $ldvl \geq \max(1, n)$;
 If *jobvr* = 'V', $ldvr \geq \max(1, n)$.

lwork INTEGER.
 The dimension of the array *work*.
 For real flavors:
 If *sense* = 'N' or 'E', $lwork \geq \max(1, 2n)$, and if *jobvl* = 'V' or *jobvr* = 'V', $lwork \geq 3n$;
 If *sense* = 'V' or 'B', $lwork \geq n*(n+6)$.
 For good performance, *lwork* must generally be larger.
 For complex flavors:

If *sense* = 'N' or 'E', $lwork \geq \max(1, 2n)$;

If *sense* = 'V' or 'B', $lwork \geq n^2 + 2n$. For good performance, *lwork* must generally be larger.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#).

rwork

REAL for *cgeevx*

DOUBLE PRECISION for *zgeevx*

Workspace array, DIMENSION at least $\max(1, 2n)$. Used in complex flavors only.

iwork

INTEGER.

Workspace array, DIMENSION at least $\max(1, 2n-2)$. Used in real flavors only. Not referenced if *sense* = 'N' or 'E'.

Output Parameters

a

On exit, this array is overwritten.

If *jobvl* = 'V' or *jobvr* = 'V', it contains the real-Schur/Schur form of the balanced version of the input matrix *A*.

wr, wi

REAL for *sgeevx*

DOUBLE PRECISION for *dgeevx*

Arrays, DIMENSION at least $\max(1, n)$ each. Contain the real and imaginary parts, respectively, of the computed eigenvalues. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

w

COMPLEX for *cgeevx*

DOUBLE COMPLEX for *zgeevx*.

Array, DIMENSION at least $\max(1, n)$. Contains the computed eigenvalues.

vl, vr

REAL for *sgeevx*

DOUBLE PRECISION for *dgeevx*

COMPLEX for *cgeevx*

DOUBLE COMPLEX for *zgeevx*.

Arrays:

$vl(ldvl,*)$; the second dimension of vl must be at least $\max(1, n)$.

If $jobvl = 'V'$, the left eigenvectors $u(j)$ are stored one after another in the columns of vl , in the same order as their eigenvalues.

If $jobvl = 'N'$, vl is not referenced.

For real flavors:

If the j -th eigenvalue is real, then $u(j) = vl(:, j)$, the j -th column of vl .

If the j -th and $(j+1)$ -st eigenvalues form a complex conjugate pair, then $u(j) = vl(:, j) + i*vl(:, j+1)$ and $u(j+1) = vl(:, j) - i*vl(:, j+1)$, where $i = \sqrt{-1}$.

For complex flavors:

$u(j) = vl(:, j)$, the j -th column of vl .

$vr(ldvr,*)$; the second dimension of vr must be at least $\max(1, n)$.

If $jobvr = 'V'$, the right eigenvectors $v(j)$ are stored one after another in the columns of vr , in the same order as their eigenvalues.

If $jobvr = 'N'$, vr is not referenced.

For real flavors:

If the j -th eigenvalue is real, then $v(j) = vr(:, j)$, the j -th column of vr .

If the j -th and $(j+1)$ -st eigenvalues form a complex conjugate pair, then $v(j) = vr(:, j) + i*vr(:, j+1)$ and $v(j+1) = vr(:, j) - i*vr(:, j+1)$, where $i = \sqrt{-1}$.

For complex flavors:

$v(j) = vr(:, j)$, the j -th column of vr .

ilo, ihi

INTEGER. *ilo* and *ihi* are integer values determined when A was balanced.

The balanced $A(i, j) = 0$ if $i > j$ and $j = 1, \dots, ilo-1$ or $i = ihi+1, \dots, n$.

If $balanc = 'N'$ or $'S'$, $ilo = 1$ and $ihi = n$.

scale

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

	Array, <code>DIMENSION</code> at least $\max(1, n)$. Details of the permutations and scaling factors applied when balancing <i>A</i>. If $P(j)$ is the index of the row and column interchanged with row and column j, and $D(j)$ is the scaling factor applied to row and column j, then $scale(j) = P(j)$, for $j = 1, \dots, ilo-1$ $= D(j)$, for $j = ilo, \dots, ihi$ $= P(j)$ for $j = ihi+1, \dots, n$. The order in which the interchanges are made is n to $ihi+1$, then 1 to $ilo-1$.
<i>abnrm</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. The one-norm of the balanced matrix (the maximum of the sum of absolute values of elements of any column).
<i>rconde, rcondv</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, <code>DIMENSION</code> at least $\max(1, n)$ each. <i>rconde(j)</i> is the reciprocal condition number of the j-th eigenvalue. <i>rcondv(j)</i> is the reciprocal condition number of the j-th right eigenvector.
<i>work(1)</i>	On exit, if <i>info</i> = 0, then <i>work(1)</i> returns the required minimal size of <i>lwork</i>.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = -<i>i</i>, the <i>i</i>th parameter had an illegal value. If <i>info</i> = <i>i</i>, the <i>QR</i> algorithm failed to compute all the eigenvalues, and no eigenvectors or condition numbers have been computed; elements 1:<i>ilo</i>-1 and <i>i</i>+1:<i>n</i> of <i>wr</i> and <i>wi</i> (for real flavors) or <i>w</i> (for complex flavors) contain eigenvalues which have converged.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `geevx` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (n, n) .
<i>wr</i>	Holds the vector of length (n) . Used in real flavors only.
<i>wi</i>	Holds the vector of length (n) . Used in real flavors only.
<i>w</i>	Holds the vector of length (n) . Used in complex flavors only.
<i>vl</i>	Holds the matrix <i>VL</i> of size (n, n) .
<i>vr</i>	Holds the matrix <i>VR</i> of size (n, n) .
<i>scale</i>	Holds the vector of length (n) .
<i>rconde</i>	Holds the vector of length (n) .
<i>rcondv</i>	Holds the vector of length (n) .
<i>balanc</i>	Must be 'N', 'B', 'P' or 'S'. The default value is 'N'.
<i>jobvl</i>	Restored based on the presence of the argument <i>vl</i> as follows: <i>jobvl</i> = 'V', if <i>vl</i> is present, <i>jobvl</i> = 'N', if <i>vl</i> is omitted.
<i>jobvr</i>	Restored based on the presence of the argument <i>vr</i> as follows: <i>jobvr</i> = 'V', if <i>vr</i> is present, <i>jobvr</i> = 'N', if <i>vr</i> is omitted.
<i>sense</i>	Restored based on the presence of arguments <i>rconde</i> and <i>rcondv</i> as follows: <i>sense</i> = 'B', if both <i>rconde</i> and <i>rcondv</i> are present, <i>sense</i> = 'E', if <i>rconde</i> is present and <i>rcondv</i> omitted, <i>sense</i> = 'V', if <i>rconde</i> is omitted and <i>rcondv</i> present, <i>sense</i> = 'N', if both <i>rconde</i> and <i>rcondv</i> are omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Singular Value Decomposition

This section describes LAPACK driver routines used for solving singular value problems. See also computational routines [computational routines](#) that can be called to solve these problems. [Table 4-12](#) lists all such driver routines for Fortran 77 interface. Respective routine names in Fortran 95 interface are without the first symbol (see [Routine Naming Conventions](#)).

Table 4-12 Driver Routines for Singular Value Decomposition

Routine Name	Operation performed
?gesvd	Computes the singular value decomposition of a general rectangular matrix.
?gesdd	Computes the singular value decomposition of a general rectangular matrix using a divide and conquer method.
?ggsvd	Computes the generalized singular value decomposition of a pair of general rectangular matrices.

?gesvd

Computes the singular value decomposition of a general rectangular matrix.

Syntax

FORTRAN 77:

```
call sgesvd(jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, info)
call dgesvd(jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, info)
call cgesvd(jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, rwork, info)
call zgesvd(jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, rwork, info)
```

Fortran 95:

```
call gesvd(a, s [,u] [,vt] [,ww] [,job] [,info])
```

Description

This routine computes the singular value decomposition (SVD) of a real/complex m -by- n matrix A , optionally computing the left and/or right singular vectors. The SVD is written

$$A = U \Sigma V^H$$

where Σ is an m -by- n matrix which is zero except for its $\min(m, n)$ diagonal elements, U is an m -by- m orthogonal/unitary matrix, and V is an n -by- n orthogonal/unitary matrix. The diagonal elements of Σ are the singular values of A ; they are real and non-negative, and are returned in descending order. The first $\min(m, n)$ columns of U and V are the left and right singular vectors of A .

Note that the routine returns V^H , not V .

Input Parameters

<i>jobu</i>	CHARACTER*1. Must be 'A', 'S', 'O', or 'N'. Specifies options for computing all or part of the matrix U. If <i>jobu</i> = 'A', all m columns of U are returned in the array <i>u</i>; if <i>jobu</i> = 'S', the first $\min(m, n)$ columns of U (the left singular vectors) are returned in the array <i>u</i>; if <i>jobu</i> = 'O', the first $\min(m, n)$ columns of U (the left singular vectors) are overwritten on the array <i>a</i>; if <i>jobu</i> = 'N', no columns of U (no left singular vectors) are computed.
<i>jobvt</i>	CHARACTER*1. Must be 'A', 'S', 'O', or 'N'. Specifies options for computing all or part of the matrix V^H. If <i>jobvt</i> = 'A', all n rows of V^H are returned in the array <i>vt</i>; if <i>jobvt</i> = 'S', the first $\min(m, n)$ rows of V^H (the right singular vectors) are returned in the array <i>vt</i>; if <i>jobvt</i> = 'O', the first $\min(m, n)$ rows of V^H (the right singular vectors) are overwritten on the array <i>a</i>; if <i>jobvt</i> = 'N', no rows of V^H (no right singular vectors) are computed. <i>jobvt</i> and <i>jobu</i> cannot both be 'O'.
<i>m</i>	INTEGER. The number of rows of the matrix A ($m \geq 0$).

<i>n</i>	INTEGER. The number of columns in <i>A</i> ($n \geq 0$).
<i>a, work</i>	REAL for sgesvd DOUBLE PRECISION for dgesvd COMPLEX for cgesvd DOUBLE COMPLEX for zgesvd. Arrays: <i>a(lda,*)</i> is an array containing the <i>m</i> -by- <i>n</i> matrix <i>A</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The first dimension of the array <i>a</i> . Must be at least $\max(1, m)$.
<i>ldu, ldvt</i>	INTEGER. The leading dimensions of the output arrays <i>u</i> and <i>vt</i> , respectively. Constraints: $ldu \geq 1; ldvt < 1$. If <i>jobu</i> = 'S' or 'A', $ldu \geq m$; If <i>jobvt</i> = 'A', $ldvt \geq n$; If <i>jobvt</i> = 'S', $ldvt \geq \min(m, n)$.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> ; $lwork \geq 1$. Constraints: $lwork \geq \max(3 * \min(m, n) + \max(m, n), 5 * \min(m, n))$ (for real flavors); $lwork \geq 2 * \min(m, n) + \max(m, n)$ (for complex flavors). For good performance, <i>lwork</i> must generally be larger. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla. See <i>Application Notes</i> for details.
<i>rwork</i>	REAL for cgesvd DOUBLE PRECISION for zgesvd Workspace array, DIMENSION at least $\max(1, 5 * \min(m, n))$. Used in complex flavors only.

Output Parameters

<i>a</i>	On exit, If <i>jobu</i> = 'O', <i>a</i> is overwritten with the first $\min(m, n)$ columns of U (the left singular vectors, stored columnwise); If <i>jobvt</i> = 'O', <i>a</i> is overwritten with the first $\min(m, n)$ rows of V^H (the right singular vectors, stored rowwise); If <i>jobu</i> ≠ 'O' and <i>jobvt</i> ≠ 'O', the contents of <i>a</i> are destroyed.
<i>s</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION at least $\max(1, \min(m, n))$. Contains the singular values of <i>A</i> sorted so that $s(i) < s(i+1)$.
<i>u, vt</i>	REAL for sgesvd DOUBLE PRECISION for dgesvd COMPLEX for cgesvd DOUBLE COMPLEX for zgesvd. Arrays: <i>u</i>(<i>ldu</i>,*); the second dimension of <i>u</i> must be at least $\max(1, m)$ if <i>jobu</i> = 'A', and at least $\max(1, \min(m, n))$ if <i>jobu</i> = 'S'. If <i>jobu</i> = 'A', <i>u</i> contains the m-by-m orthogonal/unitary matrix U. If <i>jobu</i> = 'S', <i>u</i> contains the first $\min(m, n)$ columns of U (the left singular vectors, stored columnwise). If <i>jobu</i> = 'N' or 'O', <i>u</i> is not referenced. <i>vt</i>(<i>ldvt</i>,*); the second dimension of <i>vt</i> must be at least $\max(1, n)$. If <i>jobvt</i> = 'A', <i>vt</i> contains the n-by-n orthogonal/unitary matrix V_H. If <i>jobvt</i> = 'S', <i>vt</i> contains the first $\min(m, n)$ rows of V^H (the right singular vectors, stored rowwise). If <i>jobvt</i> = 'N' or 'O', <i>vt</i> is not referenced.
<i>work</i>	On exit, if <i>info</i> = 0, then <i>work</i>(1) returns the required minimal size of <i>lwork</i>. For real flavors:

	If $info > 0$, $work(2:\min(m, n))$ contains the unconverged superdiagonal elements of an upper bidiagonal matrix B whose diagonal is in s (not necessarily sorted). B satisfies $A = u * B * vt$, so it has the same singular values as A, and singular vectors related by u and vt.
<i>rwork</i>	On exit (for complex flavors), if $info > 0$, $rwork(1:\min(m, n)-1)$ contains the unconverged superdiagonal elements of an upper bidiagonal matrix B whose diagonal is in s (not necessarily sorted). B satisfies $A = u * B * vt$, so it has the same singular values as A, and singular vectors related by u and vt.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i-th parameter had an illegal value. If $info = i$, then if <code>?bdsqr</code> did not converge, i specifies how many superdiagonals of the intermediate bidiagonal form B did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gesvd` interface are the following:

<i>a</i>	Holds the matrix A of size (m, n) .
<i>s</i>	Holds the vector of length $\min(m, n)$.
<i>u</i>	Holds the matrix U of size $(m, \min(m, n))$.
<i>vt</i>	Holds the matrix VT of size $(\min(m, n), n)$.
<i>ww</i>	Holds the vector of length $\min(m, n)-1$. <i>ww</i> contains the unconverged superdiagonal elements of an upper bidiagonal matrix B whose diagonal is in s (not necessarily sorted). B satisfies $A = U * B * VT$, so it has the same singular values as A , and singular vectors related by U and VT .
<i>job</i>	Must be either 'N', or 'U', or 'V'. The default value is 'N'. If <i>job</i> = 'U', and <i>u</i> is not present, then <i>u</i> is returned in the array <i>a</i>. If <i>job</i> = 'V', and <i>vt</i> is not present, then <i>vt</i> is returned in the array <i>a</i>.

<i>jobu</i>	Restored based on the presence of the argument <i>u</i>, value of <i>job</i> and sizes of arrays <i>u</i> and <i>a</i> as follows: <i>jobu</i> = 'A', if <i>u</i> is present and the number of columns in <i>u</i> is equal to the number of rows in <i>a</i>, <i>jobu</i> = 'S', if <i>u</i> is present and the number of columns in <i>u</i> is not equal to the number of rows in <i>a</i>, <i>jobu</i> = 'O', if <i>u</i> is not present and <i>job</i> is equal to 'U', <i>jobu</i> = 'N', if <i>u</i> is not present and <i>job</i> is not equal to 'U'.
<i>jobvt</i>	Restored based on the presence of the argument <i>vt</i>, value of <i>job</i> and sizes of arrays <i>vt</i> and <i>a</i> as follows: <i>jobvt</i> = 'A', if <i>vt</i> is present and the number of columns in <i>vt</i> is equal to the number of rows in <i>a</i>, <i>jobvt</i> = 'S', if <i>vt</i> is present and the number of columns in <i>vt</i> is not equal to the number of rows in <i>a</i>, <i>jobvt</i> = 'O', if <i>vt</i> is not present and <i>job</i> is equal to 'V', <i>jobvt</i> = 'N', if <i>vt</i> is not present and <i>job</i> is not equal to 'V',

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?gesdd

Computes the singular value decomposition of a general rectangular matrix using a divide and conquer method.

Syntax

FORTRAN 77:

```
call sgesdd(jobz, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, iwork, info)
call dgesdd(jobz, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, iwork, info)
call cgesdd(jobz, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, rwork, iwork, info)
call zgesdd(jobz, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, rwork, iwork, info)
```

Fortran 95:

```
call gesdd(a, s [,u] [,vt] [,jobz] [,info])
```

Description

This routine computes the singular value decomposition (SVD) of a real/complex m -by- n matrix A , optionally computing the left and/or right singular vectors.

If singular vectors are desired, it uses a divide and conquer algorithm. The SVD is written

$$A = U \Sigma V^H,$$

where Σ is an m -by- n matrix which is zero except for its $\min(m, n)$ diagonal elements, U is an m -by- m orthogonal/unitary matrix, and V is an n -by- n orthogonal/unitary matrix. The diagonal elements of Σ are the singular values of A ; they are real and non-negative, and are returned in descending order. The first $\min(m, n)$ columns of U and V are the left and right singular vectors of A .

Note that the routine returns V^H , not V .

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'A', 'S', 'O', or 'N'. Specifies options for computing all or part of the matrix U .

If $jobz = 'A'$, all m columns of U and all n rows of V^T are returned in the arrays u and vt ;
 if $jobz = 'S'$, the first $\min(m, n)$ columns of U and the first $\min(m, n)$ rows of V^T are returned in the arrays u and vt ;
 if $jobz = 'O'$, then
 if $m \geq n$, the first n columns of U are overwritten in the array a and all rows of V^T are returned in the array vt ;
 if $m < n$, all columns of U are returned in the array u and the first m rows of V^T are overwritten in the array a ;
 if $jobz = 'N'$, no columns of U or rows of V^T are computed.

m INTEGER. The number of rows of the matrix A ($m \geq 0$).

n INTEGER. The number of columns in A ($n \geq 0$).

$a, work$ REAL for sgesdd
 DOUBLE PRECISION for dgesdd
 COMPLEX for cgesdd
 DOUBLE COMPLEX for zgesdd.
 Arrays: $a(lda,*)$ is an array containing the m -by- n matrix A .
 The second dimension of a must be at least $\max(1, n)$.
 $work$ is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The first dimension of the array a . Must be at least $\max(1, m)$.

$ldu, ldvt$ INTEGER. The leading dimensions of the output arrays u and vt , respectively.
 Constraints:
 $ldu \geq 1; ldvt \geq 1$.
 If $jobz = 'S'$ or $'A'$, or $jobz = 'O'$ and $m < n$,
 then $ldu \geq m$;
 If $jobz = 'A'$ or $jobz = 'O'$ and $m < n$,
 then $ldvt \geq n$;
 If $jobz = 'S'$, $ldvt \geq \min(m, n)$.

$lwork$ INTEGER.
 The dimension of the array $work$; $lwork \geq 1$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the $work(1)$, and no error message related to $lwork$ is issued by [xerbla](#).

See *Application Notes* for the suggested value of $lwork$.

$rwork$

REAL for cgesdd

DOUBLE PRECISION for zgesdd

Workspace array, DIMENSION at least $\max(1, 5 \cdot \min(m, n))$ if $jobz = 'N'$.

Otherwise, the dimension of $rwork$ must be at least $\max(1, 5 \cdot (\min(m, n))^2 + 7 \cdot \min(m, n))$. This array is used in complex flavors only.

$iwork$

INTEGER. Workspace array, DIMENSION at least $\max(1, 8 \cdot \min(m, n))$.

Output Parameters

a

On exit:

If $jobz = 'O'$, then if $m \geq n$, a is overwritten with the first n columns of U (the left singular vectors, stored columnwise). If $m < n$, a is overwritten with the first m rows of V^T (the right singular vectors, stored rowwise);

If $jobz \neq 'O'$, the contents of a are destroyed.

s

REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

Array, DIMENSION at least $\max(1, \min(m, n))$. Contains the singular values of A sorted so that $s(i) \geq s(i+1)$.

u, vt

REAL for sgesdd

DOUBLE PRECISION for dgesdd

COMPLEX for cgesdd

DOUBLE COMPLEX for zgesdd.

Arrays:

$u(1du,*)$; the second dimension of u must be at least $\max(1, m)$ if $jobz = 'A'$ or $jobz = 'O'$ and $m < n$.

If $jobz = 'S'$, the second dimension of u must be at least $\max(1, \min(m, n))$.

If $jobz = 'A'$ or $jobz = 'O'$ and $m < n$, u contains the m -by- m orthogonal/unitary matrix U .
 If $jobz = 'S'$, u contains the first $\min(m, n)$ columns of U (the left singular vectors, stored columnwise).
 If $jobz = 'O'$ and $m \geq n$, or $jobz = 'N'$, u is not referenced.
 $vt(ldvt,*)$; the second dimension of vt must be at least $\max(1, n)$.
 If $jobz = 'A'$ or $jobz = 'O'$ and $m \geq n$, vt contains the n -by- n orthogonal/unitary matrix V^T .
 If $jobz = 'S'$, vt contains the first $\min(m, n)$ rows of V^T (the right singular vectors, stored rowwise).
 If $jobz = 'O'$ and $m < n$, or $jobz = 'N'$, vt is not referenced.

$work(1)$ On exit, if $info = 0$, then $work(1)$ returns the required minimal size of $lwork$.

$info$ INTEGER.
 If $info = 0$, the execution is successful.
 If $info = -i$, the i -th parameter had an illegal value.
 If $info = i$, then $?bdsdc$ did not converge, updating process failed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gesdd` interface are the following:

a	Holds the matrix A of size (m, n) .
s	Holds the vector of length $\min(m, n)$.
u	Holds the matrix U of size $(m, \min(m, n))$.
vt	Holds the matrix V^T of size $(\min(m, n), n)$.
job	Must be 'N', 'A', 'S', or 'O'. The default value is 'N'.

Application Notes

For real flavors:

If $jobz = 'N'$, $lwork \geq 3 * \min(m, n) + \max(\max(m, n), 6 * \min(m, n))$;

If $jobz = 'O'$, $lwork \geq 3 * (\min(m, n))^2 + \max(\max(m, n), 5 * (\min(m, n))^2 + 4 * \min(m, n))$;

If $jobz = 'S'$ or $'A'$, $lwork \geq 3 * (\min(m, n))^2 + \max(\max(m, n), 4 * (\min(m, n))^2 + 4 * \min(m, n))$

For complex flavors:

If $jobz = 'N'$, $lwork \geq 2 * \min(m, n) + \max(m, n)$;

If $jobz = 'O'$, $lwork \geq 2 * (\min(m, n))^2 + \max(m, n) + 2 * \min(m, n)$;

If $jobz = 'S'$ or $'A'$, $lwork \geq (\min(m, n))^2 + \max(m, n) + 2 * \min(m, n)$;

For good performance, $lwork$ should generally be larger.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?ggsvd

Computes the generalized singular value decomposition of a pair of general rectangular matrices.

Syntax

FORTRAN 77:

```
call sggsvd(jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha, beta, u,
ldu, v, ldv, q, ldq, work, iwork, info)

call dggsvd(jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha, beta, u,
ldu, v, ldv, q, ldq, work, iwork, info)

call cggsvd(jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha, beta, u,
ldu, v, ldv, q, ldq, work, rwork, iwork, info)

call zggsvd(jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha, beta, u,
ldu, v, ldv, q, ldq, work, rwork, iwork, info)
```

Fortran 95:

```
call ggsvd(a, b, alpha, beta [, k] [,l] [,u] [,v] [,q] [,iwork] [,info])
```

Description

This routine computes the generalized singular value decomposition (GSVD) of an m -by- n real/complex matrix A and p -by- n real/complex matrix B :

$$U^H A Q = D_1 \begin{pmatrix} I & R \end{pmatrix}, \quad V^H B Q = D_2 \begin{pmatrix} I & R \end{pmatrix},$$

where U , V and Q are orthogonal/unitary matrices.

Let $k+1$ = the effective numerical rank of the matrix $(A^H, B^H)^H$, then R is a $(k+1)$ -by- $(k+1)$ nonsingular upper triangular matrix, D_1 and D_2 are m -by- $(k+1)$ and p -by- $(k+1)$ "diagonal" matrices and of the following structures, respectively:

If $m-k-1 \geq 0$,

$$D_1 = \begin{matrix} & k & 1 \\ & \begin{pmatrix} I & 0 \\ 0 & C \end{pmatrix} \\ m - k - 1 & \begin{pmatrix} 0 & 0 \end{pmatrix} \end{matrix}$$

$$D_2 = \begin{matrix} & k & 1 \\ p - 1 & \begin{pmatrix} 0 & S \\ 0 & 0 \end{pmatrix} \end{matrix}$$

$$\begin{pmatrix} 0 & R \end{pmatrix} = \begin{matrix} n - k - 1 & k & 1 \\ \begin{pmatrix} 0 & R_{11} & R_{12} \\ 0 & 0 & R_{22} \end{pmatrix} \end{matrix},$$

where

$C = \text{diag}(\alpha(K+1), \dots, \alpha(K+1))$

$S = \text{diag}(\beta(K+1), \dots, \beta(K+1))$

$C^2 + S^2 = I$

R is stored in $a(1:k+1, n-k-1+1:n)$ on exit.

If $m-k-1 < 0$,

$$D_1 = \begin{matrix} & k & m-k & k+l-m \\ m-k & \begin{pmatrix} I & 0 & 0 \\ 0 & C & 0 \end{pmatrix} \end{matrix}$$

$$D_2 = \begin{matrix} & k & m-k & k+l-m \\ m-k & \begin{pmatrix} 0 & S & 0 \\ & 0 & I \end{pmatrix} \\ p-l & \begin{pmatrix} 0 & 0 & 0 \end{pmatrix} \end{matrix}$$

$$(0 \ R) = \begin{matrix} & n-k-l & k & m-k & k+l-m \\ & k & \begin{pmatrix} 0 & R_{11} & R_{12} & R_{13} \\ 0 & 0 & R_{22} & R_{23} \\ 0 & 0 & 0 & R_{33} \end{pmatrix} \end{matrix}$$

where

$C = \text{diag}(\alpha(K+1), \dots, \alpha(m)),$

$S = \text{diag}(\beta(K+1), \dots, \beta(m)),$

$C_2 + S_2 = I$

On exit, $\begin{pmatrix} R_{11} & R_{12} & R_{13} \\ 0 & R_{22} & R_{23} \end{pmatrix}$ is stored in $a(1:m, n-k-l+1:n)$ and R_{33} is stored in $b(m-k+1:l, n+m-k-l+1:n)$.

The routine computes C , S , R , and optionally the orthogonal/unitary transformation matrices U , V and Q .

In particular, if B is an n -by- n nonsingular matrix, then the GSVD of A and B implicitly gives the SVD of A^*B^{-1} :

$$A^*B^{-1} = U^* (D_1 \ D_2^{-1}) * V^H.$$

If $(A^H, B^H)^H$ has orthonormal columns, then the GSVD of A and B is also equal to the CS decomposition of A and B . Furthermore, the GSVD can be used to derive the solution of the eigenvalue problem:

$$A^H * A * X = \lambda * B^H * B * X.$$

Input Parameters

<i>jobu</i>	CHARACTER*1. Must be 'U' or 'N'. If <i>jobu</i> = 'U', orthogonal/unitary matrix U is computed. If <i>jobu</i> = 'N', U is not computed.
<i>jobv</i>	CHARACTER*1. Must be 'V' or 'N'. If <i>jobv</i> = 'V', orthogonal/unitary matrix V is computed. If <i>jobv</i> = 'N', V is not computed.
<i>jobq</i>	CHARACTER*1. Must be 'Q' or 'N'. If <i>jobq</i> = 'Q', orthogonal/unitary matrix Q is computed. If <i>jobq</i> = 'N', Q is not computed.
<i>m</i>	INTEGER. The number of rows of the matrix A ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns of the matrices A and B ($n \geq 0$).
<i>p</i>	INTEGER. The number of rows of the matrix B ($p \geq 0$).
<i>a, b, work</i>	REAL for sggsvd DOUBLE PRECISION for dggsvd COMPLEX for cggsvd DOUBLE COMPLEX for zggsvd. Arrays: <i>a(lda,*)</i> contains the m -by- n matrix A . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>b(lb,*)</i> contains the p -by- n matrix B . The second dimension of <i>b</i> must be at least $\max(1, n)$. <i>work(*)</i> is a workspace array. The dimension of <i>work</i> must be at least $\max(3n, m, p) + n$.

<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; at least $\max(1, m)$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; at least $\max(1, p)$.
<i>ldu</i>	INTEGER. The first dimension of the array <i>u</i> . $ldu \geq \max(1, m)$ if <i>jobu</i> = 'U'; $ldu \geq 1$ otherwise.
<i>ldv</i>	INTEGER. The first dimension of the array <i>v</i> . $ldv \geq \max(1, p)$ if <i>jobv</i> = 'V'; $ldv \geq 1$ otherwise.
<i>ldq</i>	INTEGER. The first dimension of the array <i>q</i> . $ldq \geq \max(1, n)$ if <i>jobq</i> = 'Q'; $ldq \geq 1$ otherwise.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.
<i>rwork</i>	REAL for cggsvd DOUBLE PRECISION for zggsvd. Workspace array, DIMENSION at least $\max(1, 2n)$. Used in complex flavors only.

Output Parameters

<i>k, l</i>	INTEGER. On exit, <i>k</i> and <i>l</i> specify the dimension of the subblocks. The sum $k+l$ is equal to the effective numerical rank of $(A^H, B^H)^H$.
<i>a</i>	On exit, <i>a</i> contains the triangular matrix <i>R</i> or part of <i>R</i> .
<i>b</i>	On exit, <i>b</i> contains part of the triangular matrix <i>R</i> if $m-k-l < 0$.
<i>alpha, beta</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays, DIMENSION at least $\max(1, n)$ each. Contain the generalized singular value pairs of <i>A</i> and <i>B</i> : $\alpha(1:k) = 1$, $\beta(1:k) = 0$, and if $m-k-l \geq 0$, $\alpha(k+1:k+l) = C$, $\beta(k+1:k+l) = S$, or if $m-k-l < 0$, $\alpha(k+1:m) = C$, $\alpha(m+1:k+l) = 0$ $\beta(k+1:m) = S$, $\beta(m+1:k+l) = 1$ and

<i>u, v, q</i>	<pre> alpha(k+l+1:n) = 0 beta(k+l+1:n) = 0. REAL for sggsvd DOUBLE PRECISION for dggsvd COMPLEX for cggsvd DOUBLE COMPLEX for zggsvd. Arrays: <i>u(ldu,*)</i>; the second dimension of <i>u</i> must be at least max(1, <i>m</i>). If <i>jobu</i> = 'U', <i>u</i> contains the <i>m</i>-by-<i>m</i> orthogonal/unitary matrix <i>U</i>. If <i>jobu</i> = 'N', <i>u</i> is not referenced. <i>v(ldv,*)</i>; the second dimension of <i>v</i> must be at least max(1, <i>p</i>). If <i>jobv</i> = 'V', <i>v</i> contains the <i>p</i>-by-<i>p</i> orthogonal/unitary matrix <i>V</i>. If <i>jobv</i> = 'N', <i>v</i> is not referenced. <i>q(ldq,*)</i>; the second dimension of <i>q</i> must be at least max(1, <i>n</i>). If <i>jobq</i> = 'Q', <i>q</i> contains the <i>n</i>-by-<i>n</i> orthogonal/unitary matrix <i>Q</i>. If <i>jobq</i> = 'N', <i>q</i> is not referenced. <i>iwork</i> On exit, <i>iwork</i> stores the sorting information. <i>info</i> INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value. If <i>info</i> = 1, the Jacobi-type procedure failed to converge. For further details, see subroutine ?tgsja. </pre>

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ggsvd` interface are the following:

<i>a</i> <i>b</i>	Holds the matrix <i>A</i> of size (<i>m</i>, <i>n</i>). Holds the matrix <i>B</i> of size (<i>p</i>, <i>n</i>).

<i>alpha</i>	Holds the vector of length (<i>n</i>).
<i>beta</i>	Holds the vector of length (<i>n</i>).
<i>u</i>	Holds the matrix <i>U</i> of size (<i>m</i> , <i>m</i>).
<i>v</i>	Holds the matrix <i>V</i> of size (<i>p</i> , <i>p</i>).
<i>q</i>	Holds the matrix <i>Q</i> of size (<i>n</i> , <i>n</i>).
<i>iwork</i>	Holds the vector of length (<i>n</i>).
<i>jobu</i>	Restored based on the presence of the argument <i>u</i> as follows: <i>jobu</i> = 'U', if <i>u</i> is present, <i>jobu</i> = 'N', if <i>u</i> is omitted.
<i>jobv</i>	Restored based on the presence of the argument <i>v</i> as follows: <i>jobz</i> = 'V', if <i>v</i> is present, <i>jobz</i> = 'N', if <i>v</i> is omitted.
<i>jobq</i>	Restored based on the presence of the argument <i>q</i> as follows: <i>jobz</i> = 'Q', if <i>q</i> is present, <i>jobz</i> = 'N', if <i>q</i> is omitted.

Generalized Symmetric Definite Eigenproblems

This section describes LAPACK driver routines used for solving generalized symmetric definite eigenproblems. See also [computational routines](#) that can be called to solve these problems. [Table 4-13](#) lists all such driver routines for Fortran 77 interface. Respective routine names in Fortran 95 interface are without the first symbol (see [Routine Naming Conventions](#)).

Table 4-13 Driver Routines for Solving Generalized Symmetric Definite Eigenproblems

Routine Name	Operation performed
?sygv/?hegv	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem.
?sygvd/?hegvd	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem. If eigenvectors are desired, it uses a divide and conquer method.
?sygvx/?hegvx	Computes selected eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem.
?spgv/?hpgv	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem with matrices in packed storage.

Routine Name	Operation performed
?spgvd/?hpgvd	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem with matrices in packed storage. If eigenvectors are desired, it uses a divide and conquer method.
?spgvx/?hpgvx	Computes selected eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem with matrices in packed storage.
?sbgv/?hbgv	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem with banded matrices.
?sbgvd/?hbgvd	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem with banded matrices. If eigenvectors are desired, it uses a divide and conquer method.
?sbgvx/?hbgvx	Computes selected eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem with banded matrices.

?sygv

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem.

Syntax

FORTRAN 77:

```
call ssygv(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, info)
call dsygv(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, info)
```

Fortran 95:

```
call sygv(a, b, w [,itype] [,jobz] [,uplo] [,info])
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$A^*x = \lambda^*B^*x, A^*B^*x = \lambda^*x, \text{ or } B^*A^*x = \lambda^*x.$$

Here A and B are assumed to be symmetric and B is also positive definite.

Input Parameters

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A^*x = \lambda^*B^*x$; if <i>itype</i> = 2, the problem type is $A^*B^*x = \lambda^*x$; if <i>itype</i> = 3, the problem type is $B^*A^*x = \lambda^*x$.
<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>a</i> and <i>b</i> store the upper triangles of A and B; If <i>uplo</i> = 'L', arrays <i>a</i> and <i>b</i> store the lower triangles of A and B.
<i>n</i>	INTEGER. The order of the matrices A and B ($n \geq 0$).
<i>a</i> , <i>b</i> , <i>work</i>	REAL for ssygv DOUBLE PRECISION for dsygv. Arrays: <i>a</i>(<i>lda</i>,*) contains the upper or lower triangle of the symmetric matrix A, as specified by <i>uplo</i>. The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>b</i>(<i>ldb</i>,*) contains the upper or lower triangle of the symmetric positive definite matrix B, as specified by <i>uplo</i>. The second dimension of <i>b</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The first dimension of <i>a</i>; at least $\max(1, n)$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i>; at least $\max(1, n)$.

lwork INTEGER.
The dimension of the array *work*;
 $lwork \geq \max(1, 3n-1)$.
If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#).
See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a On exit, if *jobz* = 'V', then if *info* = 0, *a* contains the matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:
if *itype* = 1 or 2, $Z^T * B * Z = I$;
if *itype* = 3, $Z^T * \text{inv}(B) * Z = I$;
If *jobz* = 'N', then on exit the upper triangle (if *uplo* = 'U') or the lower triangle (if *uplo* = 'L') of *A*, including the diagonal, is destroyed.

b On exit, if *info* ≤ *n*, the part of *b* containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization $B = U^T * U$ or $B = L * L^T$.

w REAL for ssygv
DOUBLE PRECISION for dsygv.
Array, DIMENSION at least max(1, *n*).
If *info* = 0, contains the eigenvalues in ascending order.

work(1) On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th argument had an illegal value.
If *info* > 0, spotrf/dpotrf and ssyev/dsyev returned an error code:
If *info* = *i* ≤ *n*, ssyev/dsyev failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;

If $info = n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sygv` interface are the following:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>b</code>	Holds the matrix B of size (n, n) .
<code>w</code>	Holds the vector of length (n) .
<code>itype</code>	Must be 1, 2, or 3. The default value is 1.
<code>jobz</code>	Must be 'N' or 'V'. The default value is 'N'.
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For optimum performance use $lwork \geq (nb+2)*n$, where nb is the blocksize for `ssytrd/dsytrd` returned by `ilaenv`.

If it is not clear how much workspace to supply, use a generous value of $lwork$ (or $liwork$) for the first run or set $lwork = -1$ ($liwork = -1$).

If $lwork$ (or $liwork$) has any of admissible sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array ($work, iwork$) on exit. Use this value ($work(1), iwork(1)$) for subsequent runs.

If $lwork = -1$ ($liwork = -1$), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work, iwork$). This operation is called a workspace query.

Note that if $work$ ($liwork$) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hegv

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem.

Syntax

FORTRAN 77:

```
call chegv(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, rwork, info)
call zhegv(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, rwork, info)
```

Fortran 95:

```
call hegv(a, b, w [,itype] [,jobz] [,uplo] [,info])
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form

$$A^*x = \lambda^*B^*x, \quad A^*B^*x = \lambda^*x, \quad \text{or} \quad B^*A^*x = \lambda^*x.$$

Here A and B are assumed to be Hermitian and B is also positive definite.

Input Parameters

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A^*x = \lambda^*B^*x$; if <i>itype</i> = 2, the problem type is $A^*B^*x = \lambda^*x$; if <i>itype</i> = 3, the problem type is $B^*A^*x = \lambda^*x$.
<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>a</i> and <i>b</i> store the upper triangles of A and B ; If <i>uplo</i> = 'L', arrays <i>a</i> and <i>b</i> store the lower triangles of A and B .

n INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).

a, *b*, *work* COMPLEX for chegv
DOUBLE COMPLEX for zhegv.

Arrays:
a(*lda*,*) contains the upper or lower triangle of the Hermitian matrix *A*, as specified by *uplo*.
 The second dimension of *a* must be at least $\max(1, n)$.
b(*ldb*,*) contains the upper or lower triangle of the Hermitian positive definite matrix *B*, as specified by *uplo*.
 The second dimension of *b* must be at least $\max(1, n)$.
work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The first dimension of *a*; at least $\max(1, n)$.

ldb INTEGER. The first dimension of *b*; at least $\max(1, n)$.

lwork INTEGER.
 The dimension of the array *work*; $lwork \geq \max(1, 2n-1)$.
 If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#).
 See *Application Notes* for the suggested value of *lwork*.

rwork REAL for chegv
DOUBLE PRECISION for zhegv.
 Workspace array, DIMENSION at least $\max(1, 3n-2)$.

Output Parameters

a On exit, if *jobz* = 'V', then if *info* = 0, *a* contains the matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:
 if *itype* = 1 or 2, $Z^H * B * Z = I$;
 if *itype* = 3, $Z^H * \text{inv}(B) * Z = I$;
 If *jobz* = 'N', then on exit the upper triangle (if *uplo* = 'U') or the lower triangle (if *uplo* = 'L') of *A*, including the diagonal, is destroyed.

<i>b</i>	On exit, if <i>info</i> ≤ <i>n</i> , the part of <i>b</i> containing the matrix is overwritten by the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $B = U^H * U$ or $B = L * L^H$.
<i>w</i>	REAL for chegv DOUBLE PRECISION for zhegv. Array, DIMENSION at least max(1, <i>n</i>). If <i>info</i> = 0, contains the eigenvalues in ascending order.
<i>work</i> (1)	On exit, if <i>info</i> = 0, then <i>work</i> (1) returns the required minimal size of <i>lwork</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th argument has an illegal value. If <i>info</i> > 0, cpotrf/zpotrf and cheev/zheev return an error code: If <i>info</i> = <i>i</i> ≤ <i>n</i> , cheev/zheev fails to converge, and <i>i</i> off-diagonal elements of an intermediate tridiagonal do not converge to zero; If <i>info</i> = <i>n</i> + <i>i</i> , for 1 ≤ <i>i</i> ≤ <i>n</i> , then the leading minor of order <i>i</i> of <i>B</i> is not positive-definite. The factorization of <i>B</i> can not be completed and no eigenvalues or eigenvectors are computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hegv` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>n</i>).
<i>w</i>	Holds the vector of length (<i>n</i>).
<i>itype</i>	Must be 1, 2, or 3. The default value is 1.
<i>jobz</i>	Must be 'N' or 'V'. The default value is 'N'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For optimum performance use $lwork \geq (nb+1)*n$, where nb is the blocksize for `chetrd/zhetrd` returned by `ilaenv`.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?sygvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

FORTRAN 77:

```
call ssygvd(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, iwork,
liwork, info)

call dsygvd(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, iwork,
liwork, info)
```

Fortran 95:

```
call sygvd(a, b, w [,itype] [,jobz] [,uplo] [,info])
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$A^*x = \lambda^*B^*x, \quad A^*B^*x = \lambda^*x, \quad \text{or} \quad B^*A^*x = \lambda^*x.$$

Here A and B are assumed to be symmetric and B is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A^*x = \lambda^*B^*x$; if <i>itype</i> = 2, the problem type is $A^*B^*x = \lambda^*x$; if <i>itype</i> = 3, the problem type is $B^*A^*x = \lambda^*x$.
<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>a</i> and <i>b</i> store the upper triangles of A and B ; If <i>uplo</i> = 'L', arrays <i>a</i> and <i>b</i> store the lower triangles of A and B .
<i>n</i>	INTEGER. The order of the matrices A and B ($n \geq 0$).
<i>a, b, work</i>	REAL for ssygvd DOUBLE PRECISION for dsygvd. Arrays: <i>a</i> (<i>lda</i> ,*) contains the upper or lower triangle of the symmetric matrix A , as specified by <i>uplo</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>b</i> (<i>ldb</i> ,*) contains the upper or lower triangle of the symmetric positive definite matrix B , as specified by <i>uplo</i> . The second dimension of <i>b</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; at least $\max(1, n)$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> ; at least $\max(1, n)$.

lwork

INTEGER.

The dimension of the array *work*.

Constraints:

If $n \leq 1$, $lwork \geq 1$;If $jobz = 'N'$ and $n > 1$, $lwork < 2n+1$;If $jobz = 'V'$ and $n > 1$, $lwork < 2n^2+6n+1$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

iwork

INTEGER.

Workspace array, its dimension $\max(1, lwork)$.*liwork*

INTEGER.

The dimension of the array *iwork*.

Constraints:

If $n \leq 1$, $liwork \geq 1$;If $jobz = 'N'$ and $n > 1$, $liwork \geq 1$;If $jobz = 'V'$ and $n > 1$, $liwork \geq 5n+3$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

a

On exit, if $jobz = 'V'$, then if $info = 0$, *a* contains the matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:

if $itype = 1$ or 2 , $Z^T * B * Z = I$;if $itype = 3$, $Z^T * \text{inv}(B) * Z = I$;

If $jobz = 'N'$, then on exit the upper triangle (if $uplo = 'U'$) or the lower triangle (if $uplo = 'L'$) of *A*, including the diagonal, is destroyed.

<i>b</i>	On exit, if <i>info</i> ≤ <i>n</i> , the part of <i>b</i> containing the matrix is overwritten by the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $B = U^T * U$ or $B = L * L^T$.
<i>w</i>	REAL for ssygvd DOUBLE PRECISION for dsygvd. Array, DIMENSION at least max(1, <i>n</i>). If <i>info</i> = 0, contains the eigenvalues in ascending order.
<i>work</i> (1)	On exit, if <i>info</i> = 0, then <i>work</i> (1) returns the required minimal size of <i>lwork</i> .
<i>iwork</i> (1)	On exit, if <i>info</i> = 0, then <i>iwork</i> (1) returns the required minimal size of <i>liwork</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th argument had an illegal value. If <i>info</i> > 0 = <i>i</i> ≤ <i>n</i> , and <i>jobz</i> = 'N', then the algorithm failed to converge; <i>i</i> off-diagonal elements of an intermediate tridiagonal form did not converge to zero; if <i>jobz</i> = 'V', then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns <i>info</i> /(<i>n</i> +1) through mod(<i>info</i> , <i>n</i> +1); If <i>info</i> > 0 = <i>n</i> + <i>i</i> , for 1 ≤ <i>i</i> ≤ <i>n</i> , then the leading minor of order <i>i</i> of <i>B</i> is not positive-definite. The factorization of <i>B</i> could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *sygvd* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>n</i>).
<i>w</i>	Holds the vector of length (<i>n</i>).
<i>itype</i>	Must be 1, 2, or 3. The default value is 1.

`jobz` Must be 'N' or 'V'. The default value is 'N'.
`uplo` Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If it is not clear how much workspace to supply, use a generous value of `lwork` (or `liwork`) for the first run or set `lwork = -1` (`liwork = -1`).

If `lwork` (or `liwork`) has any of admissible sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (`work`, `iwork`) on exit. Use this value (`work(1)`, `iwork(1)`) for subsequent runs.

If `lwork = -1` (`liwork = -1`), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`, `iwork`). This operation is called a workspace query.

Note that if `work` (`liwork`) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hegvd

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

FORTRAN 77:

```
call chegvd(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, rwork,
lrwork, iwork, liwork, info)
```

```
call zhegvd(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, rwork,
lrwork, iwork, liwork, info)
```

Fortran 95:

```
call hegvd(a, b, w [,itype] [,jobz] [,uplo] [,info])
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form

$$A^*x = \lambda^*B^*x, \quad A^*B^*x = \lambda^*x, \quad \text{or} \quad B^*A^*x = \lambda^*x.$$

Here A and B are assumed to be Hermitian and B is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A^*x = \lambda^*B^*x$; if <i>itype</i> = 2, the problem type is $A^*B^*x = \lambda^*x$; if <i>itype</i> = 3, the problem type is $B^*A^*x = \lambda^*x$.
<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>a</i> and <i>b</i> store the upper triangles of A and B; If <i>uplo</i> = 'L', arrays <i>a</i> and <i>b</i> store the lower triangles of A and B.
<i>n</i>	INTEGER. The order of the matrices A and B ($n \geq 0$).
<i>a, b, work</i>	COMPLEX for <code>chegvd</code> DOUBLE COMPLEX for <code>zhegvd</code>. Arrays: <i>a</i>(<i>lda</i>,*) contains the upper or lower triangle of the Hermitian matrix A, as specified by <i>uplo</i>. The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>b</i>(<i>ldb</i>,*) contains the upper or lower triangle of the Hermitian positive definite matrix B, as specified by <i>uplo</i>. The second dimension of <i>b</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The first dimension of <i>a</i>; at least $\max(1, n)$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i>; at least $\max(1, n)$.

lwork INTEGER.
The dimension of the array *work*.
Constraints:
If $n \leq 1$, $lwork \geq 1$;
If $jobz = 'N'$ and $n > 1$, $lwork \geq n+1$;
If $jobz = 'V'$ and $n > 1$, $lwork \geq n^2+2n$.
If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

rwork REAL for `chegvd`
DOUBLE PRECISION for `zhegvd`.
Workspace array, DIMENSION $\max(1, lrwork)$.

lrwork INTEGER.
The dimension of the array *rwork*.
Constraints:
If $n \leq 1$, $lrwork \geq 1$;
If $jobz = 'N'$ and $n > 1$, $lrwork \geq n$;
If $jobz = 'V'$ and $n > 1$, $lrwork \geq 2n^2+5n+1$.
If $lrwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

iwork INTEGER.
Workspace array, DIMENSION $\max(1, liwork)$.

liwork INTEGER.
The dimension of the array *iwork*.
Constraints:
If $n \leq 1$, $liwork \geq 1$;
If $jobz = 'N'$ and $n > 1$, $liwork \geq 1$;

If `jobz = 'V'` and $n > 1$, $liwork \geq 5n + 3$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the `work`, `rwork` and `iwork` arrays, returns these values as the first entries of the `work`, `rwork` and `iwork` arrays, and no error message related to `lwork` or `lrwork` or `liwork` is issued by `xerbla`. See *Application Notes* for details.

Output Parameters

<code>a</code>	On exit, if <code>jobz = 'V'</code> , then if <code>info = 0</code> , <code>a</code> contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows: if <code>itype = 1</code> or <code>2</code> , $Z^H * B * Z = I$; if <code>itype = 3</code> , $Z^H * \text{inv}(B) * Z = I$; If <code>jobz = 'N'</code> , then on exit the upper triangle (if <code>uplo = 'U'</code>) or the lower triangle (if <code>uplo = 'L'</code>) of A , including the diagonal, is destroyed.
<code>b</code>	On exit, if $info \leq n$, the part of <code>b</code> containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization $B = U^H * U$ or $B = L * L^H$.
<code>w</code>	REAL for <code>chegvd</code> DOUBLE PRECISION for <code>zhegvd</code> . Array, DIMENSION at least $\max(1, n)$. If $info = 0$, contains the eigenvalues in ascending order.
<code>work(1)</code>	On exit, if $info = 0$, then <code>work(1)</code> returns the required minimal size of <code>lwork</code> .
<code>rwork(1)</code>	On exit, if $info = 0$, then <code>rwork(1)</code> returns the required minimal size of <code>lrwork</code> .
<code>iwork(1)</code>	On exit, if $info = 0$, then <code>iwork(1)</code> returns the required minimal size of <code>liwork</code> .
<code>info</code>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th argument had an illegal value.

If $info = i$, and $jobz = 'N'$, then the algorithm failed to converge; i off-diagonal elements of an intermediate tridiagonal form did not converge to zero;
 if $info = i$, and $jobz = 'V'$, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns $info/(n+1)$ through $mod(info, n+1)$.
 If $info = n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hegvd` interface are the following:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>b</code>	Holds the matrix B of size (n, n) .
<code>w</code>	Holds the vector of length (n) .
<code>itype</code>	Must be 1, 2, or 3. The default value is 1.
<code>jobz</code>	Must be 'N' or 'V'. The default value is 'N'.
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of `lwork` (`liwork` or `lrwork`) for the first run or set `lwork = -1` (`liwork = -1`, `lrwork = -1`).

If you choose the first option and set any of admissible `lwork` (`liwork` or `lrwork`) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (`work`, `iwork`, `rwork`) on exit. Use this value (`work(1)`, `iwork(1)`, `rwork(1)`) for subsequent runs.

If you set `lwork = -1` (`liwork = -1`, `lrwork = -1`), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`, `iwork`, `rwork`). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*, *lrwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?sygvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem.

Syntax

FORTRAN 77:

```
call ssygvx(itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu, il, iu,
abstol, m, w, z, ldz, work, lwork, iwork, ifail, info)

call dsygvx(itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu, il, iu,
abstol, m, w, z, ldz, work, lwork, iwork, ifail, info)
```

Fortran 95:

```
call sygvx(a, b, w [,itype] [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail]
[,abstol] [,info])
```

Description

This routine computes selected eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$A^*x = \lambda^*B^*x, A^*B^*x = \lambda^*x, \text{ or } B^*A^*x = \lambda^*x.$$

Here *A* and *B* are assumed to be symmetric and *B* is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A^*x = \lambda^*B^*x$; if <i>itype</i> = 2, the problem type is $A^*B^*x = \lambda^*x$; if <i>itype</i> = 3, the problem type is $B^*A^*x = \lambda^*x$.
<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then compute eigenvalues only.
 If *jobz* = 'V', then compute eigenvalues and eigenvectors.

range CHARACTER*1. Must be 'A' or 'V' or 'I'.
 If *range* = 'A', the routine computes all eigenvalues.
 If *range* = 'V', the routine computes eigenvalues *lambda*(*i*) in the half-open interval:
 $vl < \lambda(i) \leq vu$.
 If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.

uplo CHARACTER*1. Must be 'U' or 'L'.
 If *uplo* = 'U', arrays *a* and *b* store the upper triangles of *A* and *B*;
 If *uplo* = 'L', arrays *a* and *b* store the lower triangles of *A* and *B*.

n INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).

a, b, work REAL for ssygvx
 DOUBLE PRECISION for dsygvx.
 Arrays:
a(*lda*,*) contains the upper or lower triangle of the symmetric matrix *A*, as specified by *uplo*.
 The second dimension of *a* must be at least $\max(1, n)$.
b(*ldb*,*) contains the upper or lower triangle of the symmetric positive definite matrix *B*, as specified by *uplo*.
 The second dimension of *b* must be at least $\max(1, n)$.
work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The first dimension of *a*; at least $\max(1, n)$.

ldb INTEGER. The first dimension of *b*; at least $\max(1, n)$.

vl, vu REAL for ssygvx
 DOUBLE PRECISION for dsygvx.
 If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.
 Constraint: $vl < vu$.
 If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, iu INTEGER.
 If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.

	Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$. If $range = 'A'$ or $'V'$, il and iu are not referenced.
<i>abstol</i>	REAL for ssygvx DOUBLE PRECISION for dsygvx. The absolute error tolerance for the eigenvalues. See <i>Application Notes</i> for more information.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i>. Constraints: $ldz \geq 1$; if $jobz = 'V'$, $ldz \geq \max(1, n)$.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i>; $lwork < \max(1, 8n)$. If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla. See <i>Application Notes</i> for the suggested value of <i>lwork</i>.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, 5n)$.

Output Parameters

<i>a</i>	On exit, the upper triangle (if $uplo = 'U'$) or the lower triangle (if $uplo = 'L'$) of <i>A</i> , including the diagonal, is overwritten.
<i>b</i>	On exit, if $info \leq n$, the part of <i>b</i> containing the matrix is overwritten by the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $B = U^T * U$ or $B = L * L^T$.
<i>m</i>	INTEGER. The total number of eigenvalues found, $0 \leq m \leq n$. If $range = 'A'$, $m = n$, and if $range = 'I'$, $m = iu - il + 1$.
<i>w, z</i>	REAL for ssygvx DOUBLE PRECISION for dsygvx. Arrays: <i>w</i>(*), DIMENSION at least $\max(1, n)$.

The first m elements of w contain the selected eigenvalues in ascending order.

$z(ldz,*)$.

The second dimension of z must be at least $\max(1, m)$.

If $jobz = 'V'$, then if $info = 0$, the first m columns of z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i -th column of z holding the eigenvector associated with $w(i)$.

The eigenvectors are normalized as follows:

if $itype = 1$ or 2 , $Z^T B Z = I$;

if $itype = 3$, $Z^T \text{inv}(B) Z = I$;

If $jobz = 'N'$, then z is not referenced.

If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in $ifail$.

Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array z ; if $range = 'V'$, the exact value of m is not known in advance and an upper bound must be used.

$work(1)$

On exit, if $info = 0$, then $work(1)$ returns the required minimal size of $lwork$.

$ifail$

INTEGER.

Array, DIMENSION at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, the first m elements of $ifail$ are zero; if $info > 0$, the $ifail$ contains the indices of the eigenvectors that failed to converge.

If $jobz = 'N'$, then $ifail$ is not referenced.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i th argument had an illegal value.

If $info > 0$, $spotrf/dpotrf$ and $ssyevx/dsyevx$ returned an error code:

If $info = i \leq n$, $ssyevx/dsyevx$ failed to converge, and i eigenvectors failed to converge. Their indices are stored in the array $ifail$;

If $info = n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sygvx` interface are the following:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>b</code>	Holds the matrix B of size (n, n) .
<code>w</code>	Holds the vector of length (n) .
<code>z</code>	Holds the matrix Z of size (n, n) , where the values n and m are significant.
<code>ifail</code>	Holds the vector of length (n) .
<code>itype</code>	Must be 1, 2, or 3. The default value is 1.
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.
<code>vl</code>	Default value for this element is $vl = -HUGE(vl)$.
<code>vu</code>	Default value for this element is $vu = HUGE(vl)$.
<code>il</code>	Default value for this argument is $il = 1$.
<code>iu</code>	Default value for this argument is $iu = n$.
<code>abstol</code>	Default value for this element is $abstol = 0.0\_WP$.
<code>jobz</code>	Restored based on the presence of the argument z as follows: $jobz = 'V'$, if z is present, $jobz = 'N'$, if z is omitted. Note that there will be an error condition if <code>ifail</code> is present and z is omitted.
<code>range</code>	Restored based on the presence of arguments vl , vu , il , iu as follows: $range = 'V'$, if one of or both vl and vu are present, $range = 'I'$, if one of or both il and iu are present, $range = 'A'$, if none of vl , vu , il , iu is present, Note that there will be an error condition if one of or both vl and vu are present and at the same time one of or both il and iu are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a, b]$ of width less than or equal to $abstol + \epsilon \max(|a|, |b|)$, where ϵ is the machine precision.

If $abstol$ is less than or equal to zero, then $\epsilon * ||T||_1$ is used as tolerance, where T is the tridiagonal matrix obtained by reducing A to tridiagonal form. Eigenvalues will be computed most accurately when $abstol$ is set to twice the underflow threshold $2 * \text{lamch}('S')$, not zero.

If this routine returns with $info > 0$, indicating that some eigenvectors did not converge, set $abstol$ to $2 * \text{lamch}('S')$.

For optimum performance use $lwork \geq (nb+3)*n$, where nb is the blocksize for `ssytrd/dsytrd` returned by `ilaenv`.

If it is not clear how much workspace to supply, use a generous value of $lwork$ for the first run, or set $lwork = -1$.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if $lwork$ is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hegvx

Computes selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem.

Syntax

FORTRAN 77:

```
call chegvx(itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu, il, iu,
abstol, m, w, z, ldz, work, lwork, rwork, iwork, ifail, info)
```

```
call zhegvx(itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu, il, iu,
abstol, m, w, z, ldz, work, lwork, rwork, iwork, ifail, info)
```

Fortran 95:

```
call hegvx(a, b, w [,itype] [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail]
[,abstol] [,info])
```

Description

This routine computes selected eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form

$$A^*x = \lambda^*B^*x, \quad A^*B^*x = \lambda^*x, \quad \text{or} \quad B^*A^*x = \lambda^*x.$$

Here A and B are assumed to be Hermitian and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A^*x = \lambda^*B^*x$; if <i>itype</i> = 2, the problem type is $A^*B^*x = \lambda^*x$; if <i>itype</i> = 3, the problem type is $B^*A^*x = \lambda^*x$.
<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'.

If *range* = 'A', the routine computes all eigenvalues.
 If *range* = 'V', the routine computes eigenvalues *lambda(i)* in the half-open interval:
 $vl < \lambda(i) \leq vu$.
 If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.

uplo CHARACTER*1. Must be 'U' or 'L'.
 If *uplo* = 'U', arrays *a* and *b* store the upper triangles of *A* and *B*;
 If *uplo* = 'L', arrays *a* and *b* store the lower triangles of *A* and *B*.

n INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).

a, b, work COMPLEX for *chegvx*
 DOUBLE COMPLEX for *zhegvx*.
Arrays:
a(lda,)* contains the upper or lower triangle of the Hermitian matrix *A*, as specified by *uplo*.
 The second dimension of *a* must be at least $\max(1, n)$.
b(ldb,)* contains the upper or lower triangle of the Hermitian positive definite matrix *B*, as specified by *uplo*.
 The second dimension of *b* must be at least $\max(1, n)$.
work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The first dimension of *a*; at least $\max(1, n)$.

ldb INTEGER. The first dimension of *b*; at least $\max(1, n)$.

vl, vu REAL for *chegvx*
 DOUBLE PRECISION for *zhegvx*.
 If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.
 Constraint: $vl < vu$.
 If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, iu INTEGER.
 If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.
 Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$.

	If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.
<i>abstol</i>	REAL for chegvx DOUBLE PRECISION for zhegvx. The absolute error tolerance for the eigenvalues. See <i>Application Notes</i> for more information.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i> . Constraints: $ldz \geq 1$; if <i>jobz</i> = 'V', $ldz \geq \max(1, n)$.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> ; $lwork \geq \max(1, 2n)$. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla. See <i>Application Notes</i> for the suggested value of <i>lwork</i> .
<i>rwork</i>	REAL for chegvx DOUBLE PRECISION for zhegvx. Workspace array, DIMENSION at least $\max(1, 7n)$.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, 5n)$.

Output Parameters

<i>a</i>	On exit, the upper triangle (if <i>uplo</i> = 'U') or the lower triangle (if <i>uplo</i> = 'L') of <i>A</i> , including the diagonal, is overwritten.
<i>b</i>	On exit, if <i>info</i> ≤ <i>n</i> , the part of <i>b</i> containing the matrix is overwritten by the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $B = U^H * U$ or $B = L * L^H$.
<i>m</i>	INTEGER. The total number of eigenvalues found, $0 \leq m \leq n$. If <i>range</i> = 'A', $m = n$, and if <i>range</i> = 'I', $m = iu - il + 1$.
<i>w</i>	REAL for chegvx DOUBLE PRECISION for zhegvx. Array, DIMENSION at least $\max(1, n)$.

The first m elements of w contain the selected eigenvalues in ascending order.

z
 COMPLEX for chegvx
 DOUBLE COMPLEX for zhegvx.
 Array $z(ldz,*)$. The second dimension of z must be at least $\max(1, m)$.
 If $jobz = 'V'$, then if $info = 0$, the first m columns of z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i -th column of z holding the eigenvector associated with $w(i)$. The eigenvectors are normalized as follows:
 if $itype = 1$ or 2 , $Z^H * B * Z = I$;
 if $itype = 3$, $Z^H * \text{inv}(B) * Z = I$;
 If $jobz = 'N'$, then z is not referenced.
 If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in $ifail$.
 Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array z ; if $range = 'V'$, the exact value of m is not known in advance and an upper bound must be used.

$work(1)$
 On exit, if $info = 0$, then $work(1)$ returns the required minimal size of $lwork$.

$ifail$
 INTEGER.
 Array, DIMENSION at least $\max(1, n)$.
 If $jobz = 'V'$, then if $info = 0$, the first m elements of $ifail$ are zero; if $info > 0$, the $ifail$ contains the indices of the eigenvectors that failed to converge.
 If $jobz = 'N'$, then $ifail$ is not referenced.

$info$
 INTEGER.
 If $info = 0$, the execution is successful.
 If $info = -i$, the i th argument had an illegal value.
 If $info > 0$, cpotrf/zpotrf and cheevx/zheevx returned an error code:
 If $info = i \leq n$, cheevx/zheevx failed to converge, and i eigenvectors failed to converge. Their indices are stored in the array $ifail$;

If $info = n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hegvx` interface are the following:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>b</i>	Holds the matrix B of size (n, n) .
<i>w</i>	Holds the vector of length (n) .
<i>z</i>	Holds the matrix Z of size (n, n) , where the values n and m are significant.
<i>ifail</i>	Holds the vector of length (n) .
<i>itype</i>	Must be 1, 2, or 3. The default value is 1.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>vl</i>	Default value for this element is $vl = -HUGE(vl)$.
<i>vu</i>	Default value for this element is $vu = HUGE(vl)$.
<i>il</i>	Default value for this argument is $il = 1$.
<i>iu</i>	Default value for this argument is $iu = n$.
<i>abstol</i>	Default value for this element is $abstol = 0.0\_WP$.
<i>jobz</i>	Restored based on the presence of the argument z as follows: $jobz = 'V'$, if z is present, $jobz = 'N'$, if z is omitted. Note that there will be an error condition if <i>ifail</i> is present and z is omitted.
<i>range</i>	Restored based on the presence of arguments vl , vu , il , iu as follows: $range = 'V'$, if one of or both vl and vu are present, $range = 'I'$, if one of or both il and iu are present, $range = 'A'$, if none of vl , vu , il , iu is present, Note that there will be an error condition if one of or both vl and vu are present and at the same time one of or both il and iu are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon \max(|a|, |b|)$, where ϵ is the machine precision.

If $abstol$ is less than or equal to zero, then $\epsilon * ||T||_1$ will be used in its place, where T is the tridiagonal matrix obtained by reducing A to tridiagonal form. Eigenvalues will be computed most accurately when $abstol$ is set to twice the underflow threshold $2 * \text{lamch}('S')$, not zero.

If this routine returns with $info > 0$, indicating that some eigenvectors did not converge, try setting $abstol$ to $2 * \text{lamch}('S')$.

For optimum performance use $lwork \geq (nb+1)*n$, where nb is the blocksize for `chetrd/zhetrd` returned by `ilaenv`.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?spgv

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with matrices in packed storage.

Syntax

FORTRAN 77:

```
call sspgv(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, info)
call dspgv(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, info)
```

Fortran 95:

```
call spgv(a, b, w [,itype] [,uplo] [,z] [,info])
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$A*x = \lambda*B*x, A*B*x = \lambda*x, \text{ or } B*A*x = \lambda*x.$$

Here A and B are assumed to be symmetric, stored in packed format, and B is also positive definite.

Input Parameters

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A*x = \lambda*B*x$; if <i>itype</i> = 2, the problem type is $A*B*x = \lambda*x$; if <i>itype</i> = 3, the problem type is $B*A*x = \lambda*x$.
<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ap</i> and <i>bp</i> store the upper triangles of A and B ;

If *uplo* = 'L', arrays *ap* and *bp* store the lower triangles of *A* and *B*.

n INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).

ap, *bp*, *work* REAL for sspgv
DOUBLE PRECISION for dspgv.

Arrays:
ap(*) contains the packed upper or lower triangle of the symmetric matrix *A*, as specified by *uplo*.
 The dimension of *ap* must be at least $\max(1, n*(n+1)/2)$.
bp(*) contains the packed upper or lower triangle of the symmetric matrix *B*, as specified by *uplo*.
 The dimension of *bp* must be at least $\max(1, n*(n+1)/2)$.
work(*) is a workspace array, DIMENSION at least $\max(1, 3n)$.

ldz INTEGER. The leading dimension of the output array *z*; *ldz* ≥ 1 . If *jobz* = 'V', *ldz* $\geq \max(1, n)$.

Output Parameters

ap On exit, the contents of *ap* are overwritten.

bp On exit, contains the triangular factor *U* or *L* from the Cholesky factorization $B = U^T * U$ or $B = L * L^T$, in the same storage format as *B*.

w, *z* REAL for sspgv
DOUBLE PRECISION for dspgv.

Arrays:
w(*), DIMENSION at least $\max(1, n)$.
 If *info* = 0, contains the eigenvalues in ascending order.
z(*ldz*,*).
 The second dimension of *z* must be at least $\max(1, n)$.
 If *jobz* = 'V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:
 if *itype* = 1 or 2, $Z^T * B * Z = I$;
 if *itype* = 3, $Z^T * \text{inv}(B) * Z = I$;
 If *jobz* = 'N', then *z* is not referenced.

info INTEGER.

If $info = 0$, the execution is successful.
 If $info = -i$, the i -th argument had an illegal value.
 If $info > 0$, `sppttrf/dppttrf` and `sspev/dspev` returned an error code:
 If $info = i \leq n$, `sspev/dspev` failed to converge, and i off-diagonal elements of an intermediate tridiagonal did not converge to zero;
 If $info = n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `spgv` interface are the following:

a	Stands for argument a_p in Fortran 77 interface. Holds the array A of size $(n * (n+1) / 2)$.
b	Stands for argument b_p in Fortran 77 interface. Holds the array B of size $(n * (n+1) / 2)$.
w	Holds the vector of length (n) .
z	Holds the matrix Z of size (n, n) .
$itype$	Must be 1, 2, or 3. The default value is 1.
$uplo$	Must be 'U' or 'L'. The default value is 'U'.
$jobz$	Restored based on the presence of the argument z as follows: $jobz = 'V'$, if z is present, $jobz = 'N'$, if z is omitted.

?hpgv

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem with matrices in packed storage.

Syntax

FORTRAN 77:

```
call chpgv(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, rwork, info)
call zhpv(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, rwork, info)
```

Fortran 95:

```
call hpgv(a, b, w [,itype] [,uplo] [,z] [,info])
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form

$$A^*x = \lambda^*B^*x, A^*B^*x = \lambda^*x, \text{ or } B^*A^*x = \lambda^*x.$$

Here A and B are assumed to be Hermitian, stored in packed format, and B is also positive definite.

Input Parameters

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A^*x = \lambda^*B^*x$; if <i>itype</i> = 2, the problem type is $A^*B^*x = \lambda^*x$; if <i>itype</i> = 3, the problem type is $B^*A^*x = \lambda^*x$.
<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ap</i> and <i>bp</i> store the upper triangles of A and B ;

	If <i>uplo</i> = 'L', arrays <i>ap</i> and <i>bp</i> store the lower triangles of <i>A</i> and <i>B</i> .
<i>n</i>	INTEGER. The order of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>ap</i> , <i>bp</i> , <i>work</i>	COMPLEX for <i>chpgv</i> DOUBLE COMPLEX for <i>zhpgv</i> . Arrays: <i>ap</i> (*) contains the packed upper or lower triangle of the Hermitian matrix <i>A</i> , as specified by <i>uplo</i> . The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$. <i>bp</i> (*) contains the packed upper or lower triangle of the Hermitian matrix <i>B</i> , as specified by <i>uplo</i> . The dimension of <i>bp</i> must be at least $\max(1, n*(n+1)/2)$. <i>work</i> (*) is a workspace array, DIMENSION at least $\max(1, 2n-1)$.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i> ; $ldz \geq 1$. If <i>jobz</i> = 'V', $ldz \geq \max(1, n)$.
<i>rwork</i>	REAL for <i>chpgv</i> DOUBLE PRECISION for <i>zhpgv</i> . Workspace array, DIMENSION at least $\max(1, 3n-2)$.

Output Parameters

<i>ap</i>	On exit, the contents of <i>ap</i> are overwritten.
<i>bp</i>	On exit, contains the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $B = U^H U$ or $B = L^* L^H$, in the same storage format as <i>B</i> .
<i>w</i>	REAL for <i>chpgv</i> DOUBLE PRECISION for <i>zhpgv</i> . Array, DIMENSION at least $\max(1, n)$. If <i>info</i> = 0, contains the eigenvalues in ascending order.
<i>z</i>	COMPLEX for <i>chpgv</i> DOUBLE COMPLEX for <i>zhpgv</i> . Array <i>z</i> (<i>ldz</i> ,*). The second dimension of <i>z</i> must be at least $\max(1, n)$. If <i>jobz</i> = 'V', then if <i>info</i> = 0, <i>z</i> contains the matrix <i>Z</i> of eigenvectors. The eigenvectors are normalized as follows:

if $itype = 1$ or 2 , $Z^H * B * Z = I$;
 if $itype = 3$, $Z^H * \text{inv}(B) * Z = I$;
 If $jobz = 'N'$, then z is not referenced.
info INTEGER.
 If $info = 0$, the execution is successful.
 If $info = -i$, the i -th argument had an illegal value.
 If $info > 0$, `cpptrf/zpptrf` and `chpev/zhpev` returned an error code:
 If $info = i \leq n$, `chpev/zhpev` failed to converge, and i off-diagonal elements of an intermediate tridiagonal did not converge to zero;
 If $info = n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hpgv` interface are the following:

<i>a</i>	Stands for argument <i>ap</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(n * (n+1) / 2)$.
<i>b</i>	Stands for argument <i>bp</i> in Fortran 77 interface. Holds the array <i>B</i> of size $(n * (n+1) / 2)$.
<i>w</i>	Holds the vector of length (n) .
<i>z</i>	Holds the matrix <i>Z</i> of size (n, n) .
<i>itype</i>	Must be 1, 2, or 3. The default value is 1.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: $jobz = 'V'$, if <i>z</i> is present, $jobz = 'N'$, if <i>z</i> is omitted.

?spgvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with matrices in packed storage. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

FORTRAN 77:

```
call sspgvd(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, lwork, iwork,
liwork, info)

call dspgvd(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, lwork, iwork,
liwork, info)
```

Fortran 95:

```
call spgvd(a, b, w [,itype] [,uplo] [,z] [,info])
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$A*x = \lambda*B*x, \quad A*B*x = \lambda*x, \quad \text{or} \quad B*A*x = \lambda*x.$$

Here A and B are assumed to be symmetric, stored in packed format, and B is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A*x = \lambda*B*x$; if <i>itype</i> = 2, the problem type is $A*B*x = \lambda*x$; if <i>itype</i> = 3, the problem type is $B*A*x = \lambda*x$.
<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ap</i> and <i>bp</i> store the upper triangles of <i>A</i> and <i>B</i>; If <i>uplo</i> = 'L', arrays <i>ap</i> and <i>bp</i> store the lower triangles of <i>A</i> and <i>B</i>.
<i>n</i>	INTEGER. The order of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>ap, bp, work</i>	REAL for sspgvd DOUBLE PRECISION for dspgvd. Arrays: <i>ap</i>(*) contains the packed upper or lower triangle of the symmetric matrix <i>A</i>, as specified by <i>uplo</i>. The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$. <i>bp</i>(*) contains the packed upper or lower triangle of the symmetric matrix <i>B</i>, as specified by <i>uplo</i>. The dimension of <i>bp</i> must be at least $\max(1, n*(n+1)/2)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i>; $ldz \geq 1$. If <i>jobz</i> = 'V', $ldz \geq \max(1, n)$.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i>. Constraints: If $n \leq 1$, $lwork \geq 1$; If <i>jobz</i> = 'N' and $n > 1$, $lwork \geq 2n$; If <i>jobz</i> = 'V' and $n > 1$, $lwork \geq 2n^2 + 6n + 1$. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the required sizes of the <i>work</i> and <i>iwork</i> arrays, returns these values as the first entries of the <i>work</i> and <i>iwork</i> arrays, and no error message related to <i>lwork</i> or <i>liwork</i> is issued by xerbla. See <i>Application Notes</i> for details.
<i>iwork</i>	INTEGER. Workspace array, its dimension $\max(1, lwork)$.
<i>liwork</i>	INTEGER. The dimension of the array <i>iwork</i>. Constraints:

If $n \leq 1$, $liwork \geq 1$;
 If $jobz = 'N'$ and $n > 1$, $liwork \geq 1$;
 If $jobz = 'V'$ and $n > 1$, $liwork \geq 5n + 3$.
 If $liwork = -1$, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

<i>ap</i>	On exit, the contents of <i>ap</i> are overwritten.
<i>bp</i>	On exit, contains the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $B = U^T * U$ or $B = L * L^T$, in the same storage format as <i>B</i> .
<i>w, z</i>	REAL for sspgv DOUBLE PRECISION for dspgv. Arrays: $w(*)$, DIMENSION at least $\max(1, n)$. If $info = 0$, contains the eigenvalues in ascending order. $z(ldz,*)$. The second dimension of <i>z</i> must be at least $\max(1, n)$. If $jobz = 'V'$, then if $info = 0$, <i>z</i> contains the matrix <i>Z</i> of eigenvectors. The eigenvectors are normalized as follows: if $itype = 1$ or 2 , $Z^T * B * Z = I$; if $itype = 3$, $Z^T * \text{inv}(B) * Z = I$; If $jobz = 'N'$, then <i>z</i> is not referenced.
<i>work</i> (1)	On exit, if $info = 0$, then <i>work</i> (1) returns the required minimal size of <i>lwork</i> .
<i>iwork</i> (1)	On exit, if $info = 0$, then <i>iwork</i> (1) returns the required minimal size of <i>liwork</i> .
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the <i>i</i> -th argument had an illegal value.

If $info > 0$, `sppttrf/dppttrf` and `sspevd/dspevd` returned an error code:

If $info = i \leq n$, `sspevd/dspevd` failed to converge, and i off-diagonal elements of an intermediate tridiagonal did not converge to zero;

If $info = n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `spgvd` interface are the following:

a	Stands for argument ap in Fortran 77 interface. Holds the array A of size $(n*(n+1)/2)$.
b	Stands for argument bp in Fortran 77 interface. Holds the array B of size $(n*(n+1)/2)$.
w	Holds the vector of length (n) .
z	Holds the matrix Z of size (n, n) .
$itype$	Must be 1, 2, or 3. The default value is 1.
$uplo$	Must be 'U' or 'L'. The default value is 'U'.
$jobz$	Restored based on the presence of the argument z as follows: $jobz = 'V'$, if z is present, $jobz = 'N'$, if z is omitted.

Application Notes

If it is not clear how much workspace to supply, use a generous value of $lwork$ (or $liwork$) for the first run, or set $lwork = -1$ ($liwork = -1$).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If *lwork* = -1 (*liwork* = -1), then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *lwork* (*liwork*) is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hpgvd

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem with matrices in packed storage. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

FORTRAN 77:

```
call chpgvd(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, lwork, rwork,
lrwork, iwork, liwork, info)

call zhpgvd(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, lwork, rwork,
lrwork, iwork, liwork, info)
```

Fortran 95:

```
call hpgvd(a, b, w [,itype] [,uplo] [,z] [,info])
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form

$$A^*x = \lambda^*B^*x, A^*B^*x = \lambda^*x, \text{ or } B^*A^*x = \lambda^*x.$$

Here *A* and *B* are assumed to be Hermitian, stored in packed format, and *B* is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A*x = \lambda*B*x$; if <i>itype</i> = 2, the problem type is $A*B*x = \lambda*x$; if <i>itype</i> = 3, the problem type is $B*A*x = \lambda*x$.
<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ap</i> and <i>bp</i> store the upper triangles of <i>A</i> and <i>B</i>; If <i>uplo</i> = 'L', arrays <i>ap</i> and <i>bp</i> store the lower triangles of <i>A</i> and <i>B</i>.
<i>n</i>	INTEGER. The order of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>ap, bp, work</i>	COMPLEX for <i>chpgvd</i> DOUBLE COMPLEX for <i>zhpgvd</i>. Arrays: <i>ap</i>(*) contains the packed upper or lower triangle of the Hermitian matrix <i>A</i>, as specified by <i>uplo</i>. The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$. <i>bp</i>(*) contains the packed upper or lower triangle of the Hermitian matrix <i>B</i>, as specified by <i>uplo</i>. The dimension of <i>bp</i> must be at least $\max(1, n*(n+1)/2)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i>; <i>ldz</i> ≥ 1. If <i>jobz</i> = 'V', <i>ldz</i> $\geq \max(1, n)$.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i>. Constraints: If $n \leq 1$, <i>lwork</i> ≥ 1; If <i>jobz</i> = 'N' and $n > 1$, <i>lwork</i> $\geq n$; If <i>jobz</i> = 'V' and $n > 1$, <i>lwork</i> $\geq 2n$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

rwork

REAL for *chpgvd*

DOUBLE PRECISION for *zhpgvd*.

Workspace array, its dimension $\max(1, lrwork)$.

lrwork

INTEGER.

The dimension of the array *rwork*.

Constraints:

If $n \leq 1$, $lrwork \geq 1$;

If $jobz = 'N'$ and $n > 1$, $lrwork \geq n$;

If $jobz = 'V'$ and $n > 1$, $lrwork \geq 2n^2 + 5n + 1$.

If $lrwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

iwork

INTEGER.

Workspace array, its dimension $\max(1, liwork)$.

liwork

INTEGER.

The dimension of the array *iwork*.

Constraints:

If $n \leq 1$, $liwork \geq 1$;

If $jobz = 'N'$ and $n > 1$, $liwork \geq 1$;

If $jobz = 'V'$ and $n > 1$, $liwork \geq 5n + 3$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

<i>ap</i>	On exit, the contents of <i>ap</i> are overwritten.
<i>bp</i>	On exit, contains the triangular factor U or L from the Cholesky factorization $B = U^H * U$ or $B = L * L^H$, in the same storage format as B .
<i>w</i>	REAL for <code>chpgvd</code> DOUBLE PRECISION for <code>zhpgvd</code> . Array, DIMENSION at least $\max(1, n)$. If <i>info</i> = 0, contains the eigenvalues in ascending order.
<i>z</i>	COMPLEX for <code>chpgvd</code> DOUBLE COMPLEX for <code>zhpgvd</code> . Array <i>z</i> (<i>ldz</i> ,*). The second dimension of <i>z</i> must be at least $\max(1, n)$. If <i>jobz</i> = 'V', then if <i>info</i> = 0, <i>z</i> contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows: if <i>itype</i> = 1 or 2, $Z^H * B * Z = I$; if <i>itype</i> = 3, $Z^H * \text{inv}(B) * Z = I$; If <i>jobz</i> = 'N', then <i>z</i> is not referenced.
<i>work</i> (1)	On exit, if <i>info</i> = 0, then <i>work</i> (1) returns the required minimal size of <i>lwork</i> .
<i>rwork</i> (1)	On exit, if <i>info</i> = 0, then <i>rwork</i> (1) returns the required minimal size of <i>lrwork</i> .
<i>iwork</i> (1)	On exit, if <i>info</i> = 0, then <i>iwork</i> (1) returns the required minimal size of <i>liwork</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th argument had an illegal value. If <i>info</i> > 0, <code>cpptrf/zpptrf</code> and <code>chpevd/zhpevd</code> returned an error code: If <i>info</i> = <i>i</i> ≤ <i>n</i> , <code>chpevd/zhpevd</code> failed to converge, and <i>i</i> off-diagonal elements of an intermediate tridiagonal did not converge to zero;

If $info = n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hpgvd` interface are the following:

<i>a</i>	Stands for argument <i>ap</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(n*(n+1)/2)$.
<i>b</i>	Stands for argument <i>bp</i> in Fortran 77 interface. Holds the array <i>B</i> of size $(n*(n+1)/2)$.
<i>w</i>	Holds the vector of length (n) .
<i>z</i>	Holds the matrix <i>Z</i> of size (n, n) .
<i>itype</i>	Must be 1, 2, or 3. The default value is 1.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: $jobz = 'V'$, if <i>z</i> is present, $jobz = 'N'$, if <i>z</i> is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* (*liwork* or *lrwork*) for the first run or set $lwork = -1$ ($liwork = -1$, $lrwork = -1$).

If you choose the first option and set any of admissible *lwork* (*liwork* or *lrwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*) on exit. Use this value (*work*(1), *iwork*(1), *rwork*(1)) for subsequent runs.

If you set $lwork = -1$ ($liwork = -1$, $lrwork = -1$), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*, *lrwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?spgvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with matrices in packed storage.

Syntax

FORTRAN 77:

```
call sspgvx(itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu, abstol, m,
w, z, ldz, work, iwork, ifail, info)

call dspgvx(itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu, abstol, m,
w, z, ldz, work, iwork, ifail, info)
```

Fortran 95:

```
call spgvx(a, b, w [,itype] [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail]
[,abstol] [,info])
```

Description

This routine computes selected eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$A^*x = \lambda^*B^*x, A^*B^*x = \lambda^*x, \text{ or } B^*A^*x = \lambda^*x.$$

Here *A* and *B* are assumed to be symmetric, stored in packed format, and *B* is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

itype INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved:
 if *itype* = 1, the problem type is $A^*x = \lambda^*B^*x$;
 if *itype* = 2, the problem type is $A^*B^*x = \lambda^*x$;
 if *itype* = 3, the problem type is $B^*A^*x = \lambda^*x$.

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues $\lambda(i)$ in the half-open interval: $vl < \lambda(i) \leq vu.$ If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i>.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ap</i> and <i>bp</i> store the upper triangles of <i>A</i> and <i>B</i>; If <i>uplo</i> = 'L', arrays <i>ap</i> and <i>bp</i> store the lower triangles of <i>A</i> and <i>B</i>.
<i>n</i>	INTEGER. The order of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>ap, bp, work</i>	REAL for sspgvx DOUBLE PRECISION for dspgvx. Arrays: <i>ap</i>(*) contains the packed upper or lower triangle of the symmetric matrix <i>A</i>, as specified by <i>uplo</i>. The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$. <i>bp</i>(*) contains the packed upper or lower triangle of the symmetric matrix <i>B</i>, as specified by <i>uplo</i>. The dimension of <i>bp</i> must be at least $\max(1, n*(n+1)/2)$. <i>work</i>(*) is a workspace array, DIMENSION at least $\max(1, 8n)$.
<i>vl, vu</i>	REAL for sspgvx DOUBLE PRECISION for dspgvx. If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: $vl < vu$. If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il, iu</i>	INTEGER. If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$.
 If $range = 'A'$ or $'V'$, il and iu are not referenced.

abstol REAL for sspgvx
 DOUBLE PRECISION for dspgvx.
 The absolute error tolerance for the eigenvalues. See *Application Notes* for more information.

ldz INTEGER. The leading dimension of the output array z .
 Constraints:
 $ldz \geq 1$; if $jobz = 'V'$, $ldz \geq \max(1, n)$.

iwork INTEGER.
 Workspace array, DIMENSION at least $\max(1, 5n)$.

Output Parameters

ap On exit, the contents of ap are overwritten.

bp On exit, contains the triangular factor U or L from the Cholesky factorization $B = U^T * U$ or $B = L * L^T$, in the same storage format as B .

m INTEGER. The total number of eigenvalues found,
 $0 \leq m \leq n$. If $range = 'A'$, $m = n$, and if $range = 'I'$, $m = iu - il + 1$.

w, z REAL for sspgvx
 DOUBLE PRECISION for dspgvx.
 Arrays:
 $w(*)$, DIMENSION at least $\max(1, n)$.
 If $info = 0$, contains the eigenvalues in ascending order.
 $z(ldz, *)$.
 The second dimension of z must be at least $\max(1, n)$.
 If $jobz = 'V'$, then if $info = 0$, the first m columns of z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i -th column of z holding the eigenvector associated with $w(i)$.
 The eigenvectors are normalized as follows:
 if $itype = 1$ or 2 , $Z^T * B * Z = I$;
 if $itype = 3$, $Z^T * \text{inv}(B) * Z = I$;

If $jobz = 'N'$, then z is not referenced.

If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in $ifail$.

Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array z ; if $range = 'V'$, the exact value of m is not known in advance and an upper bound must be used.

ifail

INTEGER.

Array, DIMENSION at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, the first m elements of *ifail* are zero; if $info > 0$, the *ifail* contains the indices of the eigenvectors that failed to converge.

If $jobz = 'N'$, then *ifail* is not referenced.

info

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th argument had an illegal value.

If $info > 0$, *spptrf*/*dpptf* and *sspevx*/*dspevx* returned an error code:

If $info = i \leq n$, *sspevx*/*dspevx* failed to converge, and i eigenvectors failed to converge. Their indices are stored in the array *ifail*;

If $info = n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *spgvx* interface are the following:

<i>a</i>	Stands for argument a_p in Fortran 77 interface. Holds the array A of size $(n * (n+1) / 2)$.
<i>b</i>	Stands for argument b_p in Fortran 77 interface. Holds the array B of size $(n * (n+1) / 2)$.

<i>w</i>	Holds the vector of length (<i>n</i>).
<i>z</i>	Holds the matrix <i>z</i> of size (<i>n</i> , <i>n</i>), where the values <i>n</i> and <i>m</i> are significant.
<i>ifail</i>	Holds the vector of length (<i>n</i>).
<i>itype</i>	Must be 1, 2, or 3. The default value is 1.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>vl</i>	Default value for this element is <i>vl</i> = -HUGE(<i>vl</i>).
<i>vu</i>	Default value for this element is <i>vu</i> = HUGE(<i>vl</i>).
<i>il</i>	Default value for this argument is <i>il</i> = 1.
<i>iu</i>	Default value for this argument is <i>iu</i> = <i>n</i> .
<i>abstol</i>	Default value for this element is <i>abstol</i> = 0.0_WP.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted. Note that there will be an error condition if <i>ifail</i> is present and <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: <i>range</i> = 'V', if one of or both <i>vl</i> and <i>vu</i> are present, <i>range</i> = 'I', if one of or both <i>il</i> and <i>iu</i> are present, <i>range</i> = 'A', if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present, Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon \max(|a|, |b|)$, where ϵ is the machine precision.

If *abstol* is less than or equal to zero, then $\epsilon \|T\|_1$ is used instead, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues are computed most accurately when *abstol* is set to twice the underflow threshold $2 * \text{lamch}('S')$, not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, set *abstol* to $2 * \text{lamch}('S')$.

?hpgvx

Computes selected eigenvalues and, optionally, eigenvectors of a generalized Hermitian definite eigenproblem with matrices in packed storage.

Syntax

FORTRAN 77:

```
call chpgvx(itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu, abstol, m,
w, z, ldz, work, rwork, iwork, ifail, info)
```

```
call zhpgrvx(itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu, abstol, m,
w, z, ldz, work, rwork, iwork, ifail, info)
```

Fortran 95:

```
call hpgvx(a, b, w [,itype] [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail]
[,abstol] [,info])
```

Description

This routine computes selected eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form

$$A^*x = \lambda^*B^*x, A^*B^*x = \lambda^*x, \text{ or } B^*A^*x = \lambda^*x.$$

Here A and B are assumed to be Hermitian, stored in packed format, and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A^*x = \lambda^*B^*x$; if <i>itype</i> = 2, the problem type is $A^*B^*x = \lambda^*x$; if <i>itype</i> = 3, the problem type is $B^*A^*x = \lambda^*x$.
<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'.

If *range* = 'A', the routine computes all eigenvalues.
 If *range* = 'V', the routine computes eigenvalues $\lambda(i)$ in the half-open interval:
 $vl < \lambda(i) \leq vu$.
 If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.

uplo CHARACTER*1. Must be 'U' or 'L'.
 If *uplo* = 'U', arrays *ap* and *bp* store the upper triangles of *A* and *B*;
 If *uplo* = 'L', arrays *ap* and *bp* store the lower triangles of *A* and *B*.

n INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).

ap, bp, work COMPLEX for *chpgvx*
 DOUBLE COMPLEX for *zhpgvx*.
Arrays:
ap(*) contains the packed upper or lower triangle of the Hermitian matrix *A*, as specified by *uplo*.
 The dimension of *ap* must be at least $\max(1, n*(n+1)/2)$.
bp(*) contains the packed upper or lower triangle of the Hermitian matrix *B*, as specified by *uplo*.
 The dimension of *bp* must be at least $\max(1, n*(n+1)/2)$.
work(*) is a workspace array, DIMENSION at least $\max(1, 2n)$.

vl, vu REAL for *chpgvx*
 DOUBLE PRECISION for *zhpgvx*.
 If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.
 Constraint: $vl < vu$.
 If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, iu INTEGER.
 If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.
 Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$.
 If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol REAL for *chpgvx*

	DOUBLE PRECISION for zhpqvx. The absolute error tolerance for the eigenvalues. See <i>Application Notes</i> for more information.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i> ; <i>ldz</i> ≥ 1 . If <i>jobz</i> = 'V', <i>ldz</i> $\geq \max(1, n)$.
<i>rwork</i>	REAL for chpgvx DOUBLE PRECISION for zhpqvx. Workspace array, DIMENSION at least $\max(1, 7n)$.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, 5n)$.

Output Parameters

<i>ap</i>	On exit, the contents of <i>ap</i> are overwritten.
<i>bp</i>	On exit, contains the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $B = U^H * U$ or $B = L * L^H$, in the same storage format as <i>B</i> .
<i>m</i>	INTEGER. The total number of eigenvalues found, $0 \leq m \leq n$. If <i>range</i> = 'A', $m = n$, and if <i>range</i> = 'I', $m = iu - il + 1$.
<i>w</i>	REAL for chpgvx DOUBLE PRECISION for zhpqvx. Array, DIMENSION at least $\max(1, n)$. If <i>info</i> = 0, contains the eigenvalues in ascending order.
<i>z</i>	COMPLEX for chpgvx DOUBLE COMPLEX for zhpqvx. Array <i>z</i> (<i>ldz</i> ,*). The second dimension of <i>z</i> must be at least $\max(1, n)$. If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix <i>A</i> corresponding to the selected eigenvalues, with the <i>i</i> -th column of <i>z</i> holding the eigenvector associated with <i>w</i> (<i>i</i>). The eigenvectors are normalized as follows: if <i>itype</i> = 1 or 2, $Z^H * B * Z = I$; if <i>itype</i> = 3, $Z^H * \text{inv}(B) * Z = I$; If <i>jobz</i> = 'N', then <i>z</i> is not referenced.

If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array z ; if $range = 'V'$, the exact value of m is not known in advance and an upper bound must be used.

ifail

INTEGER.

Array, DIMENSION at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, the first m elements of *ifail* are zero; if $info > 0$, the *ifail* contains the indices of the eigenvectors that failed to converge.

If $jobz = 'N'$, then *ifail* is not referenced.

info

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th argument had an illegal value.

If $info > 0$, *cpptrf/zpptrf* and *chpevx/zhpevx* returned an error code:

If $info = i \leq n$, *chpevx/zhpevx* failed to converge, and i eigenvectors failed to converge. Their indices are stored in the array *ifail*;

If $info = n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *hpgvx* interface are the following:

<i>a</i>	Stands for argument <i>a_p</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(n * (n+1) / 2)$.
<i>b</i>	Stands for argument <i>b_p</i> in Fortran 77 interface. Holds the array <i>B</i> of size $(n * (n+1) / 2)$.
<i>w</i>	Holds the vector of length (n) .

<i>z</i>	Holds the matrix <i>z</i> of size (n, n) , where the values <i>n</i> and <i>m</i> are significant.
<i>ifail</i>	Holds the vector of length (n) .
<i>itype</i>	Must be 1, 2, or 3. The default value is 1.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>vl</i>	Default value for this element is $vl = -HUGE(vl)$.
<i>vu</i>	Default value for this element is $vu = HUGE(vl)$.
<i>il</i>	Default value for this argument is $il = 1$.
<i>iu</i>	Default value for this argument is $iu = n$.
<i>abstol</i>	Default value for this element is $abstol = 0.0\_WP$.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: $jobz = 'V'$, if <i>z</i> is present, $jobz = 'N'$, if <i>z</i> is omitted. Note that there will be an error condition if <i>ifail</i> is present and <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: $range = 'V'$, if one of or both <i>vl</i> and <i>vu</i> are present, $range = 'I'$, if one of or both <i>il</i> and <i>iu</i> are present, $range = 'A'$, if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present, Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon \cdot \max(|a|, |b|)$, where ϵ is the machine precision.

If *abstol* is less than or equal to zero, then $\epsilon \cdot ||T||_1$ is used as tolerance, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold $2 \cdot \text{lamch}('S')$, not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to $2 \cdot \text{lamch}('S')$.

?sbgv

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with banded matrices.

Syntax

FORTRAN 77:

```
call ssbgv(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, info)
call dsbgv(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, info)
```

Fortran 95:

```
call sbgv(a, b, w [,uplo] [,z] [,info])
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form $A^*x = \lambda^*B^*x$. Here A and B are assumed to be symmetric and banded, and B is also positive definite.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ab</i> and <i>bb</i> store the upper triangles of A and B ; If <i>uplo</i> = 'L', arrays <i>ab</i> and <i>bb</i> store the lower triangles of A and B .
<i>n</i>	INTEGER. The order of the matrices A and B ($n \geq 0$).
<i>ka</i>	INTEGER. The number of super- or sub-diagonals in A ($ka \geq 0$).
<i>kb</i>	INTEGER. The number of super- or sub-diagonals in B ($kb \geq 0$).
<i>ab, bb, work</i>	REAL for ssbgv

DOUBLE PRECISION for dsbgv

Arrays:

$ab(ldab,*)$ is an array containing either upper or lower triangular part of the symmetric matrix A (as specified by $uplo$) in band storage format.

The second dimension of the array ab must be at least $\max(1, n)$.

$bb(lddb,*)$ is an array containing either upper or lower triangular part of the symmetric matrix B (as specified by $uplo$) in band storage format.

The second dimension of the array bb must be at least $\max(1, n)$.

$work(*)$ is a workspace array, dimension at least $\max(1, 3n)$

$ldab$

INTEGER. The first dimension of the array ab ; must be at least $ka+1$.

$ldbb$

INTEGER. The first dimension of the array bb ; must be at least $kb+1$.

ldz

INTEGER. The leading dimension of the output array z ; $ldz \geq 1$. If $jobz = 'V'$, $ldz \geq \max(1, n)$.

Output Parameters

ab

On exit, the contents of ab are overwritten.

bb

On exit, contains the factor S from the split Cholesky factorization $B = S^T * S$, as returned by [spbstf/dpbstf](#).

w, z

REAL for ssbgv

DOUBLE PRECISION for dsbgv

Arrays:

$w(*)$, DIMENSION at least $\max(1, n)$.

If $info = 0$, contains the eigenvalues in ascending order.

$z(ldz,*)$.

The second dimension of z must be at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, z contains the matrix Z of eigenvectors, with the i -th column of z holding the eigenvector associated with $w(i)$. The eigenvectors are normalized so that $Z^T * B * Z = I$.

info If *jobz* = 'N', then *z* is not referenced.
 INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th argument had an illegal value.
 If *info* > 0, and
 if $i \leq n$, the algorithm failed to converge, and *i* off-diagonal
 elements of an intermediate tridiagonal did not converge to
 zero;
 if $info = n + i$, for $1 \leq i \leq n$, then [spbstf/dpbstf](#)
 returned *info* = *i* and *B* is not positive-definite. The
 factorization of *B* could not be completed and no eigenvalues
 or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *sbgv* interface are the following:

<i>a</i>	Stands for argument <i>ab</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(ka+1, n)$.
<i>b</i>	Stands for argument <i>bb</i> in Fortran 77 interface. Holds the array <i>B</i> of size $(kb+1, n)$.
<i>w</i>	Holds the vector of length (n) .
<i>z</i>	Holds the matrix <i>Z</i> of size (n, n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted.

?hbgv

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem with banded matrices.

Syntax

FORTRAN 77:

```
call chbgv(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, rwork, info)
```

```
call zhbgv(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, rwork, info)
```

Fortran 95:

```
call hbgv(a, b, w [,uplo] [,z] [,info])
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form $A^*x = \lambda^*B^*x$. Here A and B are Hermitian and banded matrices, and matrix B is also positive definite.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ab</i> and <i>bb</i> store the upper triangles of A and B ; If <i>uplo</i> = 'L', arrays <i>ab</i> and <i>bb</i> store the lower triangles of A and B .
<i>n</i>	INTEGER. The order of the matrices A and B ($n \geq 0$).
<i>ka</i>	INTEGER. The number of super- or sub-diagonals in A ($ka \geq 0$).

<i>kb</i>	INTEGER. The number of super- or sub-diagonals in <i>B</i> (<i>kb</i> ≥ 0).
<i>ab, bb, work</i>	COMPLEX for <i>chbgv</i> DOUBLE COMPLEX for <i>zhbgv</i> Arrays: <i>ab</i> (<i>ldab</i> ,*) is an array containing either upper or lower triangular part of the Hermitian matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format. The second dimension of the array <i>ab</i> must be at least $\max(1, n)$. <i>bb</i> (<i>ldbb</i> ,*) is an array containing either upper or lower triangular part of the Hermitian matrix <i>B</i> (as specified by <i>uplo</i>) in band storage format. The second dimension of the array <i>bb</i> must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array, dimension at least $\max(1, n)$.
<i>ldab</i>	INTEGER. The first dimension of the array <i>ab</i> ; must be at least <i>ka</i> +1.
<i>ldbb</i>	INTEGER. The first dimension of the array <i>bb</i> ; must be at least <i>kb</i> +1.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i> ; <i>ldz</i> ≥ 1 . If <i>jobz</i> = 'V', <i>ldz</i> $\geq \max(1, n)$.
<i>rwork</i>	REAL for <i>chbgv</i> DOUBLE PRECISION for <i>zhbgv</i> . Workspace array, DIMENSION at least $\max(1, 3n)$.

Output Parameters

<i>ab</i>	On exit, the contents of <i>ab</i> are overwritten.
<i>bb</i>	On exit, contains the factor <i>S</i> from the split Cholesky factorization $B = S^H * S$, as returned by cpbstf/zpbstf .
<i>w</i>	REAL for <i>chbgv</i> DOUBLE PRECISION for <i>zhbgv</i> . Array, DIMENSION at least $\max(1, n)$. If <i>info</i> = 0, contains the eigenvalues in ascending order.

z COMPLEX for chbgv
 DOUBLE COMPLEX for zhbgv
 Array *z*(*ldz*,*).
 The second dimension of *z* must be at least max(1, *n*).
 If *jobz* = 'V', then if *info* = 0, *z* contains the matrix *Z*
 of eigenvectors, with the *i*-th column of *z* holding the
 eigenvector associated with $w(i)$. The eigenvectors are
 normalized so that $Z^H * B * Z = I$.
 If *jobz* = 'N', then *z* is not referenced.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th argument had an illegal value.
 If *info* > 0, and
 if $i \leq n$, the algorithm failed to converge, and *i* off-diagonal
 elements of an intermediate tridiagonal did not converge to
 zero;
 if *info* = *n* + *i*, for $1 \leq i \leq n$, then cpbstf/zpbstf
 returned *info* = *i* and *B* is not positive-definite. The
 factorization of *B* could not be completed and no eigenvalues
 or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine hbgv interface are the following:

<i>a</i>	Stands for argument <i>ab</i> in Fortran 77 interface. Holds the array <i>A</i> of size (<i>ka</i> +1, <i>n</i>).
<i>b</i>	Stands for argument <i>bb</i> in Fortran 77 interface. Holds the array <i>B</i> of size (<i>kb</i> +1, <i>n</i>).
<i>w</i>	Holds the vector of length (<i>n</i>).
<i>z</i>	Holds the matrix <i>Z</i> of size (<i>n</i> , <i>n</i>).
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present,

`jobz = 'N'`, if `z` is omitted.

?sbgvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with banded matrices. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

FORTRAN 77:

```
call ssbgvd(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, lwork,
            iwork, liwork, info)

call dsbgvd(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, lwork,
            iwork, liwork, info)
```

Fortran 95:

```
call sbgvd(a, b, w [,uplo] [,z] [,info])
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form $A^*x = \lambda^*B^*x$. Here A and B are assumed to be symmetric and banded, and B is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

<code>jobz</code>	CHARACTER*1. Must be 'N' or 'V'. If <code>jobz = 'N'</code> , then compute eigenvalues only. If <code>jobz = 'V'</code> , then compute eigenvalues and eigenvectors.
<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'. If <code>uplo = 'U'</code> , arrays <code>ab</code> and <code>bb</code> store the upper triangles of A and B ; If <code>uplo = 'L'</code> , arrays <code>ab</code> and <code>bb</code> store the lower triangles of A and B .

<i>n</i>	INTEGER. The order of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>ka</i>	INTEGER. The number of super- or sub-diagonals in <i>A</i> ($ka \geq 0$).
<i>kb</i>	INTEGER. The number of super- or sub-diagonals in <i>B</i> ($kb \geq 0$).
<i>ab, bb, work</i>	REAL for ssbgvd DOUBLE PRECISION for dsbgvd Arrays: <i>ab</i> (<i>ldab</i> ,*) is an array containing either upper or lower triangular part of the symmetric matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format. The second dimension of the array <i>ab</i> must be at least $\max(1, n)$. <i>bb</i> (<i>ldbb</i> ,*) is an array containing either upper or lower triangular part of the symmetric matrix <i>B</i> (as specified by <i>uplo</i>) in band storage format. The second dimension of the array <i>bb</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>ldab</i>	INTEGER. The first dimension of the array <i>ab</i> ; must be at least $ka+1$.
<i>ldbb</i>	INTEGER. The first dimension of the array <i>bb</i> ; must be at least $kb+1$.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i> ; $ldz \geq 1$. If <i>jobz</i> = 'V', $ldz \geq \max(1, n)$.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . Constraints: If $n \leq 1$, $lwork \geq 1$; If <i>jobz</i> = 'N' and $n > 1$, $lwork \geq 3n$; If <i>jobz</i> = 'V' and $n > 1$, $lwork \geq 2n^2+5n+1$. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> and <i>iwork</i> arrays, returns these values as the first entries of the

work and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

iwork INTEGER.
Workspace array, its dimension $\max(1, liwork)$.

liwork INTEGER.
The dimension of the array *iwork*.
Constraints:
If $n \leq 1$, $liwork \geq 1$;
If $jobz = 'N'$ and $n > 1$, $liwork \geq 1$;
If $jobz = 'V'$ and $n > 1$, $liwork \geq 5n + 3$.
If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

ab On exit, the contents of *ab* are overwritten.

bb On exit, contains the factor *s* from the split Cholesky factorization $B = S^T * S$, as returned by [spbstf/dpbstf](#).

w, *z* REAL for [ssbgvd](#)
DOUBLE PRECISION for [dsbgvd](#)
Arrays:
w(*), DIMENSION at least $\max(1, n)$.
If *info* = 0, contains the eigenvalues in ascending order.
z(*ldz*,*).
The second dimension of *z* must be at least $\max(1, n)$.
If $jobz = 'V'$, then if *info* = 0, *z* contains the matrix *Z* of eigenvectors, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized so that $Z^T * B * Z = I$.
If $jobz = 'N'$, then *z* is not referenced.

<code>work(1)</code>	On exit, if <code>info = 0</code> , then <code>work(1)</code> returns the required minimal size of <code>lwork</code> .
<code>iwork(1)</code>	On exit, if <code>info = 0</code> , then <code>iwork(1)</code> returns the required minimal size of <code>liwork</code> .
<code>info</code>	INTEGER. If <code>info = 0</code>, the execution is successful. If <code>info = -i</code>, the <i>i</i>-th argument had an illegal value. If <code>info > 0</code>, and if $i \leq n$, the algorithm failed to converge, and <i>i</i> off-diagonal elements of an intermediate tridiagonal did not converge to zero; if <code>info = n + i</code>, for $1 \leq i \leq n$, then <code>spbstf/dpbstf</code> returned <code>info = i</code> and <i>B</i> is not positive-definite. The factorization of <i>B</i> could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sbgvd` interface are the following:

<code>a</code>	Stands for argument <code>ab</code> in Fortran 77 interface. Holds the array <i>A</i> of size $(ka+1, n)$.
<code>b</code>	Stands for argument <code>bb</code> in Fortran 77 interface. Holds the array <i>B</i> of size $(kb+1, n)$.
<code>w</code>	Holds the vector of length (n) .
<code>z</code>	Holds the matrix <i>Z</i> of size (n, n) .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.
<code>jobz</code>	Restored based on the presence of the argument <i>z</i> as follows: <code>jobz = 'V'</code>, if <i>z</i> is present, <code>jobz = 'N'</code>, if <i>z</i> is omitted.

Application Notes

If it is not clear how much workspace to supply, use a generous value of `lwork` (or `liwork`) for the first run or set `lwork = -1` (`liwork = -1`).

If $lwork$ (or $liwork$) has any of admissible sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array ($work$, $iwork$) on exit. Use this value ($work(1)$, $iwork(1)$) for subsequent runs.

If $lwork = -1$ ($liwork = -1$), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$, $iwork$). This operation is called a workspace query.

Note that if $work$ ($liwork$) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hbgvd

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem with banded matrices. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

FORTRAN 77:

```
call chbgvd(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, lwork,
rwork, lrwork, iwork, liwork, info)

call zhbgvd(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, lwork,
rwork, lrwork, iwork, liwork, info)
```

Fortran 95:

```
call hbgvd(a, b, w [,uplo] [,z] [,info])
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form $A^*x = \lambda^*B^*x$. Here A and B are assumed to be Hermitian and banded, and B is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ab</i> and <i>bb</i> store the upper triangles of <i>A</i> and <i>B</i>; If <i>uplo</i> = 'L', arrays <i>ab</i> and <i>bb</i> store the lower triangles of <i>A</i> and <i>B</i>.
<i>n</i>	INTEGER. The order of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>ka</i>	INTEGER. The number of super- or sub-diagonals in <i>A</i> ($ka \geq 0$).
<i>kb</i>	INTEGER. The number of super- or sub-diagonals in <i>B</i> ($kb \geq 0$).
<i>ab, bb, work</i>	COMPLEX for chbgvd DOUBLE COMPLEX for zhbgvd Arrays: <i>ab</i> (<i>ldab</i>,*) is an array containing either upper or lower triangular part of the Hermitian matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format. The second dimension of the array <i>ab</i> must be at least $\max(1, n)$. <i>bb</i> (<i>ldbb</i>,*) is an array containing either upper or lower triangular part of the Hermitian matrix <i>B</i> (as specified by <i>uplo</i>) in band storage format. The second dimension of the array <i>bb</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>ldab</i>	INTEGER. The first dimension of the array <i>ab</i> ; must be at least $ka+1$.
<i>ldbb</i>	INTEGER. The first dimension of the array <i>bb</i> ; must be at least $kb+1$.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i>; $ldz \geq 1$. If <i>jobz</i> = 'V', $ldz \geq \max(1, n)$.

lwork INTEGER.
The dimension of the array *work*.
Constraints:
If $n \leq 1$, $lwork \geq 1$;
If $jobz = 'N'$ and $n > 1$, $lwork \geq n$;
If $jobz = 'V'$ and $n > 1$, $lwork \geq 2n^2$.
If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

rwork REAL for *chbgvd*
DOUBLE PRECISION for *zhbgvd*.
Workspace array, DIMENSION $\max(1, lrwork)$.

lrwork INTEGER.
The dimension of the array *rwork*.
Constraints:
If $n \leq 1$, $lrwork \geq 1$;
If $jobz = 'N'$ and $n > 1$, $lrwork \geq n$;
If $jobz = 'V'$ and $n > 1$, $lrwork \geq 2n^2 + 5n + 1$.
If $lrwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

iwork INTEGER.
Workspace array, DIMENSION $\max(1, liwork)$.

liwork INTEGER.
The dimension of the array *iwork*.
Constraints:
If $n \leq 1$, $lwork \geq 1$;
If $jobz = 'N'$ and $n > 1$, $liwork \geq 1$;

If $jobz = 'V'$ and $n > 1$, $liwork \geq 5n + 3$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$, $rwork$ and $iwork$ arrays, returns these values as the first entries of the $work$, $rwork$ and $iwork$ arrays, and no error message related to $lwork$ or $lrwork$ or $liwork$ is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

ab	On exit, the contents of ab are overwritten.
bb	On exit, contains the factor s from the split Cholesky factorization $B = S^H * S$, as returned by cpbstf/zpbstf .
w	REAL for chbgvd DOUBLE PRECISION for zhbgvd . Array, DIMENSION at least $\max(1, n)$. If $info = 0$, contains the eigenvalues in ascending order.
z	COMPLEX for chbgvd DOUBLE COMPLEX for zhbgvd Array $z(ldz, *)$. The second dimension of z must be at least $\max(1, n)$. If $jobz = 'V'$, then if $info = 0$, z contains the matrix Z of eigenvectors, with the i -th column of z holding the eigenvector associated with $w(i)$. The eigenvectors are normalized so that $Z^H * B * Z = I$. If $jobz = 'N'$, then z is not referenced.
$work(1)$	On exit, if $info = 0$, then $work(1)$ returns the required minimal size of $lwork$.
$rwork(1)$	On exit, if $info = 0$, then $rwork(1)$ returns the required minimal size of $lrwork$.
$iwork(1)$	On exit, if $info = 0$, then $iwork(1)$ returns the required minimal size of $liwork$.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th argument had an illegal value. If $info > 0$, and

if $i \leq n$, the algorithm failed to converge, and i off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if $info = n + i$, for $1 \leq i \leq n$, then `cpbstf/zpbstf` returned $info = i$ and B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hbgvd` interface are the following:

<i>a</i>	Stands for argument <i>ab</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(ka+1, n)$.
<i>b</i>	Stands for argument <i>bb</i> in Fortran 77 interface. Holds the array <i>B</i> of size $(kb+1, n)$.
<i>w</i>	Holds the vector of length (n) .
<i>z</i>	Holds the matrix <i>Z</i> of size (n, n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: $jobz = 'V'$, if <i>z</i> is present, $jobz = 'N'$, if <i>z</i> is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* (*liwork* or *lrwork*) for the first run or set $lwork = -1$ ($liwork = -1$, $lrwork = -1$).

If you choose the first option and set any of admissible *lwork* (*liwork* or *lrwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*) on exit. Use this value (*work*(1), *iwork*(1), *rwork*(1)) for subsequent runs.

If you set `lwork = -1` (`liwork = -1`, `lrwork = -1`), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`, `iwork`, `rwork`). This operation is called a workspace query.

Note that if you set `lwork` (`liwork`, `lrwork`) to less than the minimal required value and not `-1`, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?sbgvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with banded matrices.

Syntax

FORTRAN 77:

```
call ssbgvx(jobz, range, uplo, n, ka, kb, ab, ldab, bb, ldbb, q, ldq, vl, vu,
il, iu, abstol, m, w, z, ldz, work, iwork, ifail, info)

call dsbgvx(jobz, range, uplo, n, ka, kb, ab, ldab, bb, ldbb, q, ldq, vl, vu,
il, iu, abstol, m, w, z, ldz, work, iwork, ifail, info)
```

Fortran 95:

```
call sbgvx(a, b, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,q]
[,abstol] [,info])
```

Description

This routine computes selected eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form $A^*x = \lambda B^*x$. Here A and B are assumed to be symmetric and banded, and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either all eigenvalues, a range of values or a range of indices for the desired eigenvalues.

Input Parameters

<code>jobz</code>	CHARACTER*1. Must be 'N' or 'V'. If <code>jobz = 'N'</code> , then compute eigenvalues only. If <code>jobz = 'V'</code> , then compute eigenvalues and eigenvectors.
<code>range</code>	CHARACTER*1. Must be 'A' or 'V' or 'I'.

	If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues $\lambda(i)$ in the half-open interval: $vl < \lambda(i) \leq vu.$ If <i>range</i> = 'I', the routine computes eigenvalues in range <i>il</i> to <i>iu</i>.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ab</i> and <i>bb</i> store the upper triangles of <i>A</i> and <i>B</i>; If <i>uplo</i> = 'L', arrays <i>ab</i> and <i>bb</i> store the lower triangles of <i>A</i> and <i>B</i>.
<i>n</i>	INTEGER. The order of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>ka</i>	INTEGER. The number of super- or sub-diagonals in <i>A</i> ($ka \geq 0$).
<i>kb</i>	INTEGER. The number of super- or sub-diagonals in <i>B</i> ($kb \geq 0$).
<i>ab, bb, work</i>	REAL for <i>ssbgvx</i> DOUBLE PRECISION for <i>dsbgvx</i> Arrays: <i>ab</i> (<i>ldab</i>,*) is an array containing either upper or lower triangular part of the symmetric matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format. The second dimension of the array <i>ab</i> must be at least $\max(1, n)$. <i>bb</i> (<i>ldbb</i>,*) is an array containing either upper or lower triangular part of the symmetric matrix <i>B</i> (as specified by <i>uplo</i>) in band storage format. The second dimension of the array <i>bb</i> must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array, DIMENSION (7*n).
<i>ldab</i>	INTEGER. The first dimension of the array <i>ab</i> ; must be at least $ka+1$.
<i>ldbb</i>	INTEGER. The first dimension of the array <i>bb</i> ; must be at least $kb+1$.
<i>vl, vu</i>	REAL for <i>ssbgvx</i>

	DOUBLE PRECISION for dsbgvx. If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: <i>vl</i> < <i>vu</i> . If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il, iu</i>	INTEGER. If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; <i>il</i> =1 and <i>iu</i> =0 if $n = 0$. If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.
<i>abstol</i>	REAL for ssbgvx DOUBLE PRECISION for dsbgvx. The absolute error tolerance for the eigenvalues. See <i>Application Notes</i> for more information.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i> ; <i>ldz</i> ≥ 1. If <i>jobz</i> = 'V', <i>ldz</i> ≥ max(1, <i>n</i>).
<i>ldq</i>	INTEGER. The leading dimension of the output array <i>q</i> ; <i>ldq</i> < 1. If <i>jobz</i> = 'V', <i>ldq</i> < max(1, <i>n</i>).
<i>iwork</i>	INTEGER. Workspace array, DIMENSION (5*n).

Output Parameters

<i>ab</i>	On exit, the contents of <i>ab</i> are overwritten.
<i>bb</i>	On exit, contains the factor <i>s</i> from the split Cholesky factorization $B = S^T * S$, as returned by spbstf / dpbstf .
<i>m</i>	INTEGER. The total number of eigenvalues found, $0 \leq m \leq n$. If <i>range</i> = 'A', <i>m</i> = <i>n</i> , and if <i>range</i> = 'I', <i>m</i> = <i>iu</i> - <i>il</i> +1.
<i>w, z, q</i>	REAL for ssbgvx DOUBLE PRECISION for dsbgvx Arrays: <i>w</i> (*), DIMENSION at least max(1, <i>n</i>) . If <i>info</i> = 0, contains the eigenvalues in ascending order.

$z(ldz,*)$.

The second dimension of z must be at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, z contains the matrix Z of eigenvectors, with the i -th column of z holding the eigenvector associated with $w(i)$. The eigenvectors are normalized so that $Z^T * B * Z = I$.

If $jobz = 'N'$, then z is not referenced.

$q(ldq,*)$.

The second dimension of q must be at least $\max(1, n)$.

If $jobz = 'V'$, then q contains the n -by- n matrix used in the reduction of $A * x = \lambda * B * x$ to standard form, that is, $C * x = \lambda * x$ and consequently C to tridiagonal form.

If $jobz = 'N'$, then q is not referenced.

$ifail$

INTEGER.

Array, DIMENSION (m).

If $jobz = 'V'$, then if $info = 0$, the first m elements of $ifail$ are zero; if $info > 0$, the $ifail$ contains the indices of the eigenvectors that failed to converge.

If $jobz = 'N'$, then $ifail$ is not referenced.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th argument had an illegal value.

If $info > 0$, and

if $i \leq n$, the algorithm failed to converge, and i off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if $info = n + i$, for $1 \leq i \leq n$, then [spbstf/dpbstf](#) returned $info = i$ and B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sbgvx` interface are the following:

<i>a</i>	Stands for argument <i>ab</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(ka+1, n)$.
<i>b</i>	Stands for argument <i>bb</i> in Fortran 77 interface. Holds the array <i>B</i> of size $(kb+1, n)$.
<i>w</i>	Holds the vector of length (n) .
<i>z</i>	Holds the matrix <i>Z</i> of size (n, n) .
<i>ifail</i>	Holds the vector of length (n) .
<i>q</i>	Holds the matrix <i>Q</i> of size (n, n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>vl</i>	Default value for this element is $vl = -HUGE(vl)$.
<i>vu</i>	Default value for this element is $vu = HUGE(vl)$.
<i>il</i>	Default value for this argument is $il = 1$.
<i>iu</i>	Default value for this argument is $iu = n$.
<i>abstol</i>	Default value for this element is $abstol = 0.0\_WP$.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: $jobz = 'V'$, if <i>z</i> is present, $jobz = 'N'$, if <i>z</i> is omitted. Note that there will be an error condition if <i>ifail</i> or <i>q</i> is present and <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: $range = 'V'$, if one of or both <i>vl</i> and <i>vu</i> are present, $range = 'I'$, if one of or both <i>il</i> and <i>iu</i> are present, $range = 'A'$, if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present, Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon * \max(|a|, |b|)$, where ϵ is the machine precision.

If *abstol* is less than or equal to zero, then $\epsilon * ||T||_1$ is used as tolerance, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold $2 * ?lamch('S')$, not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

?hbgvx

Computes selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem with banded matrices.

Syntax

FORTRAN 77:

```
call chbgvx(jobz, range, uplo, n, ka, kb, ab, ldab, bb, ldbb, q, ldq, vl, vu,
il, iu, abstol, m, w, z, ldz, work, rwork, iwork, ifail, info)

call zhbgvx(jobz, range, uplo, n, ka, kb, ab, ldab, bb, ldbb, q, ldq, vl, vu,
il, iu, abstol, m, w, z, ldz, work, rwork, iwork, ifail, info)
```

Fortran 95:

```
call hbgvx(a, b, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,q]
[,abstol] [,info])
```

Description

This routine computes selected eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form $A*x = \lambda*B*x$. Here *A* and *B* are assumed to be Hermitian and banded, and *B* is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either all eigenvalues, a range of values or a range of indices for the desired eigenvalues.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues $\lambda(i)$ in the half-open interval: $vl < \lambda(i) \leq vu$.

	If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i>.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ab</i> and <i>bb</i> store the upper triangles of <i>A</i> and <i>B</i>; If <i>uplo</i> = 'L', arrays <i>ab</i> and <i>bb</i> store the lower triangles of <i>A</i> and <i>B</i>.
<i>n</i>	INTEGER. The order of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>ka</i>	INTEGER. The number of super- or sub-diagonals in <i>A</i> ($ka \geq 0$).
<i>kb</i>	INTEGER. The number of super- or sub-diagonals in <i>B</i> ($kb \geq 0$).
<i>ab, bb, work</i>	COMPLEX for chbgvx DOUBLE COMPLEX for zhbgvx Arrays: <i>ab</i> (<i>ldab</i>,*) is an array containing either upper or lower triangular part of the Hermitian matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format. The second dimension of the array <i>ab</i> must be at least $\max(1, n)$. <i>bb</i> (<i>ldbb</i>,*) is an array containing either upper or lower triangular part of the Hermitian matrix <i>B</i> (as specified by <i>uplo</i>) in band storage format. The second dimension of the array <i>bb</i> must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array, DIMENSION at least $\max(1, n)$.
<i>ldab</i>	INTEGER. The first dimension of the array <i>ab</i>; must be at least <i>ka</i>+1.
<i>ldbb</i>	INTEGER. The first dimension of the array <i>bb</i>; must be at least <i>kb</i>+1.
<i>vl, vu</i>	REAL for chbgvx DOUBLE PRECISION for zhbgvx. If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.

	Constraint: $vl < vu$. If $range = 'A'$ or $'I'$, vl and vu are not referenced.
il, iu	INTEGER. If $range = 'I'$, the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$. If $range = 'A'$ or $'V'$, il and iu are not referenced.
$abstol$	REAL for chbgvx DOUBLE PRECISION for zhbgbvx. The absolute error tolerance for the eigenvalues. See <i>Application Notes</i> for more information.
ldz	INTEGER. The leading dimension of the output array z ; $ldz \geq 1$. If $jobz = 'V'$, $ldz \geq \max(1, n)$.
ldq	INTEGER. The leading dimension of the output array q ; $ldq \geq 1$. If $jobz = 'V'$, $ldq \geq \max(1, n)$.
$rwork$	REAL for chbgvx DOUBLE PRECISION for zhbgbvx. Workspace array, DIMENSION at least $\max(1, 7n)$.
$iwork$	INTEGER. Workspace array, DIMENSION at least $\max(1, 5n)$.

Output Parameters

ab	On exit, the contents of ab are overwritten.
bb	On exit, contains the factor s from the split Cholesky factorization $B = S^H * S$, as returned by cpbstf/zpbstf .
m	INTEGER. The total number of eigenvalues found, $0 \leq m \leq n$. If $range = 'A'$, $m = n$, and if $range = 'I'$, $m = iu-il+1$.
w	REAL for chbgvx DOUBLE PRECISION for zhbgbvx. Array $w(*)$, DIMENSION at least $\max(1, n)$. If $info = 0$, contains the eigenvalues in ascending order.
z, q	COMPLEX for chbgvx

DOUBLE COMPLEX for zhbgvx

Arrays:

$z(ldz,*)$.

The second dimension of z must be at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, z contains the matrix Z of eigenvectors, with the i -th column of z holding the eigenvector associated with $w(i)$. The eigenvectors are normalized so that $Z^H * B * Z = I$.

If $jobz = 'N'$, then z is not referenced.

$q(ldq,*)$.

The second dimension of q must be at least $\max(1, n)$.

If $jobz = 'V'$, then q contains the n -by- n matrix used in the reduction of $Ax = \lambda Bx$ to standard form, that is, $Cx = \lambda x$ and consequently C to tridiagonal form.

If $jobz = 'N'$, then q is not referenced.

$ifail$

INTEGER.

Array, DIMENSION at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, the first m elements of $ifail$ are zero; if $info > 0$, the $ifail$ contains the indices of the eigenvectors that failed to converge.

If $jobz = 'N'$, then $ifail$ is not referenced.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th argument had an illegal value.

If $info > 0$, and

if $i \leq n$, the algorithm failed to converge, and i off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if $info = n + i$, for $1 \leq i \leq n$, then [cpbstf/zpbstf](#) returned $info = i$ and B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hbgvx` interface are the following:

<i>a</i>	Stands for argument <i>ab</i> in Fortran 77 interface. Holds the array <i>A</i> of size $(ka+1, n)$.
<i>b</i>	Stands for argument <i>bb</i> in Fortran 77 interface. Holds the array <i>B</i> of size $(kb+1, n)$.
<i>w</i>	Holds the vector of length (n) .
<i>z</i>	Holds the matrix <i>Z</i> of size (n, n) .
<i>ifail</i>	Holds the vector of length (n) .
<i>q</i>	Holds the matrix <i>Q</i> of size (n, n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>vl</i>	Default value for this element is $vl = -HUGE(vl)$.
<i>vu</i>	Default value for this element is $vu = HUGE(vl)$.
<i>il</i>	Default value for this argument is $il = 1$.
<i>iu</i>	Default value for this argument is $iu = n$.
<i>abstol</i>	Default value for this element is $abstol = 0.0\_WP$.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: $jobz = 'V'$, if <i>z</i> is present, $jobz = 'N'$, if <i>z</i> is omitted. Note that there will be an error condition if <i>ifail</i> or <i>q</i> is present and <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: $range = 'V'$, if one of or both <i>vl</i> and <i>vu</i> are present, $range = 'I'$, if one of or both <i>il</i> and <i>iu</i> are present, $range = 'A'$, if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present, Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon \max(|a|, |b|)$, where ϵ is the machine precision.

If $abstol$ is less than or equal to zero, then $\epsilon * ||T||_1$ will be used in its place, where T is the tridiagonal matrix obtained by reducing A to tridiagonal form. Eigenvalues will be computed most accurately when $abstol$ is set to twice the underflow threshold $2 * \text{lamch}('S')$, not zero.

If this routine returns with $info > 0$, indicating that some eigenvectors did not converge, try setting $abstol$ to $2 * \text{lamch}('S')$.

Generalized Nonsymmetric Eigenproblems

This section describes LAPACK driver routines used for solving generalized nonsymmetric eigenproblems. See also computational routines [computational routines](#) that can be called to solve these problems. [Table 4-14](#) lists all such driver routines for Fortran 77 interface. Respective routine names in Fortran 95 interface are without the first symbol (see [Routine Naming Conventions](#)).

Table 4-14 Driver Routines for Solving Generalized Nonsymmetric Eigenproblems

Routine Name	Operation performed
?gges	Computes the generalized eigenvalues, Schur form, and the left and/or right Schur vectors for a pair of nonsymmetric matrices.
?ggesx	Computes the generalized eigenvalues, Schur form, and, optionally, the left and/or right matrices of Schur vectors.
?ggeev	Computes the generalized eigenvalues, and the left and/or right generalized eigenvectors for a pair of nonsymmetric matrices.
?ggevx	Computes the generalized eigenvalues, and, optionally, the left and/or right generalized eigenvectors.

?gges

Computes the generalized eigenvalues, Schur form, and the left and/or right Schur vectors for a pair of nonsymmetric matrices.

Syntax

FORTRAN 77:

```
call sgges(jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim, alphas,
alpha, beta, vsl, ldvsl, vsr, ldvsr, work, lwork, bwork, info)

call dgges(jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim, alphas,
alpha, beta, vsl, ldvsl, vsr, ldvsr, work, lwork, bwork, info)

call cgges(jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim, alpha, beta,
vsl, ldvsl, vsr, ldvsr, work, lwork, rwork, bwork, info)

call zgges(jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim, alpha, beta,
vsl, ldvsl, vsr, ldvsr, work, lwork, rwork, bwork, info)
```

Fortran 95:

```
call gges(a, b, alphas, alpha, beta [,vsl] [,vsr] [,select] [,sdim] [,info])
call gges(a, b, alpha, beta [, vsl] [,vsr] [,select] [,sdim] [,info])
```

Description

This routine computes for a pair of n -by- n real/complex nonsymmetric matrices (A,B) , the generalized eigenvalues, the generalized real/complex Schur form (S,T) , optionally, the left and/or right matrices of Schur vectors (vsl and vsr). This gives the generalized Schur factorization

$$(A,B) = (vsl * S * vsr^H, vsl * T * vsr^H)$$

Optionally, it also orders the eigenvalues so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix S and the upper triangular matrix T . The leading columns of vsl and vsr then form an orthonormal/unitary basis for the corresponding left and right eigenspaces (deflating subspaces).

(If only the generalized eigenvalues are needed, use the driver [?ggeev](#) instead, which is faster.)

A generalized eigenvalue for a pair of matrices (A, B) is a scalar w or a ratio $\alpha / \beta = w$, such that $A - w*B$ is singular. It is usually represented as the pair (α, β) , as there is a reasonable interpretation for $\beta=0$ or for both being zero. A pair of matrices (S, T) is in generalized real Schur form if T is upper triangular with non-negative diagonal and S is block upper triangular with 1-by-1 and 2-by-2 blocks. 1-by-1 blocks correspond to real generalized eigenvalues, while 2-by-2 blocks of S will be "standardized" by making the corresponding elements of T have the form:

$$\begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}$$

and the pair of corresponding 2-by-2 blocks in S and T will have a complex conjugate pair of generalized eigenvalues. A pair of matrices (S, T) is in generalized complex Schur form if S and T are upper triangular and, in addition, the diagonal of T are non-negative real numbers.

Input Parameters

<i>jobvsl</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobvsl</i> = 'N', then the left Schur vectors are not computed. If <i>jobvsl</i> = 'V', then the left Schur vectors are computed.
<i>jobvsr</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobvsr</i> = 'N', then the right Schur vectors are not computed. If <i>jobvsr</i> = 'V', then the right Schur vectors are computed.
<i>sort</i>	CHARACTER*1. Must be 'N' or 'S'. Specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form. If <i>sort</i> = 'N', then eigenvalues are not ordered. If <i>sort</i> = 'S', eigenvalues are ordered (see <i>selctg</i>).
<i>selctg</i>	LOGICAL FUNCTION of three REAL arguments for real flavors. LOGICAL FUNCTION of two COMPLEX arguments for complex flavors. <i>selctg</i> must be declared EXTERNAL in the calling subroutine.

If *sort* = 'S', *selctg* is used to select eigenvalues to sort to the top left of the Schur form.

If *sort* = 'N', *selctg* is not referenced.

For real flavors:

An eigenvalue $(\text{alphan}(j) + \text{alphai}(j))/\text{betan}(j)$ is selected if *selctg*(*alphan*(*j*), *alphai*(*j*), *betan*(*j*)) is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.

Note that in the ill-conditioned case, a selected complex eigenvalue may no longer satisfy *selctg*(*alphan*(*j*), *alphai*(*j*), *betan*(*j*)) = .TRUE. after ordering. In this case *info* is set to *n*+2 .

For complex flavors:

An eigenvalue $\text{alpha}(j) / \text{beta}(j)$ is selected if *selctg*(*alpha*(*j*), *beta*(*j*)) is true.

Note that a selected complex eigenvalue may no longer satisfy *selctg*(*alpha*(*j*), *beta*(*j*)) = .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case *info* is set to *n*+2 (see *info* below).

n INTEGER. The order of the matrices *A*, *B*, *vsl*, and *vsr* (*n* ≥ 0).

a, *b*, *work* REAL for sgges
DOUBLE PRECISION for dgges
COMPLEX for cgges
DOUBLE COMPLEX for zgges.

Arrays:

a(*lda*,*) is an array containing the *n*-by-*n* matrix *A* (first of the pair of matrices).

The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) is an array containing the *n*-by-*n* matrix *B* (second of the pair of matrices).

The second dimension of *b* must be at least max(1, *n*).

work is a workspace array, its dimension max(1, *lwork*) .

lda INTEGER. The first dimension of the array *a*. Must be at least max(1, *n*).

<i>ldb</i>	INTEGER. The first dimension of the array <i>b</i> . Must be at least $\max(1, n)$.
<i>ldvsl, ldvsr</i>	INTEGER. The first dimensions of the output matrices <i>vsl</i> and <i>vsr</i> , respectively. Constraints: $ldvsl \geq 1$. If <i>jobvsl</i> = 'V', $ldvsl \geq \max(1, n)$. $ldvsr \geq 1$. If <i>jobvsr</i> = 'V', $ldvsr \geq \max(1, n)$.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . $lwork \geq \max(1, 8n+16)$ for real flavors; $lwork \geq \max(1, 2n)$ for complex flavors. For good performance, <i>lwork</i> must generally be larger. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla .
<i>rwork</i>	REAL for <i>cgges</i> DOUBLE PRECISION for <i>zgges</i> Workspace array, DIMENSION at least $\max(1, 8n)$. This array is used in complex flavors only.
<i>bwork</i>	LOGICAL. Workspace array, DIMENSION at least $\max(1, n)$. Not referenced if <i>sort</i> = 'N'.

Output Parameters

<i>a</i>	On exit, this array has been overwritten by its generalized Schur form <i>S</i> .
<i>b</i>	On exit, this array has been overwritten by its generalized Schur form <i>T</i> .
<i>sdim</i>	INTEGER. If <i>sort</i> = 'N', <i>sdim</i> = 0. If <i>sort</i> = 'S', <i>sdim</i> is equal to the number of eigenvalues (after sorting) for which <i>selctg</i> is true. Note that for real flavors complex conjugate pairs for which <i>selctg</i> is true for either eigenvalue count as 2.
<i>alphar, alpha</i>	REAL for <i>sgges</i> ;

DOUBLE PRECISION for dgges.
 Arrays, DIMENSION at least $\max(1, n)$ each. Contain values that form generalized eigenvalues in real flavors.
 See *beta*.

alpha COMPLEX for cgges;
 DOUBLE COMPLEX for zgges.
 Array, DIMENSION at least $\max(1, n)$. Contain values that form generalized eigenvalues in complex flavors. See *beta*.

beta REAL for sgges
 DOUBLE PRECISION for dgges
 COMPLEX for cgges
 DOUBLE COMPLEX for zgges.
 Array, DIMENSION at least $\max(1, n)$.
For real flavors:
 On exit, $(\text{alphar}(j) + \text{alphai}(j)*i)/\text{beta}(j)$, $j=1, \dots, n$, will be the generalized eigenvalues.
 $\text{alphar}(j) + \text{alphai}(j)*i$ and $\text{beta}(j)$, $j=1, \dots, n$ are the diagonals of the complex Schur form (S, T) that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of (A, B) were further reduced to triangular form using complex unitary transformations. If $\text{alphai}(j)$ is zero, then the j -th eigenvalue is real; if positive, then the j -th and $(j+1)$ -st eigenvalues are a complex conjugate pair, with $\text{alphai}(j+1)$ negative.
For complex flavors:
 On exit, $\text{alpha}(j)/\text{beta}(j)$, $j=1, \dots, n$, will be the generalized eigenvalues. $\text{alpha}(j)$, $j=1, \dots, n$, and $\text{beta}(j)$, $j=1, \dots, n$ are the diagonals of the complex Schur form (S, T) output by cgges/zgges. The $\text{beta}(j)$ will be non-negative real.
 See also *Application Notes* below.

vsl, vsr REAL for sgges
 DOUBLE PRECISION for dgges
 COMPLEX for cgges
 DOUBLE COMPLEX for zgges.
 Arrays:
vsl(*ldvsl*,*), the second dimension of *vsl* must be at least $\max(1, n)$.

If `jobvsl = 'V'`, this array will contain the left Schur vectors.

If `jobvsl = 'N'`, `vsl` is not referenced.

`vsr(ldvsr,*)`, the second dimension of `vsr` must be at least $\max(1, n)$.

If `jobvsr = 'V'`, this array will contain the right Schur vectors.

If `jobvsr = 'N'`, `vsr` is not referenced.

`work(1)` On exit, if `info = 0`, then `work(1)` returns the required minimal size of `lwork`.

`info` INTEGER.

If `info = 0`, the execution is successful.

If `info = -i`, the *i*th parameter had an illegal value.

If `info = i`, and

$i \leq n$:

the *QZ* iteration failed. (*A*, *B*) is not in Schur form, but `alphar(j)`, `alphai(j)` (for real flavors), or `alpha(j)` (for complex flavors), and `beta(j)`, $j = \text{info} + 1, \dots, n$ should be correct.

$i > n$: errors that usually indicate LAPACK problems:

$i = n + 1$: other than *QZ* iteration failed in [?hgeqz](#);

$i = n + 2$: after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the generalized Schur form no longer satisfy `selctg = .TRUE..`
This could also be caused due to scaling;

$i = n + 3$: reordering failed in [?tgsen](#).

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gges` interface are the following:

<code>a</code>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<code>b</code>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>n</i>).
<code>alphar</code>	Holds the vector of length (<i>n</i>). Used in real flavors only.
<code>alphai</code>	Holds the vector of length (<i>n</i>). Used in real flavors only.

<i>alpha</i>	Holds the vector of length (<i>n</i>). Used in complex flavors only.
<i>beta</i>	Holds the vector of length (<i>n</i>).
<i>vsl</i>	Holds the matrix <i>VSL</i> of size (<i>n</i> , <i>n</i>).
<i>vsr</i>	Holds the matrix <i>VSR</i> of size (<i>n</i> , <i>n</i>).
<i>jobvsl</i>	Restored based on the presence of the argument <i>vsl</i> as follows: <i>jobvsl</i> = 'V', if <i>vsl</i> is present, <i>jobvsl</i> = 'N', if <i>vsl</i> is omitted.
<i>jobvsr</i>	Restored based on the presence of the argument <i>vsr</i> as follows: <i>jobvsr</i> = 'V', if <i>vsr</i> is present, <i>jobvsr</i> = 'N', if <i>vsr</i> is omitted.
<i>sort</i>	Restored based on the presence of the argument <i>select</i> as follows: <i>sort</i> = 'S', if <i>select</i> is present, <i>sort</i> = 'N', if <i>select</i> is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The quotients *alphar*(*j*)/*beta*(*j*) and *alphai*(*j*)/*beta*(*j*) may easily over- or underflow, and *beta*(*j*) may even be zero. Thus, you should avoid simply computing the ratio. However, *alphar* and *alphai* will be always less than and usually comparable with *norm*(*A*) in magnitude, and *beta* always less than and usually comparable with *norm*(*B*).

sggesx

Computes the generalized eigenvalues, Schur form, and, optionally, the left and/or right matrices of Schur vectors.

Syntax

FORTRAN 77:

```
call sggesx (jobvsl, jobvsr, sort, selctg, sense, n, a, lda, b, ldb, sdim,
            alphas, alphas, beta, vsl, ldvsl, vsr, ldvsr, rconde, rcondv, work, lwork,
            iwork, liwork, bwork, info)
```

```
call dggesx (jobvsl, jobvsr, sort, selctg, sense, n, a, lda, b, ldb, sdim,
            alphas, alphas, beta, vsl, ldvsl, vsr, ldvsr, rconde, rcondv, work, lwork,
            iwork, liwork, bwork, info)
```

```
call cggesx (jobvsl, jobvsr, sort, selctg, sense, n, a, lda, b, ldb, sdim,
            alpha, beta, vsl, ldvsl, vsr, ldvsr, rconde, rcondv, work, lwork, rwork,
            iwork, liwork, bwork, info)
```

```
call zggesx (jobvsl, jobvsr, sort, selctg, sense, n, a, lda, b, ldb, sdim,
            alpha, beta, vsl, ldvsl, vsr, ldvsr, rconde, rcondv, work, lwork, rwork,
            iwork, liwork, bwork, info)
```

Fortran 95:

```
call ggesx(a, b, alphas, alphas, beta [,vsl] [,vsr] [,select] [,sdim] [,rconde]
[, rcondv] [,info])
```

```
call ggesx(a, b, alpha, beta [, vsl] [,vsr] [,select] [,sdim] [,rconde]
[, rcondv] [, info])
```

Description

This routine computes for a pair of n -by- n real/complex nonsymmetric matrices (A,B) , the generalized eigenvalues, the generalized real/complex Schur form (S,T) , optionally, the left and/or right matrices of Schur vectors (vsl and vsr). This gives the generalized Schur factorization

$$(A,B) = (vsl * S * vsr^H, vsl * T * vsr^H)$$

Optionally, it also orders the eigenvalues so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix S and the upper triangular matrix T ; computes a reciprocal condition number for the average of the selected eigenvalues ($rconde$); and computes a reciprocal condition number for the right and left deflating subspaces corresponding to the selected eigenvalues ($rcondv$). The leading columns of vsl and vsr then form an orthonormal/unitary basis for the corresponding left and right eigenspaces (deflating subspaces).

A generalized eigenvalue for a pair of matrices (A,B) is a scalar w or a ratio $\alpha / \beta = w$, such that $A - w*B$ is singular. It is usually represented as the pair (α, β) , as there is a reasonable interpretation for $\beta=0$ or for both being zero. A pair of matrices (S,T) is in generalized real Schur form if T is upper triangular with non-negative diagonal and S is block upper triangular with 1-by-1 and 2-by-2 blocks. 1-by-1 blocks correspond to real generalized eigenvalues, while 2-by-2 blocks of S will be "standardized" by making the corresponding elements of T have the form:

$$\begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}$$

and the pair of corresponding 2-by-2 blocks in S and T will have a complex conjugate pair of generalized eigenvalues. A pair of matrices (S,T) is in generalized complex Schur form if S and T are upper triangular and, in addition, the diagonal of T are non-negative real numbers.

Input Parameters

jobvsl CHARACTER*1. Must be 'N' or 'V'.
 If *jobvsl* = 'N', then the left Schur vectors are not computed.
 If *jobvsl* = 'V', then the left Schur vectors are computed.

jobvsr CHARACTER*1. Must be 'N' or 'V'.
 If *jobvsr* = 'N', then the right Schur vectors are not computed.
 If *jobvsr* = 'V', then the right Schur vectors are computed.

<i>sort</i>	CHARACTER*1. Must be 'N' or 'S'. Specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form. If <i>sort</i> = 'N', then eigenvalues are not ordered. If <i>sort</i> = 'S', eigenvalues are ordered (see <i>selctg</i>).
<i>selctg</i>	LOGICAL FUNCTION of three REAL arguments for real flavors. LOGICAL FUNCTION of two COMPLEX arguments for complex flavors. <i>selctg</i> must be declared EXTERNAL in the calling subroutine. If <i>sort</i> = 'S', <i>selctg</i> is used to select eigenvalues to sort to the top left of the Schur form. If <i>sort</i> = 'N', <i>selctg</i> is not referenced. For real flavors: An eigenvalue $(\text{alphan}(j) + \text{alphai}(j))/\text{beta}(j)$ is selected if <i>selctg</i>(<i>alphan</i>(<i>j</i>), <i>alphai</i>(<i>j</i>), <i>beta</i>(<i>j</i>)) is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected. Note that in the ill-conditioned case, a selected complex eigenvalue may no longer satisfy <i>selctg</i>(<i>alphan</i>(<i>j</i>), <i>alphai</i>(<i>j</i>), <i>beta</i>(<i>j</i>)) = .TRUE. after ordering. In this case <i>info</i> is set to <i>n</i>+2. For complex flavors: An eigenvalue $\text{alpha}(j) / \text{beta}(j)$ is selected if <i>selctg</i>(<i>alpha</i>(<i>j</i>), <i>beta</i>(<i>j</i>)) is true. Note that a selected complex eigenvalue may no longer satisfy <i>selctg</i>(<i>alpha</i>(<i>j</i>), <i>beta</i>(<i>j</i>)) = .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case <i>info</i> is set to <i>n</i>+2 (see <i>info</i> below).
<i>sense</i>	CHARACTER*1. Must be 'N', 'E', 'V', or 'B'. Determines which reciprocal condition number are computed. If <i>sense</i> = 'N', none are computed; If <i>sense</i> = 'E', computed for average of selected eigenvalues only; If <i>sense</i> = 'V', computed for selected deflating subspaces only; If <i>sense</i> = 'B', computed for both.

If *sense* is 'E', 'V', or 'B', then *sort* must equal 'S'.

n
INTEGER. The order of the matrices *A*, *B*, *vsl*, and *vsr* ($n \geq 0$).

a, *b*, *work*
REAL for sggesx
DOUBLE PRECISION for dggesx
COMPLEX for cggesx
DOUBLE COMPLEX for zggesx.
Arrays:
a(lda,)* is an array containing the *n*-by-*n* matrix *A* (first of the pair of matrices).
The second dimension of *a* must be at least $\max(1, n)$.
b(ldb,)* is an array containing the *n*-by-*n* matrix *B* (second of the pair of matrices).
The second dimension of *b* must be at least $\max(1, n)$.
work is a workspace array, its dimension $\max(1, lwork)$.

lda
INTEGER. The first dimension of the array *a*.
Must be at least $\max(1, n)$.

ldb
INTEGER. The first dimension of the array *b*.
Must be at least $\max(1, n)$.

ldvsl, *ldvsr*
INTEGER. The first dimensions of the output matrices *vsl* and *vsr*, respectively. Constraints:
 $ldvsl \geq 1$. If *jobvsl* = 'V', $ldvsl \geq \max(1, n)$.
 $ldvsr \geq 1$. If *jobvsr* = 'V', $ldvsr \geq \max(1, n)$.

lwork
INTEGER.
The dimension of the array *work*.
For real flavors:
If $n=0$ then $lwork \geq 1$.
If $n>0$ and *sense* = 'N', then $lwork \geq \max(8*n, 6*n+16)$.
If $n>0$ and *sense* = 'E', 'V', or 'B', then $lwork \geq \max(8*n, 6*n+16, 2*sdim*(n-sdim))$;
For complex flavors:
If $n=0$ then $lwork \geq 1$.
If $n>0$ and *sense* = 'N', then $lwork \geq \max(1, 2*n)$;

If $n > 0$ and $sense = 'E', 'V', \text{ or } 'B'$, then $lwork \geq \max(1, 2*n, 2*sdim*(n-sdim))$.

Note that $2*sdim*(n-sdim) \leq n*n/2$.

An error is only returned if $lwork < \max(8*n, 6*n+16)$ for real flavors, and $lwork < \max(1, 2*n)$ for complex flavors, but if $sense = 'E', 'V', \text{ or } 'B'$, this may not be large enough.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the bound on the optimal size of the *work* array and the minimum size of the *iwork* array, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla.

rwork

REAL for cggexx

DOUBLE PRECISION for zggesx

Workspace array, DIMENSION at least $\max(1, 8n)$.

This array is used in complex flavors only.

iwork

INTEGER.

Workspace array, DIMENSION $\max(1, liwork)$.

liwork

INTEGER.

The dimension of the array *iwork*.

If $sense = 'N'$, or $n = 0$, then $liwork \geq 1$,

otherwise $liwork \geq (n+6)$ for real flavors, and $liwork \geq (n+2)$ for complex flavors.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the bound on the optimal size of the *work* array and the minimum size of the *iwork* array, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla.

bwork

LOGICAL.

Workspace array, DIMENSION at least $\max(1, n)$.

Not referenced if $sort = 'N'$.

Output Parameters

<i>a</i>	On exit, this array has been overwritten by its generalized Schur form <i>S</i> .
<i>b</i>	On exit, this array has been overwritten by its generalized Schur form <i>T</i> .
<i>sdim</i>	INTEGER. If <i>sort</i> = 'N', <i>sdim</i> = 0. If <i>sort</i> = 'S', <i>sdim</i> is equal to the number of eigenvalues (after sorting) for which <i>selctg</i> is true. Note that for real flavors complex conjugate pairs for which <i>selctg</i> is true for either eigenvalue count as 2.
<i>alphar, alphas</i>	REAL for <i>sggesx</i> ; DOUBLE PRECISION for <i>dggesx</i> . Arrays, DIMENSION at least $\max(1, n)$ each. Contain values that form generalized eigenvalues in real flavors. See <i>beta</i> .
<i>alpha</i>	COMPLEX for <i>cggesx</i> ; DOUBLE COMPLEX for <i>zggesx</i> . Array, DIMENSION at least $\max(1, n)$. Contain values that form generalized eigenvalues in complex flavors. See <i>beta</i> .
<i>beta</i>	REAL for <i>sggesx</i> DOUBLE PRECISION for <i>dggesx</i> COMPLEX for <i>cggesx</i> DOUBLE COMPLEX for <i>zggesx</i> . Array, DIMENSION at least $\max(1, n)$. <i>For real flavors:</i> On exit, $(\text{alphar}(j) + \text{alphas}(j)*i)/\text{beta}(j)$, $j=1, \dots, n$ will be the generalized eigenvalues. $\text{alphar}(j) + \text{alphas}(j)*i$ and $\text{beta}(j)$, $j=1, \dots, n$ are the diagonals of the complex Schur form (<i>S</i> , <i>T</i>) that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of (<i>A</i> , <i>B</i>) were further reduced to triangular form using complex unitary transformations. If $\text{alphas}(j)$ is zero, then the <i>j</i> -th eigenvalue is real; if positive, then the <i>j</i> -th and (<i>j</i> +1)-st eigenvalues are a complex conjugate pair, with $\text{alphas}(j+1)$ negative. <i>For complex flavors:</i>

On exit, $\alpha(j)/\beta(j)$, $j=1,\dots, n$ will be the generalized eigenvalues. $\alpha(j)$, $j=1,\dots, n$, and $\beta(j)$, $j=1,\dots, n$ are the diagonals of the complex Schur form (S, T) output by `cggesx/zggesx`. The $\beta(j)$ will be non-negative real. See also *Application Notes* below.

vsl, vsr

REAL for `sggesx`
 DOUBLE PRECISION for `dggesx`
 COMPLEX for `cggesx`
 DOUBLE COMPLEX for `zggesx`.

Arrays:

`vsl(ldvsl,*)`, the second dimension of *vsl* must be at least $\max(1, n)$.

If `jobvsl = 'V'`, this array will contain the left Schur vectors.

If `jobvsl = 'N'`, *vsl* is not referenced.

`vsr(ldvsr,*)`, the second dimension of *vsr* must be at least $\max(1, n)$.

If `jobvsr = 'V'`, this array will contain the right Schur vectors.

If `jobvsr = 'N'`, *vsr* is not referenced.

rconde, rcondv

REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.

Arrays, DIMENSION (2) each

If `sense = 'E' or 'B'`, `rconde(1)` and `rconde(2)` contain the reciprocal condition numbers for the average of the selected eigenvalues.

Not referenced if `sense = 'N' or 'V'`.

If `sense = 'V' or 'B'`, `rcondv(1)` and `rcondv(2)` contain the reciprocal condition numbers for the selected deflating subspaces.

Not referenced if `sense = 'N' or 'E'`.

work(1)

On exit, if `info = 0`, then `work(1)` returns the required minimal size of *lwork*.

iwork(1)

On exit, if `info = 0`, then `iwork(1)` returns the required minimal size of *liwork*.

info

INTEGER.

If `info = 0`, the execution is successful.

If $info = -i$, the i th parameter had an illegal value.
 If $info = i$, and
 $i \leq n$:
 the QZ iteration failed. (A, B) is not in Schur form, but $alpha_r(j)$, $alpha_i(j)$ (for real flavors), or $alpha(j)$ (for complex flavors), and $beta(j)$, $j=info+1, \dots, n$ should be correct.
 $i > n$: errors that usually indicate LAPACK problems:
 $i = n+1$: other than QZ iteration failed in `?hgeqz`;
 $i = n+2$: after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the generalized Schur form no longer satisfy `selctg = .TRUE..`
 This could also be caused due to scaling;
 $i = n+3$: reordering failed in `?tgsen`.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ggesx` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (n, n) .
<i>b</i>	Holds the matrix <i>B</i> of size (n, n) .
<i>alphar</i>	Holds the vector of length (n) . Used in real flavors only.
<i>alphai</i>	Holds the vector of length (n) . Used in real flavors only.
<i>alpha</i>	Holds the vector of length (n) . Used in complex flavors only.
<i>beta</i>	Holds the vector of length (n) .
<i>vsl</i>	Holds the matrix <i>VSL</i> of size (n, n) .
<i>vsr</i>	Holds the matrix <i>VSR</i> of size (n, n) .
<i>rconde</i>	Holds the vector of length (2) .
<i>rcondv</i>	Holds the vector of length (2) .
<i>jobvsl</i>	Restored based on the presence of the argument <i>vsl</i> as follows: <code>jobvsl = 'V'</code> , if <i>vsl</i> is present, <code>jobvsl = 'N'</code> , if <i>vsl</i> is omitted.
<i>jobvsr</i>	Restored based on the presence of the argument <i>vsr</i> as follows:

	$jobvsr = 'V',$ if vsr is present, $jobvsr = 'N',$ if vsr is omitted.
$sort$	Restored based on the presence of the argument $select$ as follows: $sort = 'S',$ if $select$ is present, $sort = 'N',$ if $select$ is omitted.
$sense$	Restored based on the presence of arguments $rconde$ and $rcondv$ as follows: $sense = 'B',$ if both $rconde$ and $rcondv$ are present, $sense = 'E',$ if $rconde$ is present and $rcondv$ omitted, $sense = 'V',$ if $rconde$ is omitted and $rcondv$ present, $sense = 'N',$ if both $rconde$ and $rcondv$ are omitted.

Note that there will be an error condition if $rconde$ or $rcondv$ are present and $select$ is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of $lwork$ (or $liwork$) for the first run or set $lwork = -1$ ($liwork = -1$).

If you choose the first option and set any of admissible $lwork$ (or $liwork$) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array ($work, iwork$) on exit. Use this value ($work(1), iwork(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work, iwork$). This operation is called a workspace query.

Note that if you set $lwork$ ($liwork$) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The quotients $\alpha(j)/\beta(j)$ and $\alpha_{\text{hi}}(j)/\beta(j)$ may easily over- or underflow, and $\beta(j)$ may even be zero. Thus, you should avoid simply computing the ratio. However, α and α_{hi} will be always less than and usually comparable with $\text{norm}(A)$ in magnitude, and β always less than and usually comparable with $\text{norm}(B)$.

?ggeev

Computes the generalized eigenvalues, and the left and/or right generalized eigenvectors for a pair of nonsymmetric matrices.

Syntax

FORTRAN 77:

```
call sggev(jobvl, jobvr, n, a, lda, b, ldb, alphas, alphas, beta, vl, ldvl,
vr, ldvr, work, lwork, info)

call dggev(jobvl, jobvr, n, a, lda, b, ldb, alphas, alphas, beta, vl, ldvl,
vr, ldvr, work, lwork, info)

call cggev(jobvl, jobvr, n, a, lda, b, ldb, alpha, beta, vl, ldvl, vr, ldvr,
work, lwork, rwork, info)

call zggev(jobvl, jobvr, n, a, lda, b, ldb, alpha, beta, vl, ldvl, vr, ldvr,
work, lwork, rwork, info)
```

Fortran 95:

```
call ggev(a, b, alphas, alphas, beta [,vl] [,vr] [,info])
call ggev(a, b, alpha, beta [, vl] [,vr] [,info])
```

Description

This routine computes for a pair of n -by- n real/complex nonsymmetric matrices (A,B) , the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors.

A generalized eigenvalue for a pair of matrices (A,B) is a scalar λ or a ratio $\alpha / \beta = \lambda$, such that $A - \lambda B$ is singular. It is usually represented as the pair (α, β) , as there is a reasonable interpretation for $\beta = 0$ and even for both being zero.

The right generalized eigenvector $v(j)$ corresponding to the generalized eigenvalue $\lambda(j)$ of (A,B) satisfies

$$A * v(j) = \lambda(j) * B * v(j).$$

The left generalized eigenvector $u(j)$ corresponding to the generalized eigenvalue $\lambda(j)$ of (A,B) satisfies

$$u(j)^H A = \lambda(j) * u(j)^H B$$

where $u(j)^H$ denotes the conjugate transpose of $u(j)$.

Input Parameters

<i>jobvl</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobvl</i> = 'N', the left generalized eigenvectors are not computed; If <i>jobvl</i> = 'V', the left generalized eigenvectors are computed.
<i>jobvr</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobvr</i> = 'N', the right generalized eigenvectors are not computed; If <i>jobvr</i> = 'V', the right generalized eigenvectors are computed.
<i>n</i>	INTEGER. The order of the matrices <i>A</i>, <i>B</i>, <i>vl</i>, and <i>vr</i> ($n \geq 0$).
<i>a</i> , <i>b</i> , <i>work</i>	REAL for sggev DOUBLE PRECISION for dggev COMPLEX for cggev DOUBLE COMPLEX for zggev. Arrays: <i>a</i>(<i>lda</i>,*) is an array containing the <i>n</i>-by-<i>n</i> matrix <i>A</i> (first of the pair of matrices). The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>b</i>(<i>ldb</i>,*) is an array containing the <i>n</i>-by-<i>n</i> matrix <i>B</i> (second of the pair of matrices). The second dimension of <i>b</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The first dimension of the array <i>a</i>. Must be at least $\max(1, n)$.
<i>ldb</i>	INTEGER. The first dimension of the array <i>b</i>. Must be at least $\max(1, n)$.
<i>ldvl</i> , <i>ldvr</i>	INTEGER. The first dimensions of the output matrices <i>vl</i> and <i>vr</i>, respectively. Constraints:

$ldvl \geq 1$. If $jobvl = 'V'$, $ldvl \geq \max(1, n)$.
 $ldvr \geq 1$. If $jobvr = 'V'$, $ldvr \geq \max(1, n)$.
lwork INTEGER.
 The dimension of the array *work*.
 $lwork \geq \max(1, 8n+16)$ for real flavors;
 $lwork \geq \max(1, 2n)$ for complex flavors.
 For good performance, *lwork* must generally be larger.
 If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#).
rwork REAL for *cggev*
 DOUBLE PRECISION for *zggev*
 Workspace array, DIMENSION at least $\max(1, 8n)$.
 This array is used in complex flavors only.

Output Parameters

a, b On exit, these arrays have been overwritten.
alphar, alphas REAL for *sggev*;
 DOUBLE PRECISION for *dggev*.
 Arrays, DIMENSION at least $\max(1, n)$ each. Contain values that form generalized eigenvalues in real flavors.
 See *beta*.
alpha COMPLEX for *cggev*;
 DOUBLE COMPLEX for *zggev*.
 Array, DIMENSION at least $\max(1, n)$. Contain values that form generalized eigenvalues in complex flavors. See *beta*.
beta REAL for *sggev*
 DOUBLE PRECISION for *dggev*
 COMPLEX for *cggev*
 DOUBLE COMPLEX for *zggev*.
 Array, DIMENSION at least $\max(1, n)$.
 For real flavors:
 On exit, $(\text{alphar}(j) + \text{alphai}(j)*i)/\text{beta}(j)$, $j=1, \dots, n$, are the generalized eigenvalues.

vl, vr

If $\alpha_i(j)$ is zero, then the j -th eigenvalue is real; if positive, then the j -th and $(j+1)$ -st eigenvalues are a complex conjugate pair, with $\alpha_i(j+1)$ negative.

For complex flavors:

On exit, $\alpha(j)/\beta(j)$, $j=1, \dots, n$, are the generalized eigenvalues.

See also *Application Notes* below.

REAL for sggev
DOUBLE PRECISION for dggev
COMPLEX for cggev
DOUBLE COMPLEX for zggev.

Arrays:

$vl(ldvl,*)$; the second dimension of vl must be at least $\max(1, n)$.

If $jobvl = 'V'$, the left generalized eigenvectors $u(j)$ are stored one after another in the columns of vl , in the same order as their eigenvalues. Each eigenvector is scaled so the largest component has $\text{abs}(\text{Re}) + \text{abs}(\text{Im}) = 1$.

If $jobvl = 'N'$, vl is not referenced.

For real flavors:

If the j -th eigenvalue is real, then $u(j) = vl(:, j)$, the j -th column of vl .

If the j -th and $(j+1)$ -st eigenvalues form a complex conjugate pair, then $u(j) = vl(:, j) + i*vl(:, j+1)$ and $u(j+1) = vl(:, j) - i*vl(:, j+1)$, where $i = \text{sqrt}(-1)$.

For complex flavors:

$u(j) = vl(:, j)$, the j -th column of vl .

$vr(ldvr,*)$; the second dimension of vr must be at least $\max(1, n)$.

If $jobvr = 'V'$, the right generalized eigenvectors $v(j)$ are stored one after another in the columns of vr , in the same order as their eigenvalues. Each eigenvector is scaled so the largest component has $\text{abs}(\text{Re}) + \text{abs}(\text{Im}) = 1$.

If $jobvr = 'N'$, vr is not referenced.

For real flavors:

If the j -th eigenvalue is real, then $v(j) = vr(:, j)$, the j -th column of vr .

If the j -th and $(j+1)$ -st eigenvalues form a complex conjugate pair, then $v(j) = vr(:,j) + i*vr(:,j+1)$ and $v(j+1) = vr(:,j) - i*vr(:,j+1)$.
For complex flavors:
 $v(j) = vr(:,j)$, the j -th column of vr .

work(1) On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.
 If *info* = *i*, and
 i ≤ *n*: the QZ iteration failed. No eigenvectors have been calculated, but *alphar*(*j*), *alphai*(*j*) (for real flavors), or *alpha*(*j*) (for complex flavors), and *beta*(*j*), *j*=*info*+1, ..., *n* should be correct.
 i > *n*: errors that usually indicate LAPACK problems:
 i = *n*+1: other than QZ iteration failed in ?hgeqz;
 i = *n*+2: error return from ?tgevc.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *ggeev* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>n</i>).
<i>alphar</i>	Holds the vector of length (<i>n</i>). Used in real flavors only.
<i>alphai</i>	Holds the vector of length (<i>n</i>). Used in real flavors only.
<i>alpha</i>	Holds the vector of length (<i>n</i>). Used in complex flavors only.
<i>beta</i>	Holds the vector of length (<i>n</i>).
<i>vl</i>	Holds the matrix <i>VL</i> of size (<i>n</i> , <i>n</i>).
<i>vr</i>	Holds the matrix <i>VR</i> of size (<i>n</i> , <i>n</i>).
<i>jobvl</i>	Restored based on the presence of the argument <i>vl</i> as follows: <i>jobvl</i> = 'V', if <i>vl</i> is present,

jobvl = 'N', if *vl* is omitted.

jobvr Restored based on the presence of the argument *vr* as follows:

jobvr = 'V', if *vr* is present,

jobvr = 'N', if *vr* is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The quotients *alphar*(j)/*beta*(j) and *alphai*(j)/*beta*(j) may easily over- or underflow, and *beta*(j) may even be zero. Thus, you should avoid simply computing the ratio. However, *alphar* and *alphai* (for real flavors) or *alpha* (for complex flavors) will be always less than and usually comparable with *norm(A)* in magnitude, and *beta* always less than and usually comparable with *norm(B)*.

?ggev

Computes the generalized eigenvalues, and, optionally, the left and/or right generalized eigenvectors.

Syntax

FORTRAN 77:

```
call sggev(balanc, jobvl, jobvr, sense, n, a, lda, b, ldb, alphas, alphas,
beta, vl, ldvl, vr, ldvr, ilo, ihi, lscale, rscale, abnrm, bbnrm, rconde,
rcondv, work, lwork, iwork, bwork, info)

call dggev(balanc, jobvl, jobvr, sense, n, a, lda, b, ldb, alphas, alphas,
beta, vl, ldvl, vr, ldvr, ilo, ihi, lscale, rscale, abnrm, bbnrm, rconde,
rcondv, work, lwork, iwork, bwork, info)

call cggev(balanc, jobvl, jobvr, sense, n, a, lda, b, ldb, alpha, beta, vl,
ldvl, vr, ldvr, ilo, ihi, lscale, rscale, abnrm, bbnrm, rconde, rcondv, work,
lwork, rwork, iwork, bwork, info)

call zggev(balanc, jobvl, jobvr, sense, n, a, lda, b, ldb, alpha, beta, vl,
ldvl, vr, ldvr, ilo, ihi, lscale, rscale, abnrm, bbnrm, rconde, rcondv, work,
lwork, rwork, iwork, bwork, info)
```

Fortran 95:

```
call ggev(a, b, alphas, alphas, beta [,vl] [,vr] [,balanc] [,ilo] [,ihi] [,
lscale] [,rscale] [,abnrm] [,bbnrm] [,rconde] [,rcondv] [,info])

call ggev(a, b, alpha, beta [, vl] [,vr] [,balanc] [,ilo] [,ihi] [,lscale]
[, rscale] [,abnrm] [,bbnrm] [,rconde] [,rcondv] [,info])
```

Description

This routine computes for a pair of n -by- n real/complex nonsymmetric matrices (A,B) , the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors.

Optionally also, it computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (*ilo*, *ihi*, *lscale*, *rscale*, *abnrm*, and *bbnrm*), reciprocal condition numbers for the eigenvalues (*rconde*), and reciprocal condition numbers for the right eigenvectors (*rcondv*).

A generalized eigenvalue for a pair of matrices (A, B) is a scalar λ or a ratio $\alpha / \beta = \lambda$, such that $A - \lambda B$ is singular. It is usually represented as the pair (α, β) , as there is a reasonable interpretation for $\beta=0$ and even for both being zero. The right generalized eigenvector $v(j)$ corresponding to the generalized eigenvalue $\lambda(j)$ of (A, B) satisfies

$$A * v(j) = \lambda(j) * B * v(j).$$

The left generalized eigenvector $u(j)$ corresponding to the generalized eigenvalue $\lambda(j)$ of (A, B) satisfies

$$u(j)^H * A = \lambda(j) * u(j)^H * B$$

where $u(j)^H$ denotes the conjugate transpose of $u(j)$.

Input Parameters

<i>balanc</i>	CHARACTER*1. Must be 'N', 'P', 'S', or 'B'. Specifies the balance option to be performed. If <i>balanc</i> = 'N', do not diagonally scale or permute; If <i>balanc</i> = 'P', permute only; If <i>balanc</i> = 'S', scale only; If <i>balanc</i> = 'B', both permute and scale. Computed reciprocal condition numbers will be for the matrices after balancing and/or permuting. Permuting does not change condition numbers (in exact arithmetic), but balancing does.
<i>jobvl</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobvl</i> = 'N', the left generalized eigenvectors are not computed; If <i>jobvl</i> = 'V', the left generalized eigenvectors are computed.
<i>jobvr</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobvr</i> = 'N', the right generalized eigenvectors are not computed; If <i>jobvr</i> = 'V', the right generalized eigenvectors are computed.
<i>sense</i>	CHARACTER*1. Must be 'N', 'E', 'V', or 'B'. Determines which reciprocal condition number are computed. If <i>sense</i> = 'N', none are computed;

If *sense* = 'E', computed for eigenvalues only;
 If *sense* = 'V', computed for eigenvectors only;
 If *sense* = 'B', computed for eigenvalues and eigenvectors.

n INTEGER. The order of the matrices *A*, *B*, *vl*, and *vr* ($n \geq 0$).

a, *b*, *work* REAL for sggevx
 DOUBLE PRECISION for dggevx
 COMPLEX for cggevx
 DOUBLE COMPLEX for zggevx.

Arrays:
a(*lda*,*) is an array containing the *n*-by-*n* matrix *A* (first of the pair of matrices).
 The second dimension of *a* must be at least $\max(1, n)$.
b(*ldb*,*) is an array containing the *n*-by-*n* matrix *B* (second of the pair of matrices).
 The second dimension of *b* must be at least $\max(1, n)$.
work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The first dimension of the array *a*.
 Must be at least $\max(1, n)$.

ldb INTEGER. The first dimension of the array *b*.
 Must be at least $\max(1, n)$.

ldvl, *ldvr* INTEGER. The first dimensions of the output matrices *vl* and *vr*, respectively.

Constraints:
 $ldvl \geq 1$. If *jobvl* = 'V', $ldvl \geq \max(1, n)$.
 $ldvr \geq 1$. If *jobvr* = 'V', $ldvr \geq \max(1, n)$.

lwork INTEGER.
 The dimension of the array *work*. $lwork \geq \max(1, 2*n)$;
For real flavors:
 If *balanc* = 'S', or 'B', or *jobvl* = 'V', or *jobvr* = 'V', then $lwork \geq \max(1, 6*n)$;
 if *sense* = 'E', or 'B', then $lwork \geq \max(1, 10*n)$;
 if *sense* = 'V', or 'B', $lwork \geq (2n^2 + 8*n + 16)$.
For complex flavors:

if *sense* = 'E', *lwork* $\geq \max(1, 4*n)$;
 if *sense* = 'V', or 'B', *lwork* $\geq \max(1, 2*n^2 + 2*n)$.
 If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#).

rwork REAL for *cggev*x
 DOUBLE PRECISION for *zggev*x
 Workspace array, DIMENSION at least $\max(1, 6*n)$ if *balanc* = 'S', or 'B', and at least $\max(1, 2*n)$ otherwise.
 This array is used in complex flavors only.

iwork INTEGER.
 Workspace array, DIMENSION at least $(n+6)$ for real flavors and at least $(n+2)$ for complex flavors.
 Not referenced if *sense* = 'E'.

bwork LOGICAL. Workspace array, DIMENSION at least $\max(1, n)$.
 Not referenced if *sense* = 'N'.

Output Parameters

a, b On exit, these arrays have been overwritten.
 If *jobvl* = 'V' or *jobvr* = 'V' or both, then *a* contains the first part of the real Schur form of the “balanced” versions of the input *A* and *B*, and *b* contains its second part.

alphas, *alphai* REAL for *sggev*x;
 DOUBLE PRECISION for *dggev*x.
 Arrays, DIMENSION at least $\max(1, n)$ each. Contain values that form generalized eigenvalues in real flavors.
 See *beta*.

alpha COMPLEX for *cggev*x;
 DOUBLE COMPLEX for *zggev*x.
 Array, DIMENSION at least $\max(1, n)$. Contain values that form generalized eigenvalues in complex flavors. See *beta*.

beta REAL for *sggev*x
 DOUBLE PRECISION for *dggev*x
 COMPLEX for *cggev*x

v_l, v_r

DOUBLE COMPLEX for zggevz.

Array, DIMENSION at least $\max(1, n)$.

For real flavors:

On exit, $(\alpha(j) + \beta(j)i)/\gamma(j)$, $j=1, \dots, n$, will be the generalized eigenvalues.

If $\beta(j)$ is zero, then the j -th eigenvalue is real; if positive, then the j -th and $(j+1)$ -st eigenvalues are a complex conjugate pair, with $\beta(j+1)$ negative.

For complex flavors:

On exit, $\alpha(j)/\gamma(j)$, $j=1, \dots, n$, will be the generalized eigenvalues.

See also *Application Notes* below.

REAL for sggevz

DOUBLE PRECISION for dggevz

COMPLEX for cggevz

DOUBLE COMPLEX for zggevz.

Arrays:

$v_l(ldv_l, *)$; the second dimension of v_l must be at least $\max(1, n)$.

If $jobv_l = 'V'$, the left generalized eigenvectors $u(j)$ are stored one after another in the columns of v_l , in the same order as their eigenvalues. Each eigenvector will be scaled so the largest component have $\text{abs}(\text{Re}) + \text{abs}(\text{Im}) = 1$.

If $jobv_l = 'N'$, v_l is not referenced.

For real flavors:

If the j -th eigenvalue is real, then $u(j) = v_l(:, j)$, the j -th column of v_l .

If the j -th and $(j+1)$ -st eigenvalues form a complex conjugate pair, then $u(j) = v_l(:, j) + i*v_l(:, j+1)$ and $u(j+1) = v_l(:, j) - i*v_l(:, j+1)$, where $i = \text{sqrt}(-1)$.

For complex flavors:

$u(j) = v_l(:, j)$, the j -th column of v_l .

$v_r(ldv_r, *)$; the second dimension of v_r must be at least $\max(1, n)$.

If $jobvr = 'V'$, the right generalized eigenvectors $v(j)$ are stored one after another in the columns of vr , in the same order as their eigenvalues. Each eigenvector will be scaled so the largest component have $abs(Re) + abs(Im) = 1$.

If $jobvr = 'N'$, vr is not referenced.

For real flavors:

If the j -th eigenvalue is real, then $v(j) = vr(:, j)$, the j -th column of vr .

If the j -th and $(j+1)$ -st eigenvalues form a complex conjugate pair, then $v(j) = vr(:, j) + i*vr(:, j+1)$ and $v(j+1) = vr(:, j) - i*vr(:, j+1)$.

For complex flavors:

$v(j) = vr(:, j)$, the j -th column of vr .

ilo, ihi

INTEGER. *ilo* and *ihi* are integer values such that on exit $A(i, j) = 0$ and $B(i, j) = 0$ if $i > j$ and $j = 1, \dots, ilo-1$ or $i = ihi+1, \dots, n$.

If $balanc = 'N'$ or $'S'$, $ilo = 1$ and $ihi = n$.

lscale, rscale

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Arrays, DIMENSION at least $\max(1, n)$ each.

lscale contains details of the permutations and scaling factors applied to the left side of A and B .

If $PL(j)$ is the index of the row interchanged with row j , and $DL(j)$ is the scaling factor applied to row j , then

$lscale(j) = PL(j)$, for $j = 1, \dots, ilo-1$
 $= DL(j)$, for $j = ilo, \dots, ihi$
 $= PL(j)$ for $j = ihi+1, \dots, n$.

The order in which the interchanges are made is n to $ihi+1$, then 1 to $ilo-1$.

rscale contains details of the permutations and scaling factors applied to the right side of A and B .

If $PR(j)$ is the index of the column interchanged with column j , and $DR(j)$ is the scaling factor applied to column j , then

$rscale(j) = PR(j)$, for $j = 1, \dots, ilo-1$
 $= DR(j)$, for $j = ilo, \dots, ihi$
 $= PR(j)$ for $j = ihi+1, \dots, n$.

The order in which the interchanges are made is n to $ihi+1$, then 1 to $ilo-1$.

abnrm, bbnrm REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
The one-norms of the balanced matrices A and B , respectively.

rconde, rcondv REAL for single precision flavors DOUBLE PRECISION for double precision flavors.
Arrays, DIMENSION at least $\max(1, n)$ each.
If $sense = 'E'$, or $'B'$, *rconde* contains the reciprocal condition numbers of the eigenvalues, stored in consecutive elements of the array. For a complex conjugate pair of eigenvalues two consecutive elements of *rconde* are set to the same value. Thus *rconde*(j), *rcondv*(j), and the j -th columns of v_l and v_r all correspond to the same eigenpair (but not in general the j -th eigenpair, unless all eigenpairs are selected).
If $sense = 'N'$, or $'V'$, *rconde* is not referenced.
If $sense = 'V'$, or $'B'$, *rcondv* contains the estimated reciprocal condition numbers of the eigenvectors, stored in consecutive elements of the array. For a complex eigenvector two consecutive elements of *rcondv* are set to the same value.
If the eigenvalues cannot be reordered to compute *rcondv*(j), *rcondv*(j) is set to 0; this can only occur when the true value would be very small anyway.
If $sense = 'N'$, or $'E'$, *rcondv* is not referenced.

work(1) On exit, if $info = 0$, then *work(1)* returns the required minimal size of *lwork*.

info INTEGER.
If $info = 0$, the execution is successful.
If $info = -i$, the i -th parameter had an illegal value.
If $info = i$, and
 $i \leq n$:

the *QZ* iteration failed. No eigenvectors have been calculated, but *alphar*(*j*), *alphai*(*j*) (for real flavors), or *alpha*(*j*) (for complex flavors), and *beta*(*j*), *j*=*info*+1, ..., *n* should be correct.

i > *n*: errors that usually indicate LAPACK problems:

i = *n*+1: other than *QZ* iteration failed in ?*hgeqz*;

i = *n*+2: error return from ?*tgevc*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *ggevx* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>n</i>).
<i>alphar</i>	Holds the vector of length (<i>n</i>). Used in real flavors only.
<i>alphai</i>	Holds the vector of length (<i>n</i>). Used in real flavors only.
<i>alpha</i>	Holds the vector of length (<i>n</i>). Used in complex flavors only.
<i>beta</i>	Holds the vector of length (<i>n</i>).
<i>vl</i>	Holds the matrix <i>VL</i> of size (<i>n</i> , <i>n</i>).
<i>vr</i>	Holds the matrix <i>VR</i> of size (<i>n</i> , <i>n</i>).
<i>lscale</i>	Holds the vector of length (<i>n</i>).
<i>rscale</i>	Holds the vector of length (<i>n</i>).
<i>rconde</i>	Holds the vector of length (<i>n</i>).
<i>rcondv</i>	Holds the vector of length (<i>n</i>).
<i>balanc</i>	Must be 'N', 'B', or 'P'. The default value is 'N'.
<i>jobvl</i>	Restored based on the presence of the argument <i>vl</i> as follows: <i>jobvl</i> = 'V', if <i>vl</i> is present, <i>jobvl</i> = 'N', if <i>vl</i> is omitted.
<i>jobvr</i>	Restored based on the presence of the argument <i>vr</i> as follows: <i>jobvr</i> = 'V', if <i>vr</i> is present, <i>jobvr</i> = 'N', if <i>vr</i> is omitted.

sense Restored based on the presence of arguments *rconde* and *rcondv* as follows:

sense = 'B', if both *rconde* and *rcondv* are present,
sense = 'E', if *rconde* is present and *rcondv* omitted,
sense = 'V', if *rconde* is omitted and *rcondv* present,
sense = 'N', if both *rconde* and *rcondv* are omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The quotients *alphar*(j)/*beta*(j) and *alphai*(j)/*beta*(j) may easily over- or underflow, and *beta*(j) may even be zero. Thus, you should avoid simply computing the ratio. However, *alphar* and *alphai* (for real flavors) or *alpha* (for complex flavors) will be always less than and usually comparable with *norm*(A) in magnitude, and *beta* always less than and usually comparable with *norm*(B).

LAPACK Auxiliary and Utility Routines

5

This chapter describes the Intel® Math Kernel Library implementation of LAPACK [auxiliary](#) and [utility routines](#). The library includes auxiliary routines for both real and complex data.

Auxiliary Routines

Routine naming conventions, mathematical notation, and matrix storage schemes used for LAPACK auxiliary routines are the same as for the driver and computational routines described in previous chapters.

The table below summarizes information about the available LAPACK auxiliary routines.

Table 5-1 LAPACK Auxiliary Routines

Routine Name	Data Types	Description
?lacgv	c, z	Conjugates a complex vector.
?lacrm	c, z	Multiplies a complex matrix by a square real matrix.
?lacrt	c, z	Performs a linear transformation of a pair of complex vectors.
?laesy	c, z	Computes the eigenvalues and eigenvectors of a 2-by-2 complex symmetric matrix.
?rot	c, z	Applies a plane rotation with real cosine and complex sine to a pair of complex vectors.
?spmv	c, z	Computes a matrix-vector product for complex vectors using a complex symmetric packed matrix
?spr	c, z	Performs the symmetrical rank-1 update of a complex symmetric packed matrix.
?symv	c, z	Computes a matrix-vector product for a complex symmetric matrix.
?syr	c, z	Performs the symmetric rank-1 update of a complex symmetric matrix.
i?max1	c, z	Finds the index of the vector element whose real part has maximum absolute value.

Routine Name	Data Types	Description
?sum1	s, d, dz	Forms the 1-norm of the complex vector using the true absolute value.
?gbtf2	s, d, c, z	Computes the LU factorization of a general band matrix using the unblocked version of the algorithm.
?gebd2	s, d, c, z	Reduces a general matrix to bidiagonal form using an unblocked algorithm.
?gehd2	s, d, c, z	Reduces a general square matrix to upper Hessenberg form using an unblocked algorithm.
?gelq2	s, d, c, z	Computes the LQ factorization of a general rectangular matrix using an unblocked algorithm.
?geql2	s, d, c, z	Computes the QL factorization of a general rectangular matrix using an unblocked algorithm.
?geqr2	s, d, c, z	Computes the QR factorization of a general rectangular matrix using an unblocked algorithm.
?gerq2	s, d, c, z	Computes the RQ factorization of a general rectangular matrix using an unblocked algorithm.
?gesc2	s, d, c, z	Solves a system of linear equations using the LU factorization with complete pivoting computed by ?getc2 .
?getc2	s, d, c, z	Computes the LU factorization with complete pivoting of the general n-by-n matrix.
?getf2	s, d, c, z	Computes the LU factorization of a general m -by- n matrix using partial pivoting with row interchanges (unblocked algorithm).
?gtts2	s, d, c, z	Solves a system of linear equations with a tridiagonal matrix using the LU factorization computed by ?gttrf .
?isnan	s, d,	Tests input for NaN.
?laisnan	s, d,	Tests input for NaN by comparing it two arguments for inequality.

Routine Name	Data Types	Description
?labrd	s, d, c, z	Reduces the first nb rows and columns of a general matrix to a bidiagonal form.
?laln2	s, d, c, z	Estimates the 1-norm of a square matrix, using reverse communication for evaluating matrix-vector products.
?lacon	s, d, c, z	Estimates the 1-norm of a square matrix, using reverse communication for evaluating matrix-vector products.
?lacpy	s, d, c, z	Copies all or part of one two-dimensional array to another.
?ladiv	s, d, c, z	Performs complex division in real arithmetic, avoiding unnecessary overflow.
?lae2	s, d	Computes the eigenvalues of a 2-by-2 symmetric matrix.
?laebz	s, d	Computes the number of eigenvalues of a real symmetric tridiagonal matrix which are less than or equal to a given value, and performs other tasks required by the routine ?stebz .
?laed0	s, d, c, z	Used by ?stedc . Computes all eigenvalues and corresponding eigenvectors of an unreduced symmetric tridiagonal matrix using the divide and conquer method.
?laed1	s, d	Used by sstedc / dstedc . Computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. Used when the original matrix is tridiagonal.
?laed2	s, d	Used by sstedc / dstedc . Merges eigenvalues and deflates secular equation. Used when the original matrix is tridiagonal.
?laed3	s, d	Used by sstedc / dstedc . Finds the roots of the secular equation and updates the eigenvectors. Used when the original matrix is tridiagonal.
?laed4	s, d	Used by sstedc / dstedc . Finds a single root of the secular equation.
?laed5	s, d	Used by sstedc / dstedc . Solves the 2-by-2 secular equation.

Routine Name	Data Types	Description
?laed6	s, d	Used by <code>sstedc/dstedc</code> . Computes one Newton step in solution of the secular equation.
?laed7	s, d, c, z	Used by <code>?stedc</code> . Computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. Used when the original matrix is dense.
?laed8	s, d, c, z	Used by <code>?stedc</code> . Merges eigenvalues and deflates secular equation. Used when the original matrix is dense.
?laed9	s, d	Used by <code>sstedc/dstedc</code> . Finds the roots of the secular equation and updates the eigenvectors. Used when the original matrix is dense.
?laeda	s, d	Used by <code>?stedc</code> . Computes the z vector determining the rank-one modification of the diagonal matrix. Used when the original matrix is dense.
?laein	s, d, c, z	Computes a specified right or left eigenvector of an upper Hessenberg matrix by inverse iteration.
?laev2	s, d, c, z	Computes the eigenvalues and eigenvectors of a 2-by-2 symmetric/Hermitian matrix.
?laexc	s, d	Swaps adjacent diagonal blocks of a real upper quasi-triangular matrix in Schur canonical form, by an orthogonal similarity transformation.
?lag2	s, d	Computes the eigenvalues of a 2-by-2 generalized eigenvalue problem, with scaling as necessary to avoid over-/underflow.
?lags2	s, d	Computes 2-by-2 orthogonal matrices U , V , and Q , and applies them to matrices A and B such that the rows of the transformed A and B are parallel.
?lagtf	s, d	Computes an LU factorization of a matrix $T-\lambda I$, where T is a general tridiagonal matrix, and λ a scalar, using partial pivoting with row interchanges.

Routine Name	Data Types	Description
?lagtm	s, d, c, z	Performs a matrix-matrix product of the form $C = \alpha ab + \beta C$, where A is a tridiagonal matrix, B and C are rectangular matrices, and α and β are scalars, which may be 0, 1, or -1.
?lagts	s, d	Solves the system of equations $(T - \lambda I)x = y$ or $(T - \lambda I)^T x = y$, where T is a general tridiagonal matrix and λ a scalar, using the LU factorization computed by ?lagtf .
?lagv2	s, d	Computes the Generalized Schur factorization of a real 2-by-2 matrix pencil (A, B) where B is upper triangular.
?lahqr	s, d, c, z	Computes the eigenvalues and Schur factorization of an upper Hessenberg matrix, using the double-shift/single-shift QR algorithm.
?lahrd	s, d, c, z	Reduces the first nb columns of a general rectangular matrix A so that elements below the k -th subdiagonal are zero, and returns auxiliary matrices which are needed to apply the transformation to the unreduced part of A .
?lahr2	s, d, c, z	Reduces the specified number of first columns of a general rectangular matrix A so that elements below the specified subdiagonal are zero, and returns auxiliary matrices which are needed to apply the transformation to the unreduced part of A .
?laic1	s, d, c, z	Applies one step of incremental condition estimation.
?laln2	s, d	Solves a 1-by-1 or 2-by-2 linear system of equations of the specified form.
?lals0	s, d, c, z	Applies back multiplying factors in solving the least squares problem using divide and conquer SVD approach. Used by ?gelsd .
?lalsa	s, d, c, z	Computes the SVD of the coefficient matrix in compact form. Used by ?gelsd .
?lalsd	s, d, c, z	Uses the singular value decomposition of A to solve the least squares problem.

Routine Name	Data Types	Description
?lamrg	s, d	Creates a permutation list to merge the entries of two independently sorted sets into a single set sorted in ascending order.
?laneg	s, d	Computes the Sturm count.
?langb	s, d, c, z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of general band matrix.
?lange	s, d, c, z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a general rectangular matrix.
?langt	s, d, c, z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a general tridiagonal matrix.
?lanhs	s, d, c, z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of an upper Hessenberg matrix.
?lansb	s, d, c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric band matrix.
?lanhb	c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a Hermitian band matrix.
?lansp	s, d, c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric matrix supplied in packed form.
?lanhp	c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix supplied in packed form.
?lanst/?lanht	s, d/c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real symmetric or complex Hermitian tridiagonal matrix.

Routine Name	Data Types	Description
?lansy	s, d, c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex symmetric matrix.
?lanhe	c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix.
?lantb	s, d, c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular band matrix.
?lantp	s, d, c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular matrix supplied in packed form.
?lantr	s, d, c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular matrix.
?lanv2	s, d	Computes the Schur factorization of a real 2-by-2 nonsymmetric matrix in standard form.
?lap11	s, d, c, z	Measures the linear dependence of two vectors.
?lapmt	s, d, c, z	Performs a forward or backward permutation of the columns of a matrix.
?lapy2	s, d	Returns $\sqrt{x^2+y^2}$.
?lapy3	s, d	Returns $\sqrt{x^2+y^2+z^2}$.
?laqgb	s, d, c, z	Scales a general band matrix, using row and column scaling factors computed by ?gbequ .
?laqge	s, d, c, z	Scales a general rectangular matrix, using row and column scaling factors computed by ?geequ .
?laqhb	c, z	Scales a Hermetian band matrix, using scaling factors computed by ?pbequ .

Routine Name	Data Types	Description
?laqp2	s, d, c, z	Computes a QR factorization with column pivoting of the matrix block.
?laqps	s, d, c, z	Computes a step of QR factorization with column pivoting of a real m-by-n matrix <i>A</i> by using BLAS level 3.
?laqr0	s, d, c, z	Computes the eigenvalues of a Hessenberg matrix, and optionally the matrices from the Schur decomposition.
?laqr1	s, d, c, z	Sets a scalar multiple of the first column of the product of 2-by-2 or 3-by-3 matrix <i>H</i> and specified shifts.
?laqr2	s, d, c, z	Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).
?laqr3	s, d, c, z	Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).
?laqr4	s, d, c, z	Computes the eigenvalues of a Hessenberg matrix, and optionally the matrices from the Schur decomposition.
?laqr5	s, d, c, z	Performs a single small-bulge multi-shift QR sweep.
?laqsb	s, d, c, z	Scales a symmetric/Hermitian band matrix, using scaling factors computed by ?pbequ .
?laqsp	s, d, c, z	Scales a symmetric/Hermitian matrix in packed storage, using scaling factors computed by ?ppequ .
?laqsy	s, d, c, z	Scales a symmetric/Hermitian matrix, using scaling factors computed by ?poequ .
?laqtr	s, d	Solves a real quasi-triangular system of equations, or a complex quasi-triangular system of special form, in real arithmetic.

Routine Name	Data Types	Description
?lar1v	s, d, c, z	Computes the (scaled) r -th column of the inverse of the submatrix in rows $b1$ through bn of the tridiagonal matrix $1dL^T - \sigma I$.
?lar2v	s, d, c, z	Applies a vector of plane rotations with real cosines and real/complex sines from both sides to a sequence of 2-by-2 symmetric/Hermitian matrices.
?larf	s, d, c, z	Applies an elementary reflector to a general rectangular matrix.
?larfb	s, d, c, z	Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.
?larfg	s, d, c, z	Generates an elementary reflector (Householder matrix).
?larft	s, d, c, z	Forms the triangular factor T of a block reflector $H = I - vtv^H$.
?larfx	s, d, c, z	Applies an elementary reflector to a general rectangular matrix, with loop unrolling when the reflector has order ≤ 10 .
?largv	s, d, c, z	Generates a vector of plane rotations with real cosines and real/complex sines.
?larnv	s, d, c, z	Returns a vector of random numbers from a uniform or normal distribution.
?larra	s, d	Computes the splitting points with the specified threshold.
?larrb	s, d	Provides limited bisection to locate eigenvalues for more accuracy.
?larrc	s, d	Computes the number of eigenvalues of the symmetric tridiagonal matrix.
?larrd	s, d	Computes the eigenvalues of a symmetric tridiagonal matrix to suitable accuracy.
?larre	s, d	Given the tridiagonal matrix T , sets small off-diagonal elements to zero and for each unreduced block T_i , finds base representations and eigenvalues.

Routine Name	Data Types	Description
?larrf	s, d	Finds a new relatively robust representation such that at least one of the eigenvalues is relatively isolated.
?larrj	s, d	Performs refinement of the initial estimates of the eigenvalues of the matrix T .
?larrk	s, d	Computes one eigenvalue of a symmetric tridiagonal matrix T to suitable accuracy.
?larrrr	s, d	Performs tests to decide whether the symmetric tridiagonal matrix T warrants expensive computations which guarantee high relative accuracy in the eigenvalues.
?larrv	s, d, c, z	Computes the eigenvectors of the tridiagonal matrix $T = L D L^T$ given L , D and the eigenvalues of $L D L^T$.
?lartg	s, d, c, z	Generates a plane rotation with real cosine and real/complex sine.
?lartv	s, d, c, z	Applies a vector of plane rotations with real cosines and real/complex sines to the elements of a pair of vectors.
?laruv	s, d	Returns a vector of n random real numbers from a uniform distribution.
?larz	s, d, c, z	Applies an elementary reflector (as returned by ?tzzrf) to a general matrix.
?larzb	s, d, c, z	Applies a block reflector or its transpose/conjugate-transpose to a general matrix.
?larzt	s, d, c, z	Forms the triangular factor T of a block reflector $H = I - vtv^H$.
?las2	s, d	Computes singular values of a 2-by-2 triangular matrix.
?lascl	s, d, c, z	Multiplies a general rectangular matrix by a real scalar defined as c_{to}/c_{from} .
?lasd0	s, d	Computes the singular values of a real upper bidiagonal n-by-m matrix B with diagonal d and off-diagonal e . Used by ?bdsdc .

Routine Name	Data Types	Description
?lasd1	s, d	Computes the SVD of an upper bidiagonal matrix B of the specified size. Used by ?bdsdc .
?lasd2	s, d	Merges the two sets of singular values together into a single sorted set. Used by ?bdsdc .
?lasd3	s, d	Finds all square roots of the roots of the secular equation, as defined by the values in D and Z , and then updates the singular vectors by matrix multiplication. Used by ?bdsdc .
?lasd4	s, d	Computes the square root of the i -th updated eigenvalue of a positive symmetric rank-one modification to a positive diagonal matrix. Used by ?bdsdc .
?lasd5	s, d	Computes the square root of the i -th eigenvalue of a positive symmetric rank-one modification of a 2-by-2 diagonal matrix. Used by ?bdsdc .
?lasd6	s, d	Computes the SVD of an updated upper bidiagonal matrix obtained by merging two smaller ones by appending a row. Used by ?bdsdc .
?lasd7	s, d	Merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. Used by ?bdsdc .
?lasd8	s, d	Finds the square roots of the roots of the secular equation, and stores, for each element in D , the distance to its two nearest poles. Used by ?bdsdc .
?lasd9	s, d	Finds the square roots of the roots of the secular equation, and stores, for each element in D , the distance to its two nearest poles. Used by ?bdsdc .
?lasda	s, d	Computes the singular value decomposition (SVD) of a real upper bidiagonal matrix with diagonal d and off-diagonal e . Used by ?bdsdc .
?lasdq	s, d	Computes the SVD of a real bidiagonal matrix with diagonal d and off-diagonal e . Used by ?bdsdc .

Routine Name	Data Types	Description
<code>?lasdt</code>	<code>s, d</code>	Creates a tree of subproblems for bidiagonal divide and conquer. Used by <code>?bdsdc</code> .
<code>?laset</code>	<code>s, d, c, z</code>	Initializes the off-diagonal elements and the diagonal elements of a matrix to given values.
<code>?lasq1</code>	<code>s, d</code>	Computes the singular values of a real square bidiagonal matrix. Used by <code>?bdsqr</code> .
<code>?lasq2</code>	<code>s, d</code>	Computes all the eigenvalues of the symmetric positive definite tridiagonal matrix associated with the <code>qd</code> Array <code>z</code> to high relative accuracy. Used by <code>?bdsqr</code> and <code>?stegr</code> .
<code>?lasq3</code>	<code>s, d</code>	Checks for deflation, computes a shift and calls <code>dqds</code> . Used by <code>?bdsqr</code> .
<code>?lasq4</code>	<code>s, d</code>	Computes an approximation to the smallest eigenvalue using values of <code>d</code> from the previous transform. Used by <code>?bdsqr</code> .
<code>?lasq5</code>	<code>s, d</code>	Computes one <code>dqds</code> transform in ping-pong form. Used by <code>?bdsqr</code> and <code>?stegr</code> .
<code>?lasq6</code>	<code>s, d</code>	Computes one <code>dqd</code> transform in ping-pong form. Used by <code>?bdsqr</code> and <code>?stegr</code> .
<code>?lasr</code>	<code>s, d, c, z</code>	Applies a sequence of plane rotations to a general rectangular matrix.
<code>?lasrt</code>	<code>s, d</code>	Sorts numbers in increasing or decreasing order.
<code>?lassq</code>	<code>s, d, c, z</code>	Updates a sum of squares represented in scaled form.
<code>?lasv2</code>	<code>s, d</code>	Computes the singular value decomposition of a 2-by-2 triangular matrix.
<code>?laswp</code>	<code>s, d, c, z</code>	Performs a series of row interchanges on a general rectangular matrix.
<code>?lasy2</code>	<code>s, d</code>	Solves the Sylvester matrix equation where the matrices are of order 1 or 2.

Routine Name	Data Types	Description
?lasyf	s, d, c, z	Computes a partial factorization of a real/complex symmetric matrix, using the diagonal pivoting method.
?lahef	c, z	Computes a partial factorization of a complex Hermitian indefinite matrix, using the diagonal pivoting method.
?latbs	s, d, c, z	Solves a triangular banded system of equations.
?latdf	s, d, c, z	Uses the LU factorization of the n-by-n matrix computed by ?getc2 and computes a contribution to the reciprocal Dif-estimate.
?latps	s, d, c, z	Solves a triangular system of equations with the matrix held in packed storage.
?latrd	s, d, c, z	Reduces the first <i>nb</i> rows and columns of a symmetric/Hermitian matrix <i>A</i> to real tridiagonal form by an orthogonal/unitary similarity transformation.
?latrs	s, d, c, z	Solves a triangular system of equations with the scale factor set to prevent overflow.
?latrz	s, d, c, z	Factors an upper trapezoidal matrix by means of orthogonal/unitary transformations.
?lauu2	s, d, c, z	Computes the product UU^H or L^HL , where <i>U</i> and <i>L</i> are upper or lower triangular matrices (unblocked algorithm).
?lauum	s, d, c, z	Computes the product UU^H or L^HL , where <i>U</i> and <i>L</i> are upper or lower triangular matrices (blocked algorithm).
?lazq3	s, d,	Checks for deflation, computes a shift and calls <i>dqds</i>
?lazq4	s, d,	Computes an approximation to the smallest eigenvalue using values of d from the previous transform.
?org2l/?ung2l	s, d/c, z	Generates all or part of the orthogonal/unitary matrix <i>Q</i> from a QL factorization determined by ?geqlf (unblocked algorithm).

Routine Name	Data Types	Description
?org2r/?ung2r	s, d/c, z	Generates all or part of the orthogonal/unitary matrix Q from a QR factorization determined by ?geqrf (unblocked algorithm).
?orgl2/?ungl2	s, d/c, z	Generates all or part of the orthogonal/unitary matrix Q from an LQ factorization determined by ?gelqf (unblocked algorithm).
?org2r/?ungr2	s, d/c, z	Generates all or part of the orthogonal/unitary matrix Q from an RQ factorization determined by ?gerqf (unblocked algorithm).
?orm2l/?unm2l	s, d/c, z	Multiplies a general matrix by the orthogonal/unitary matrix from a QL factorization determined by ?geqlf (unblocked algorithm).
?orm2r/?unm2r	s, d/c, z	Multiplies a general matrix by the orthogonal/unitary matrix from a QR factorization determined by ?geqrf (unblocked algorithm).
?orml2/?unml2	s, d/c, z	Multiplies a general matrix by the orthogonal/unitary matrix from a LQ factorization determined by ?gelqf (unblocked algorithm).
?ormr2/?unmr2	s, d/c, z	Multiplies a general matrix by the orthogonal/unitary matrix from a RQ factorization determined by ?gerqf (unblocked algorithm).
?ormr3/?unmr3	s, d/c, z	Multiplies a general matrix by the orthogonal/unitary matrix from a RZ factorization determined by ?tzzrf (unblocked algorithm).
?pbt2	s, d, c, z	Computes the Cholesky factorization of a symmetric/Hermitian positive definite band matrix (unblocked algorithm).
?pot2	s, d, c, z	Computes the Cholesky factorization of a symmetric/Hermitian positive definite matrix (unblocked algorithm).
?ptts2	s, d, c, z	Solves a tridiagonal system of the form $AX=B$ using the $L D L^H$ factorization computed by ?pttrf .

Routine Name	Data Types	Description
?rscl	s, d, cs, zd	Multiplies a vector by the reciprocal of a real scalar.
?sygs2/?hegs2	s, d/c, z	Reduces a symmetric/Hermitian definite generalized eigenproblem to standard form, using the factorization results obtained from ?potrf (unblocked algorithm).
?sytd2/?hetd2	s, d/c, z	Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation (unblocked algorithm).
?sytf2	s, d, c, z	Computes the factorization of a real/complex symmetric indefinite matrix, using the diagonal pivoting method (unblocked algorithm).
?hetf2	c, z	Computes the factorization of a complex Hermitian matrix, using the diagonal pivoting method (unblocked algorithm).
?tgex2	s, d, c, z	Swaps adjacent diagonal blocks in an upper (quasi) triangular matrix pair by an orthogonal/unitary equivalence transformation.
?tgsy2	s, d, c, z	Solves the generalized Sylvester equation (unblocked algorithm).
?trti2	s, d, c, z	Computes the inverse of a triangular matrix (unblocked algorithm).
clag2z	c $\rightarrow$ z	Converts a complex single precision matrix to a complex double precision matrix.
dlag2s	d $\rightarrow$ s	Converts a double precision matrix to a single precision matrix.
slag2d	s $\rightarrow$ d	Converts a single precision matrix to a double precision matrix.
zlag2c	z $\rightarrow$ c	Converts a complex double precision matrix to a complex single precision matrix.

?lacgv

Conjugates a complex vector.

Syntax

```
call clacgv( n, x, incx )
```

```
call zlacgv( n, x, incx )
```

Description

This routine conjugates a complex vector x of length n and increment $incx$ (see “[Vector Arguments in BLAS](#)” in Appendix B).

Input Parameters

n	INTEGER. The length of the vector x ($n \geq 0$).
x	COMPLEX for clacgv COMPLEX*16 for zlacgv. Array, dimension $(1+(n-1) * incx)$. Contains the vector of length n to be conjugated.
$incx$	INTEGER. The spacing between successive elements of x .

Output Parameters

x	On exit, overwritten with <code>conjg(x)</code> .

?lacrm

Multiplies a complex matrix by a square real matrix.

Syntax

```
call clacrm( m, n, a, lda, b, ldb, c, ldc, rwork )
```

```
call zlacrm( m, n, a, lda, b, ldb, c, ldc, rwork )
```

Description

This routine performs a simple matrix-matrix multiplication of the form

$$C = A*B,$$

where A is m -by- n and complex, B is n -by- n and real, C is m -by- n and complex.

Input Parameters

m	INTEGER. The number of rows of the matrix A and of the matrix C ($m \geq 0$).
n	INTEGER. The number of columns and rows of the matrix B and the number of columns of the matrix C ($n \geq 0$).
a	COMPLEX for <code>clacrm</code> COMPLEX*16 for <code>zlacrm</code> Array, DIMENSION (lda, n). Contains the m -by- n matrix A .
lda	INTEGER. The leading dimension of the array a , $lda \geq \max(1, m)$.
b	REAL for <code>clacrm</code> DOUBLE PRECISION for <code>zlacrm</code> Array, DIMENSION (ldb, n). Contains the n -by- n matrix B .
ldb	INTEGER. The leading dimension of the array b , $ldb \geq \max(1, n)$.
ldc	INTEGER. The leading dimension of the output array c , $ldc \geq \max(1, n)$.
$rwork$	REAL for <code>clacrm</code> DOUBLE PRECISION for <code>zlacrm</code> Workspace array, DIMENSION ($2*m*n$).

Output Parameters

c	COMPLEX for <code>clacrm</code> COMPLEX*16 for <code>zlacrm</code> Array, DIMENSION (ldc, n). Contains the m -by- n matrix C .

zlacrt

Performs a linear transformation of a pair of complex vectors.

Syntax

```
call clacrt( n, cx, incx, cy, incy, c, s )
```

```
call zlacrt( n, cx, incx, cy, incy, c, s )
```

Description

This routine performs the following transformation

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \Rightarrow \begin{pmatrix} x \\ y \end{pmatrix},$$

where c, s are complex scalars and x, y are complex vectors.

Input Parameters

n	INTEGER. The number of elements in the vectors cx and cy ($n \geq 0$).
cx, cy	COMPLEX for clacrt COMPLEX*16 for zlacrt Arrays, dimension (n). Contain input vectors x and y , respectively.
$incx$	INTEGER. The increment between successive elements of cx .
$incy$	INTEGER. The increment between successive elements of cy .
c, s	COMPLEX for clacrt COMPLEX*16 for zlacrt Complex scalars that define the transform matrix

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix}$$

Output Parameters

<i>cx</i>	On exit, overwritten with $c*x + s*y$.
<i>cy</i>	On exit, overwritten with $-s*x + c*y$.

?laesy

Computes the eigenvalues and eigenvectors of a 2-by-2 complex symmetric matrix, and checks that the norm of the matrix of eigenvectors is larger than a threshold value.

Syntax

```
call claesy( a, b, c, rt1, rt2, evscal, cs1, sn1 )
call zlaesy( a, b, c, rt1, rt2, evscal, cs1, sn1 )
```

Description

This routine performs the eigendecomposition of a 2-by-2 symmetric matrix

$$\begin{bmatrix} a & b \\ b & c \end{bmatrix},$$

provided the norm of the matrix of eigenvectors is larger than some threshold value.

rt1 is the eigenvalue of larger absolute value, and *rt2* of smaller absolute value. If the eigenvectors are computed, then on return (*cs1*, *sn1*) is the unit eigenvector for *rt1*, hence

$$\begin{bmatrix} cs1 & sn1 \\ -sn1 & cs1 \end{bmatrix} \cdot \begin{bmatrix} a & b \\ b & c \end{bmatrix} \cdot \begin{bmatrix} cs1 & -sn1 \\ sn1 & cs1 \end{bmatrix} = \begin{bmatrix} rt1 & 0 \\ 0 & rt2 \end{bmatrix}$$

Input Parameters

a, b, c COMPLEX for `claesy`
 COMPLEX*16 for `zlaesy`
 Elements of the input matrix.

Output Parameters

rt1, rt2 COMPLEX for `claesy`
 COMPLEX*16 for `zlaesy`
 Eigenvalues of larger and smaller modulus, respectively.

evscal COMPLEX for `claesy`
 COMPLEX*16 for `zlaesy`
 The complex value by which the eigenvector matrix was scaled to make it orthonormal. If *evscal* is zero, the eigenvectors were not computed. This means one of two things: the 2-by-2 matrix could not be diagonalized, or the norm of the matrix of eigenvectors before scaling was larger than the threshold value `thresh` (set to 0.1E0).

cs1, sn1 COMPLEX for `claesy`
 COMPLEX*16 for `zlaesy`
 If *evscal* is not zero, then (*cs1, sn1*) is the unit right eigenvector for *rt1*.

?rot

Applies a plane rotation with real cosine and complex sine to a pair of complex vectors.

Syntax

```
call crot( n, cx, incx, cy, incy, c, s )
call zrot( n, cx, incx, cy, incy, c, s )
```


Description

This routine applies a plane rotation, where the cosine (c) is real and the sine (s) is complex, and the vectors cx and cy are complex. This routine has its real equivalents in BLAS (see [?rot](#) in Chapter 2).

Input Parameters

n	INTEGER. The number of elements in the vectors cx and cy .
cx, cy	COMPLEX for <code>crot</code> COMPLEX*16 for <code>zrot</code> Arrays of dimension (n), contain input vectors x and y , respectively.
$incx$	INTEGER. The increment between successive elements of cx .
$incy$	INTEGER. The increment between successive elements of cy .
c	REAL for <code>crot</code> DOUBLE PRECISION for <code>zrot</code>
s	COMPLEX for <code>crot</code> COMPLEX*16 for <code>zrot</code> Values that define a rotation

$$\begin{bmatrix} c & s \\ -\text{conjg}(s) & c \end{bmatrix}$$

where $c*c + s*\text{conjg}(s) = 1.0$.

Output Parameters

cx	On exit, overwritten with $c*x + s*y$.
cy	On exit, overwritten with $-\text{conjg}(s)*x + c*y$.

?spmv

Computes a matrix-vector product for complex vectors using a complex symmetric packed matrix.

Syntax

```
call cspmv( uplo, n, alpha, ap, x, incx, beta, y, incy )
call zspmv( uplo, n, alpha, ap, x, incx, beta, y, incy )
```

Description

These routines perform a matrix-vector operation defined as

$$y := \alpha * a * x + \beta * y,$$

where:

alpha and *beta* are complex scalars,

x and *y* are *n*-element complex vectors

a is an *n*-by-*n* complex symmetric matrix, supplied in packed form.

These routines have their real equivalents in BLAS (see [?spmv](#) in Chapter 2).

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix <i>a</i> is supplied in the packed array <i>ap</i>. If <i>uplo</i> = 'U' or 'u', the upper triangular part of the matrix <i>a</i> is supplied in the array <i>ap</i>. If <i>uplo</i> = 'L' or 'l', the lower triangular part of the matrix <i>a</i> is supplied in the array <i>ap</i>.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>a</i>. The value of <i>n</i> must be at least zero.
<i>alpha, beta</i>	COMPLEX for cspmv COMPLEX*16 for zspmv Specify complex scalars <i>alpha</i> and <i>beta</i>. When <i>beta</i> is supplied as zero, then <i>y</i> need not be set on input.
<i>ap</i>	COMPLEX for cspmv

	COMPLEX*16 for <code>zspmv</code> Array, DIMENSION at least $((n * (n + 1)) / 2)$. Before entry, with <code>uplo = 'U'</code> or <code>'u'</code>, the array <code>ap</code> must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that <code>ap(1)</code> contains $A(1, 1)$, <code>ap(2)</code> and <code>ap(3)</code> contain $A(1, 2)$ and $A(2, 2)$ respectively, and so on. Before entry, with <code>uplo = 'L'</code> or <code>'l'</code>, the array <code>ap</code> must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that <code>ap(1)</code> contains $a(1, 1)$, <code>ap(2)</code> and <code>ap(3)</code> contain $a(2, 1)$ and $a(3, 1)$ respectively, and so on.
<code>x</code>	COMPLEX for <code>cspmv</code> COMPLEX*16 for <code>zspmv</code> Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <code>x</code> must contain the n-element vector <code>x</code>.
<code>incx</code>	INTEGER. Specifies the increment for the elements of <code>x</code>. The value of <code>incx</code> must not be zero.
<code>y</code>	COMPLEX for <code>cspmv</code> COMPLEX*16 for <code>zspmv</code> Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array <code>y</code> must contain the n-element vector <code>y</code>.
<code>incy</code>	INTEGER. Specifies the increment for the elements of <code>y</code>. The value of <code>incy</code> must not be zero.

Output Parameters

<code>y</code>	Overwritten by the updated vector <code>y</code> .

?spr

Performs the symmetrical rank-1 update of a complex symmetric packed matrix.

Syntax

```
call cspr( uplo, n, alpha, x, incx, ap )
```

```
call zspr( uplo, n, alpha, x, incx, ap )
```

Description

The ?spr routines perform a matrix-vector operation defined as

$$a := \alpha * x * \text{conjg}(x') + a,$$

where:

alpha is a complex scalar

x is an *n*-element complex vector

a is an *n*-by-*n* complex symmetric matrix, supplied in packed form.

These routines have their real equivalents in BLAS (see ?spr in Chapter 2).

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix <i>a</i> is supplied in the packed array <i>ap</i>, as follows: If <i>uplo</i> = 'U' or 'u', the upper triangular part of the matrix <i>a</i> is supplied in the array <i>ap</i>. If <i>uplo</i> = 'L' or 'l', the lower triangular part of the matrix <i>a</i> is supplied in the array <i>ap</i>.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>a</i>. The value of <i>n</i> must be at least zero.
<i>alpha</i>	COMPLEX for cspr COMPLEX*16 for zspr Specifies the scalar <i>alpha</i>.
<i>x</i>	COMPLEX for cspr COMPLEX*16 for zspr

Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array x must contain the n -element vector x .

incx INTEGER. Specifies the increment for the elements of x . The value of *incx* must not be zero.

ap COMPLEX for *cspr*
COMPLEX*16 for *zspr*
Array, DIMENSION at least $((n * (n + 1)) / 2)$. Before entry, with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains $A(1, 1)$, *ap*(2) and *ap*(3) contain $A(1, 2)$ and $A(2, 2)$ respectively, and so on.
Before entry, with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains $a(1, 1)$, *ap*(2) and *ap*(3) contain $a(2, 1)$ and $a(3, 1)$ respectively, and so on.
Note that the imaginary parts of the diagonal elements need not be set, they are assumed to be zero, and on exit they are set to zero.

Output Parameters

ap With *uplo* = 'U' or 'u', overwritten by the upper triangular part of the updated matrix.
With *uplo* = 'L' or 'l', overwritten by the lower triangular part of the updated matrix.

?symv

Computes a matrix-vector product for a complex symmetric matrix.

Syntax

```
call csymv( uplo, n, alpha, a, lda, x, incx, beta, y, incy )
call zsymv( uplo, n, alpha, a, lda, x, incx, beta, y, incy )
```

Description

These routines perform the matrix-vector operation defined as

$$y := \alpha * a * x + \beta * y,$$

where:

α and β are complex scalars

x and y are n -element complex vectors

a is an n -by- n symmetric complex matrix.

These routines have their real equivalents in BLAS (see [?symv](#) in Chapter 2).

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>a</i> is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the array <i>a</i> is used. If <i>uplo</i> = 'L' or 'l', then the lower triangular part of the array <i>a</i> is used.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>a</i>. The value of <i>n</i> must be at least zero.
<i>alpha, beta</i>	COMPLEX for <i>csymv</i> COMPLEX*16 for <i>zsymv</i> Specify the scalars <i>alpha</i> and <i>beta</i>. When <i>beta</i> is supplied as zero, then <i>y</i> need not be set on input.
<i>a</i>	COMPLEX for <i>csymv</i> COMPLEX*16 for <i>zsymv</i> Array, DIMENSION (<i>lda</i>, <i>n</i>). Before entry with <i>uplo</i> = 'U' or 'u', the leading n-by-n upper triangular part of the array <i>a</i> must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of <i>a</i> is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', the leading n-by-n lower triangular part of the array <i>a</i> must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of <i>a</i> is not referenced.

<i>lda</i>	INTEGER. Specifies the first dimension of <i>A</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, n)$.
<i>x</i>	COMPLEX for <i>csymv</i> COMPLEX*16 for <i>zsymv</i> Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>y</i>	COMPLEX for <i>csymv</i> COMPLEX*16 for <i>zsymv</i> Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array <i>y</i> must contain the <i>n</i> -element vector <i>y</i> .
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.

Output Parameters

<i>y</i>	Overwritten by the updated vector <i>y</i> .

?syr

Performs the symmetric rank-1 update of a complex symmetric matrix.

Syntax

```
call csyr( uplo, n, alpha, x, incx, a, lda )
call zsyr( uplo, n, alpha, x, incx, a, lda )
```

Description

These routines perform the symmetric rank 1 operation defined as

$$a := \alpha x x^* + a,$$

where:

alpha is a complex scalar

x is an n -element complex vector

a is an n -by- n complex symmetric matrix.

These routines have their real equivalents in BLAS (see [?syr](#) in Chapter 2).

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array a is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the array a is used. If <i>uplo</i> = 'L' or 'l', then the lower triangular part of the array a is used.
<i>n</i>	INTEGER. Specifies the order of the matrix a. The value of n must be at least zero.
<i>alpha</i>	COMPLEX for csyr COMPLEX*16 for zsyrr Specifies the scalar <i>alpha</i>.
<i>x</i>	COMPLEX for csyr COMPLEX*16 for zsyrr Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array x must contain the n-element vector x.
<i>incx</i>	INTEGER. Specifies the increment for the elements of x. The value of <i>incx</i> must not be zero.
<i>a</i>	COMPLEX for csyr COMPLEX*16 for zsyrr Array, DIMENSION (<i>lda</i>, n). Before entry with <i>uplo</i> = 'U' or 'u', the leading n-by-n upper triangular part of the array a must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of a is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', the leading n-by-n lower triangular part of the array a must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of a is not referenced.

lda INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least $\max(1, n)$.

Output Parameters

a With *uplo* = 'U' or 'u', the upper triangular part of the array *a* is overwritten by the upper triangular part of the updated matrix.
With *uplo* = 'L' or 'l', the lower triangular part of the array *a* is overwritten by the lower triangular part of the updated matrix.

i?max1

Finds the index of the vector element whose real part has maximum absolute value.

Syntax

```
index = icmax1( n, cx, incx )
```

```
index = izmax1( n, cx, incx )
```

Description

Given a complex vector *cx*, the i?max1 functions return the index of the vector element whose real part has maximum absolute value. These functions are based on the BLAS functions icamax/izamax, but using the absolute value of the real part. They are designed for use with clacon/zlacon.

Input Parameters

n INTEGER. Specifies the number of elements in the vector *cx*.

cx COMPLEX for icmax1
COMPLEX*16 for izmax1
Array, DIMENSION at least $(1 + (n-1) * \text{abs}(incx))$.
Contains the input vector.

incx INTEGER. Specifies the spacing between successive elements of *cx*.

Output Parameters

<i>index</i>	INTEGER. Contains the index of the vector element whose real part has maximum absolute value.

?sum1

Forms the 1-norm of the complex vector using the true absolute value.

Syntax

```
res = scsum1( n, cx, incx )
res = dzsum1( n, cx, incx )
```

Description

Given a complex vector *cx*, *scsum1*/*dzsum1* functions take the sum of the absolute values of vector elements and return a single/DOUBLE PRECISION result, respectively. These functions are based on *scasum*/*dzasum* from Level 1 BLAS, but use the true absolute value and were designed for use with *clacon*/*zlacon*.

Input Parameters

<i>n</i>	INTEGER. Specifies the number of elements in the vector <i>cx</i> .
<i>cx</i>	COMPLEX for <i>scsum1</i> COMPLEX*16 for <i>dzsum1</i> Array, DIMENSION at least $(1+(n-1)*abs(incx))$. Contains the input vector whose elements will be summed.
<i>incx</i>	INTEGER. Specifies the spacing between successive elements of <i>cx</i> (<i>incx</i> > 0).

Output Parameters

<i>res</i>	REAL for <i>scsum1</i> DOUBLE PRECISION for <i>dzsum1</i> Contains the sum of absolute values.

?gbtf2

Computes the LU factorization of a general band matrix using the unblocked version of the algorithm.

Syntax

```
call sgbtf2( m, n, kl, ku, ab, ldab, ipiv, info )
call dgbtf2( m, n, kl, ku, ab, ldab, ipiv, info )
call cgbtf2( m, n, kl, ku, ab, ldab, ipiv, info )
call zgbtf2( m, n, kl, ku, ab, ldab, ipiv, info )
```

Description

The routine forms the *LU* factorization of a general real/complex *m*-by-*n* band matrix *A* with *kl* sub-diagonals and *ku* super-diagonals. The routine uses partial pivoting with row interchanges and implements the unblocked version of the algorithm, calling Level 2 BLAS. See also [?gbtrf](#).

Input Parameters

<i>m</i>	INTEGER. The number of rows of the matrix <i>A</i> ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in <i>A</i> ($n \geq 0$).
<i>kl</i>	INTEGER. The number of sub-diagonals within the band of <i>A</i> ($kl \geq 0$).
<i>ku</i>	INTEGER. The number of super-diagonals within the band of <i>A</i> ($ku \geq 0$).
<i>ab</i>	REAL for sgbtf2 DOUBLE PRECISION for dgbtf2 COMPLEX for cgbtf2 COMPLEX*16 for zgbtf2. Array, DIMENSION (<i>ldab</i> ,*). The array <i>ab</i> contains the matrix <i>A</i> in band storage (see Matrix Arguments). The second dimension of <i>ab</i> must be at least $\max(1, n)$.
<i>ldab</i>	INTEGER. The first dimension of the array <i>ab</i> .

$$(ldab \geq 2kl + ku + 1)$$

Output Parameters

<i>ab</i>	Overwritten by details of the factorization. The diagonal and $kl + ku$ super-diagonals of U are stored in the first $1 + kl + ku$ rows of <i>ab</i> . The multipliers used during the factorization are stored in the next kl rows.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, \min(m, n))$. The pivot indices: row i was interchanged with row $ipiv(i)$.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value. If $info = i$, u_{ii} is 0. The factorization has been completed, but U is exactly singular. Division by 0 will occur if you use the factor U for solving a system of linear equations.

?gebd2

Reduces a general matrix to bidiagonal form using an unblocked algorithm.

Syntax

```
call sgebd2( m, n, a, lda, d, e, tauq, taup, work, info )
call dgebd2( m, n, a, lda, d, e, tauq, taup, work, info )
call cgebd2( m, n, a, lda, d, e, tauq, taup, work, info )
call zgebd2( m, n, a, lda, d, e, tauq, taup, work, info )
```

Description

The routine reduces a general m -by- n matrix A to upper or lower bidiagonal form B by an orthogonal (unitary) transformation: $Q'^*A^*P = B$

If $m \geq n$, B is upper bidiagonal; if $m < n$, B is lower bidiagonal.

The routine does not form the matrices Q and P explicitly, but represents them as products of elementary reflectors. if $m \geq n$,

$Q = H(1) * H(2) * \dots * H(n)$, and $P = G(1) * G(2) * \dots * G(n-1)$

if $m < n$,

$Q = H(1) * H(2) * \dots * H(m-1)$, and $P = G(1) * G(2) * \dots * G(m)$

Each $H(i)$ and $G(i)$ has the form

$H(i) = I - \text{tauq} * v * v'$ and $G(i) = I - \text{taup} * u * u'$

where tauq and taup are scalars (real for sgebd2/dgebd2, complex for cgebd2/zgebd2), and v and u are vectors (real for sgebd2/dgebd2, complex for cgebd2/zgebd2).

Input Parameters

m INTEGER. The number of rows in the matrix A ($m \geq 0$).

n INTEGER. The number of columns in A ($n \geq 0$).

a, work REAL for sgebd2
DOUBLE PRECISION for dgebd2
COMPLEX for cgebd2
COMPLEX*16 for zgebd2.
Arrays:
 $a(\text{lda}, *)$ contains the m -by- n general matrix A to be reduced. The second dimension of a must be at least $\max(1, n)$.
 $\text{work}(*)$ is a workspace array, the dimension of work must be at least $\max(1, m, n)$.

lda INTEGER. The first dimension of a ; at least $\max(1, m)$.

Output Parameters

a if $m \geq n$, the diagonal and first super-diagonal of a are overwritten with the upper bidiagonal matrix B . Elements below the diagonal, with the array tauq , represent the orthogonal/unitary matrix Q as a product of elementary reflectors, and elements above the first superdiagonal, with the array taup , represent the orthogonal/unitary matrix P as a product of elementary reflectors.
if $m < n$, the diagonal and first sub-diagonal of a are overwritten by the lower bidiagonal matrix B . Elements below the first subdiagonal, with the array tauq , represent

the orthogonal/unitary matrix Q as a product of elementary reflectors, and elements above the diagonal, with the array τ_{aup} , represent the orthogonal/unitary matrix P as a product of elementary reflectors.

d	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Array, DIMENSION at least $\max(1, \min(m, n))$. Contains the diagonal elements of the bidiagonal matrix B: $d(i) = a(i, i)$.
e	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Array, DIMENSION at least $\max(1, \min(m, n) - 1)$. Contains the off-diagonal elements of the bidiagonal matrix B: if $m \geq n$, $e(i) = a(i, i+1)$ for $i = 1, 2, \dots, n-1$; if $m < n$, $e(i) = a(i+1, i)$ for $i = 1, 2, \dots, m-1$.
τ_{auq}, τ_{aup}	REAL for sgebd2 DOUBLE PRECISION for dgebd2 COMPLEX for cgebd2 COMPLEX*16 for zgebd2. Arrays, DIMENSION at least $\max(1, \min(m, n))$. Contain scalar factors of the elementary reflectors which represent orthogonal/unitary matrices Q and P, respectively.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the ith parameter had an illegal value.

?gehd2

*Reduces a general square matrix to upper
Hessenberg form using an unblocked algorithm.*

Syntax

```
call sgehd2( n, ilo, ihi, a, lda, tau, work, info )
call dgehd2( n, ilo, ihi, a, lda, tau, work, info )
call cgehd2( n, ilo, ihi, a, lda, tau, work, info )
call zgehd2( n, ilo, ihi, a, lda, tau, work, info )
```

Description

The routine reduces a real/complex general matrix A to upper Hessenberg form H by an orthogonal or unitary similarity transformation $Q^* A Q = H$.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of *elementary reflectors*.

Input Parameters

n	INTEGER The order of the matrix A ($n \geq 0$).
ilo, ihi	INTEGER. It is assumed that A is already upper triangular in rows and columns $1:ilo-1$ and $ihi+1:n$. If A has been output by ?gebal, then ilo and ihi must contain the values returned by that routine. Otherwise they should be set to $ilo = 1$ and $ihi = n$. Constraint: $1 \leq ilo \leq ihi \leq \max(1, n)$.
$a, work$	REAL for sgehd2 DOUBLE PRECISION for dgehd2 COMPLEX for cgehd2 COMPLEX*16 for zgehd2. Arrays: $a(lda, *)$ contains the n -by- n matrix A to be reduced. The second dimension of a must be at least $\max(1, n)$. $work(n)$ is a workspace array.
lda	INTEGER. The first dimension of a ; at least $\max(1, n)$.

Output Parameters

<i>a</i>	On exit, the upper triangle and the first subdiagonal of <i>A</i> are overwritten with the upper Hessenberg matrix <i>H</i> and the elements below the first subdiagonal, with the array <i>tau</i> , represent the orthogonal/unitary matrix <i>Q</i> as a product of elementary reflectors. See <i>Application Notes</i> below.
<i>tau</i>	REAL for sgehd2 DOUBLE PRECISION for dgehd2 COMPLEX for cgehd2 COMPLEX*16 for zgehd2. Array, DIMENSION at least $\max(1, n-1)$. Contains the scalar factors of elementary reflectors. See <i>Application Notes</i> below.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Application Notes

The matrix *Q* is represented as a product of (*ihi* - *ilo*) elementary reflectors

$$Q = H(ilo) * H(ilo + 1) * \dots * H(ihi - 1)$$

Each *H*(*i*) has the form

$$H(i) = I - tau * v * v'$$

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(1:*i*) = 0, *v*(*i*+1) = 1 and *v*(*ihi*+1:*n*) = 0.

On exit, *v*(*i*+2:*ihi*) is stored in *a*(*i*+2:*ihi*, *i*) and *tau* in *tau*(*i*).

The contents of *a* are illustrated by the following example, with *n* = 7, *ilo* = 2 and *ihi* = 6:

$$\begin{array}{cc}
 \text{on entry} & \text{on exit} \\
 \left[\begin{array}{cccccc} a & a & a & a & a & a \\ & a & a & a & a & a \\ & & a & a & a & a \\ & & & a & a & a \\ & & & & a & a \\ & & & & & a \end{array} \right] & \left[\begin{array}{cccccc} a & a & h & h & h & h \\ & a & h & h & h & h \\ & & h & h & h & h \\ & & & v_2 & h & h \\ & & & v_2 & v_3 & h \\ & & & v_2 & v_3 & v_4 \end{array} \right] \\
 & \left[\begin{array}{cccccc} & & & & & a \end{array} \right]
 \end{array}$$

where a denotes an element of the original matrix A , h denotes a modified element of the upper Hessenberg matrix H , and v_i denotes an element of the vector defining $H(i)$.

?gelq2

Computes the LQ factorization of a general rectangular matrix using an unblocked algorithm.

Syntax

```
call sgelq2( m, n, a, lda, tau, work, info )
call dgelq2( m, n, a, lda, tau, work, info )
call cgelq2( m, n, a, lda, tau, work, info )
call zgelq2( m, n, a, lda, tau, work, info )
```

Description

The routine computes an LQ factorization of a real/complex m -by- n matrix A as $A = L^*Q$.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ elementary reflectors :

$Q = H(k) \dots H(2) H(1)$ (or $Q = H(k)' \dots H(2)' H(1)'$ for complex flavors), where $k = \min(m, n)$

Each $H(i)$ has the form

$H(i) = I - \tau v v'$

where τ is a real/complex scalar stored in $\tau(i)$, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$.

On exit, $v(i+1:n)$ is stored in $a(i, i+1:n)$.

Input Parameters

m	INTEGER. The number of rows in the matrix A ($m \geq 0$).
n	INTEGER. The number of columns in A ($n \geq 0$).
$a, work$	REAL for <code>sgelq2</code> DOUBLE PRECISION for <code>dgelq2</code> COMPLEX for <code>cgelq2</code> COMPLEX*16 for <code>zgelq2</code> . Arrays: $a(lda,*)$ contains the m -by- n matrix A . The second dimension of a must be at least $\max(1, n)$. $work(m)$ is a workspace array.
lda	INTEGER. The first dimension of a ; at least $\max(1, m)$.

Output Parameters

a	Overwritten by the factorization data as follows: on exit, the elements on and below the diagonal of the array a contain the m -by- $\min(n, m)$ lower trapezoidal matrix L (L is lower triangular if $n \geq m$); the elements above the diagonal, with the array τ , represent the orthogonal/unitary matrix Q as a product of $\min(n, m)$ elementary reflectors.
τ	REAL for <code>sgelq2</code> DOUBLE PRECISION for <code>dgelq2</code> COMPLEX for <code>cgelq2</code> COMPLEX*16 for <code>zgelq2</code> . Array, DIMENSION at least $\max(1, \min(m, n))$. Contains scalar factors of the elementary reflectors.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

?geql2

Computes the QL factorization of a general rectangular matrix using an unblocked algorithm.

Syntax

```
call sgeql2( m, n, a, lda, tau, work, info )
call dgeql2( m, n, a, lda, tau, work, info )
call cgeql2( m, n, a, lda, tau, work, info )
call zgeql2( m, n, a, lda, tau, work, info )
```

Description

The routine computes a QL factorization of a real/complex m -by- n matrix A as $A = Q^*L$.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ *elementary reflectors* :

$Q = H(k)^* \dots ^*H(2)^*H(1)$, where $k = \min(m, n)$

Each $H(i)$ has the form

$H(i) = I - \tau v v^*$

where τ is a real/complex scalar stored in $\tau(i)$, and v is a real/complex vector with $v(m-k+i+1:m) = 0$ and $v(m-k+i) = 1$.

On exit, $v(1:m-k+i-1)$ is stored in $a(1:m-k+i-1, n-k+i)$.

Input Parameters

m INTEGER. The number of rows in the matrix A ($m \geq 0$).

n INTEGER. The number of columns in A ($n \geq 0$).

$a, work$ REAL for sgeql2
DOUBLE PRECISION for dgeql2
COMPLEX for cgeql2
COMPLEX*16 for zgeql2.

Arrays:

$a(lda, *)$ contains the m -by- n matrix A .

The second dimension of a must be at least $\max(1, n)$.

work(*m*) is a workspace array.

lda INTEGER. The first dimension of *a*; at least $\max(1, m)$.

Output Parameters

a Overwritten by the factorization data as follows:
on exit, if $m \geq n$, the lower triangle of the subarray
 $a(m-n+1:m, 1:n)$ contains the n -by- n lower triangular
matrix L ; if $m < n$, the elements on and below the $(n-m)$ th
superdiagonal contain the m -by- n lower trapezoidal matrix
 L ; the remaining elements, with the array *tau*, represent
the orthogonal/unitary matrix Q as a product of elementary
reflectors.

tau REAL for sgeql2
DOUBLE PRECISION for dgeql2
COMPLEX for cgeql2
COMPLEX*16 for zgeql2.
Array, DIMENSION at least $\max(1, \min(m, n))$.
Contains scalar factors of the elementary reflectors.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

?geqr2

Computes the QR factorization of a general rectangular matrix using an unblocked algorithm.

Syntax

```
call sgeqr2( m, n, a, lda, tau, work, info )
call dgeqr2( m, n, a, lda, tau, work, info )
call cgeqr2( m, n, a, lda, tau, work, info )
call zgeqr2( m, n, a, lda, tau, work, info )
```

Description

The routine computes a QR factorization of a real/complex m -by- n matrix A as $A = Q * R$.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ *elementary reflectors* :

$$Q = H(1) * H(2) * \dots * H(k), \text{ where } k = \min(m, n)$$

Each $H(i)$ has the form

$$H(i) = I - \tau v v'$$

where τ is a real/complex scalar stored in $\tau(i)$, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$.

On exit, $v(i+1:m)$ is stored in $a(i+1:m, i)$.

Input Parameters

m INTEGER. The number of rows in the matrix A ($m \geq 0$).

n INTEGER. The number of columns in A ($n \geq 0$).

$a, work$ REAL for sgeqr2
DOUBLE PRECISION for dgeqr2
COMPLEX for cgeqr2
COMPLEX*16 for zgeqr2.
Arrays:
 $a(lda, *)$ contains the m -by- n matrix A .
The second dimension of a must be at least $\max(1, n)$.
 $work(n)$ is a workspace array.

lda INTEGER. The first dimension of a ; at least $\max(1, m)$.

Output Parameters

a Overwritten by the factorization data as follows:
on exit, the elements on and above the diagonal of the array
 a contain the $\min(n, m)$ -by- n upper trapezoidal matrix R (R
is upper triangular if $m \geq n$); the elements below the
diagonal, with the array τ , represent the
orthogonal/unitary matrix Q as a product of elementary
reflectors.

τ REAL for sgeqr2
DOUBLE PRECISION for dgeqr2
COMPLEX for cgeqr2

COMPLEX*16 for zgeqr2.
 Array, DIMENSION at least $\max(1, \min(m, n))$.
 Contains scalar factors of the elementary reflectors.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

?gerq2

Computes the RQ factorization of a general rectangular matrix using an unblocked algorithm.

Syntax

```
call sgerq2( m, n, a, lda, tau, work, info )
call dgerq2( m, n, a, lda, tau, work, info )
call cgerq2( m, n, a, lda, tau, work, info )
call zgerq2( m, n, a, lda, tau, work, info )
```

Description

The routine computes a RQ factorization of a real/complex m -by- n matrix A as $A = R^*Q$.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ *elementary reflectors* :

$Q = H(1) * H(2) * \dots * H(k)$, where $k = \min(m, n)$

Each $H(i)$ has the form

$H(i) = I - \tau v v'$

where τ is a real/complex scalar stored in $\tau(i)$, and v is a real/complex vector with $v(n-k+i+1:n) = 0$ and $v(n-k+i) = 1$.

On exit, $v(1:n-k+i-1)$ is stored in $a(m-k+i, 1:n-k+i-1)$.

Input Parameters

m INTEGER. The number of rows in the matrix A ($m \geq 0$).

n INTEGER. The number of columns in A ($n \geq 0$).

a, work REAL for sgerq2
 DOUBLE PRECISION for dgerq2
 COMPLEX for cgerq2
 COMPLEX*16 for zgerq2.
Arrays:
a(lda,)* contains the m -by- n matrix A .
 The second dimension of a must be at least $\max(1, n)$.
work(m) is a workspace array.

lda INTEGER. The first dimension of a ; at least $\max(1, m)$.

Output Parameters

a Overwritten by the factorization data as follows:
 on exit, if $m \leq n$, the upper triangle of the subarray $a(1:m, n-m+1:n)$ contains the m -by- m upper triangular matrix R ;
 if $m > n$, the elements on and above the $(m-n)$ -th
 subdiagonal contain the m -by- n upper trapezoidal matrix R ;
 the remaining elements, with the array *tau*, represent the
 orthogonal/unitary matrix Q as a product of elementary
 reflectors.

tau REAL for sgerq2
 DOUBLE PRECISION for dgerq2
 COMPLEX for cgerq2
 COMPLEX*16 for zgerq2.
Array, DIMENSION at least $\max(1, \min(m, n))$.
 Contains scalar factors of the elementary reflectors.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = $-i$, the i -th parameter had an illegal value.

?gesc2

Solves a system of linear equations using the LU factorization with complete pivoting computed by ?getc2.

Syntax

```
call sgesc2( n, a, lda, rhs, ipiv, jpiv, scale )
call dgesec2( n, a, lda, rhs, ipiv, jpiv, scale )
call cgesec2( n, a, lda, rhs, ipiv, jpiv, scale )
call zgesec2( n, a, lda, rhs, ipiv, jpiv, scale )
```

Description

This routine solves a system of linear equations

$$A * X = scale * RHS$$

with a general n -by- n matrix A using the LU factorization with complete pivoting computed by [?getc2](#).

Input Parameters

n	INTEGER. The order of the matrix A .
a, rhs	REAL for <code>sgesc2</code> DOUBLE PRECISION for <code>dgesec2</code> COMPLEX for <code>cgesec2</code> COMPLEX*16 for <code>zgesec2</code> . Arrays: $a(lda, *)$ contains the LU part of the factorization of the n -by- n matrix A computed by <code>?getc2</code> : $A = P * L * U * Q$. The second dimension of a must be at least $\max(1, n)$; $rhs(n)$ contains on entry the right hand side vector for the system of equations.
lda	INTEGER. The first dimension of a ; at least $\max(1, n)$.
$ipiv$	INTEGER. Array, DIMENSION at least $\max(1, n)$.

jpiv The pivot indices: for $1 \leq i \leq n$, row i of the matrix has been interchanged with row $ipiv(i)$.
 INTEGER.
 Array, DIMENSION at least $\max(1, n)$.
 The pivot indices: for $1 \leq j \leq n$, column j of the matrix has been interchanged with column $jpiv(j)$.

Output Parameters

rhs On exit, overwritten with the solution vector x .
scale REAL for sgesc2/cgesc2
 DOUBLE PRECISION for dgesc2/zgesc2
 Contains the scale factor. *scale* is chosen in the range $0 \leq scale \leq 1$ to prevent overflow in the solution.

?getc2

Computes the LU factorization with complete pivoting of the general n -by- n matrix.

Syntax

```
call sgetc2( n, a, lda, ipiv, jpiv, info )
call dgetc2( n, a, lda, ipiv, jpiv, info )
call cgetc2( n, a, lda, ipiv, jpiv, info )
call zgetc2( n, a, lda, ipiv, jpiv, info )
```

Description

This routine computes an LU factorization with complete pivoting of the n -by- n matrix A . The factorization has the form $A = P^*L^*U^*Q$, where P and Q are permutation matrices, L is lower triangular with unit diagonal elements and U is upper triangular.

The LU factorization computed by this routine is used by [?latdf](#) to compute a contribution to the reciprocal Dif-estimate.

Input Parameters

<i>n</i>	INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).
<i>a</i>	REAL for sgetc2 DOUBLE PRECISION for dgetc2 COMPLEX for cgetc2 COMPLEX*16 for zgetc2. Array <i>a</i> (<i>lda</i> ,*) contains the <i>n</i> -by- <i>n</i> matrix <i>A</i> to be factored. The second dimension of <i>a</i> must be at least $\max(1, n)$;
<i>lda</i>	INTEGER. The first dimension of <i>a</i> ; at least $\max(1, n)$.

Output Parameters

<i>a</i>	On exit, the factors <i>L</i> and <i>U</i> from the factorization $A = P^*L^*U^*Q$; the unit diagonal elements of <i>L</i> are not stored. If <i>U</i> (<i>k</i> , <i>k</i>) appears to be less than <i>smin</i> , <i>U</i> (<i>k</i> , <i>k</i>) is given the value of <i>smin</i> , that is giving a nonsingular perturbed system.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The pivot indices: for $1 \leq i \leq n$, row <i>i</i> of the matrix has been interchanged with row <i>ipiv</i> (<i>i</i>).
<i>jpiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The pivot indices: for $1 \leq j \leq n$, column <i>j</i> of the matrix has been interchanged with column <i>jpiv</i> (<i>j</i>).
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>k</i> > 0, <i>U</i> (<i>k</i> , <i>k</i>) is likely to produce overflow if we try to solve for <i>x</i> in $A*x = b$. So <i>U</i> is perturbed to avoid the overflow.

?getf2

Computes the LU factorization of a general m -by- n matrix using partial pivoting with row interchanges (unblocked algorithm).

Syntax

```
call sgetf2( m, n, a, lda, ipiv, info )
call dgetf2( m, n, a, lda, ipiv, info )
call cgetf2( m, n, a, lda, ipiv, info )
call zgetf2( m, n, a, lda, ipiv, info )
```

Description

The routine computes the LU factorization of a general m -by- n matrix A using partial pivoting with row interchanges. The factorization has the form

$$A = P * L * U$$

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U is upper triangular (upper trapezoidal if $m < n$).

Input Parameters

m	INTEGER. The number of rows in the matrix A ($m \geq 0$).
n	INTEGER. The number of columns in A ($n \geq 0$).
a	REAL for sgetf2 DOUBLE PRECISION for dgetf2 COMPLEX for cgetf2 COMPLEX*16 for zgetf2. Array, DIMENSION ($lda, *$). Contains the matrix A to be factored. The second dimension of a must be at least $\max(1, n)$.
lda	INTEGER. The first dimension of a ; at least $\max(1, m)$.

Output Parameters

<i>a</i>	Overwritten by <i>L</i> and <i>U</i> . The unit diagonal elements of <i>L</i> are not stored.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, \min(m, n))$. The pivot indices: for $1 \leq i \leq n$, row <i>i</i> was interchanged with row <i>ipiv</i> (<i>i</i>).
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> > 0, <i>u</i> _{<i>ii</i>} is 0. The factorization has been completed, but <i>U</i> is exactly singular. Division by 0 will occur if you use the factor <i>U</i> for solving a system of linear equations.

?gtts2

Solves a system of linear equations with a tridiagonal matrix using the LU factorization computed by ?gttrf.

Syntax

```
call sgtts2( itrans, n, nrhs, dl, d, du, du2, ipiv, b, ldb )
call dgtts2( itrans, n, nrhs, dl, d, du, du2, ipiv, b, ldb )
call cgtts2( itrans, n, nrhs, dl, d, du, du2, ipiv, b, ldb )
call zgtts2( itrans, n, nrhs, dl, d, du, du2, ipiv, b, ldb )
```

Description

This routine solves for *x* one of the following systems of linear equations with multiple right hand sides:

$A * X = B$, $A^T * X = B$, or $A^H * X = B$ (for complex matrices only), with a tridiagonal matrix *A* using the *LU* factorization computed by [?gttrf](#).

Input Parameters

<i>itrans</i>	INTEGER. Must be 0, 1, or 2.

	Indicates the form of the equations to be solved: If $itrans = 0$, then $A * X = B$ (no transpose). If $itrans = 1$, then $A^T * X = B$ (transpose). If $itrans = 2$, then $A^H * X = B$ (conjugate transpose).
n	INTEGER. The order of the matrix A ($n \geq 0$).
$nrhs$	INTEGER. The number of right-hand sides, i.e., the number of columns in B ($nrhs \geq 0$).
$dl, d, du, du2, b$	REAL for <code>sgtts2</code> DOUBLE PRECISION for <code>dgtts2</code> COMPLEX for <code>cgtts2</code> COMPLEX*16 for <code>zgtts2</code> . Arrays: $dl(n - 1)$, $d(n)$, $du(n - 1)$, $du2(n - 2)$, $b(ldb, nrhs)$. The array dl contains the $(n - 1)$ multipliers that define the matrix L from the LU factorization of A . The array d contains the n diagonal elements of the upper triangular matrix U from the LU factorization of A . The array du contains the $(n - 1)$ elements of the first super-diagonal of U . The array $du2$ contains the $(n - 2)$ elements of the second super-diagonal of U . The array b contains the matrix B whose columns are the right-hand sides for the systems of equations.
ldb	INTEGER. The leading dimension of b ; must be $ldb \geq \max(1, n)$.
$ipiv$	INTEGER. Array, DIMENSION (n) . The pivot indices array, as returned by <code>?gttrf</code> .

Output Parameters

b	Overwritten by the solution matrix X .

?isnan

Tests input for NaN.

Syntax

```
val = sisnan( sin )
```

```
val = disnan( din )
```

Description

This logical routine returns `.TRUE.` if its argument is NaN, and `.FALSE.` otherwise.

Input Parameters

<i>sin</i>	REAL for <code>sisnan</code> Input to test for NaN.
<i>din</i>	DOUBLE PRECISION for <code>disnan</code> Input to test for NaN.

Output Parameters

<i>val</i>	Logical. Result of the test.

?laisnan

Tests input for NaN.

Syntax

```
val = slaisnan( sin1, sin2 )
```

```
val = dlaisnan( din1, din2 )
```

Description

This logical routine checks for NaNs (NaN stands for 'Not A Number') by comparing its two arguments for inequality. NaN is the only floating-point value where `NaN ≠ NaN` returns `.TRUE.` To check for NaNs, pass the same variable as both arguments.

This routine is not for general use. It exists solely to avoid over-optimization in `?isnan`.

Input Parameters

sin1, sin2 REAL for *sisnan*
Two numbers to compare for inequality.

din2, din2 DOUBLE PRECISION for *disnan*
Two numbers to compare for inequality.

Output Parameters

val Logical. Result of the comparison.

?labrd

Reduces the first nb rows and columns of a general matrix to a bidiagonal form.

Syntax

```
call slabrd( m, n, nb, a, lda, d, e, tauq, taup, x, ldx, y, ldy )
call dlabrd( m, n, nb, a, lda, d, e, tauq, taup, x, ldx, y, ldy )
call clabrd( m, n, nb, a, lda, d, e, tauq, taup, x, ldx, y, ldy )
call zlabrd( m, n, nb, a, lda, d, e, tauq, taup, x, ldx, y, ldy )
```

Description

The routine reduces the first nb rows and columns of a general m -by- n matrix A to upper or lower bidiagonal form by an orthogonal/unitary transformation Q'^*A^*P , and returns the matrices X and Y which are needed to apply the transformation to the unreduced part of A .

if $m \geq n$, A is reduced to upper bidiagonal form; if $m < n$, to lower bidiagonal form.

The matrices Q and P are represented as products of elementary reflectors: $Q = H(1) * (2) * \dots * H(nb)$, and $P = G(1) * G(2) * \dots * G(nb)$

Each $H(i)$ and $G(i)$ has the form

$$H(i) = I - \tau v v' \text{ and } G(i) = I - \tau u u'$$

where τ and τ are scalars, and v and u are vectors.

The elements of the vectors v and u together form the m -by- nb matrix V and the nb -by- n matrix U' which are needed, with X and Y , to apply the transformation to the unreduced part of the matrix, using a block update of the form: $A := A - V*Y' - X*U'$.

This is an auxiliary routine called by `?gebrd`.

Input Parameters

m	INTEGER. The number of rows in the matrix A ($m \geq 0$).
n	INTEGER. The number of columns in A ($n \geq 0$).
nb	INTEGER. The number of leading rows and columns of A to be reduced.
a	REAL for <code>slabrd</code> DOUBLE PRECISION for <code>dlabrd</code> COMPLEX for <code>clabrd</code> COMPLEX*16 for <code>zlabrd</code> . Array $a(lda,*)$ contains the matrix A to be reduced. The second dimension of a must be at least $\max(1, n)$.
lda	INTEGER. The first dimension of a ; at least $\max(1, m)$.
ldx	INTEGER. The first dimension of the output array x ; must beat least $\max(1, m)$.
ldy	INTEGER. The first dimension of the output array y ; must beat least $\max(1, n)$.

Output Parameters

a	On exit, the first nb rows and columns of the matrix are overwritten; the rest of the array is unchanged. if $m \geq n$, elements on and below the diagonal in the first nb columns, with the array τ_{auq} , represent the orthogonal/unitary matrix Q as a product of elementary reflectors; and elements above the diagonal in the first nb rows, with the array τ_{aup} , represent the orthogonal/unitary matrix P as a product of elementary reflectors. if $m < n$, elements below the diagonal in the first nb columns, with the array τ_{auq} , represent the orthogonal/unitary matrix Q as a product of elementary

reflectors, and elements on and above the diagonal in the first nb rows, with the array *taup*, represent the orthogonal/unitary matrix p as a product of elementary reflectors.

d, *e* REAL for single-precision flavors
 DOUBLE PRECISION for double-precision flavors. Arrays, DIMENSION (nb) each. The array *d* contains the diagonal elements of the first nb rows and columns of the reduced matrix:
 $d(i) = a(i, i)$.
 The array *e* contains the off-diagonal elements of the first nb rows and columns of the reduced matrix.

tauq, *taup* REAL for slabrd
 DOUBLE PRECISION for dlabrd
 COMPLEX for clabrd
 COMPLEX*16 for zlabrd.
 Arrays, DIMENSION (nb) each. Contain scalar factors of the elementary reflectors which represent the orthogonal/unitary matrices Q and P , respectively.

x, *y* REAL for slabrd
 DOUBLE PRECISION for dlabrd
 COMPLEX for clabrd
 COMPLEX*16 for zlabrd.
 Arrays, dimension $x(ldx, nb)$, $y(ldy, nb)$.
 The array *x* contains the m -by- nb matrix X required to update the unreduced part of A .
 The array *y* contains the n -by- nb matrix Y required to update the unreduced part of A .

Application Notes

if $m \geq n$, then for the elementary reflectors $H(i)$ and $G(i)$,

$v(1:i-1) = 0$, $v(i) = 1$, and $v(i:m)$ is stored on exit in $a(i:m, i)$; $u(1:i) = 0$, $u(i+1) = 1$, and $u(i+1:n)$ is stored on exit in $a(i, i+1:n)$;

tauq is stored in *tauq*(*i*) and *taup* in *taup*(*i*).

if $m < n$,

$v(1:i) = 0$, $v(i+1) = 1$, and $v(i+1:m)$ is stored on exit in $a(i+2:m, i)$; $u(1:i-1) = 0$, $u(i) = 1$, and $u(i:n)$ is stored on exit in $a(i, i+1:n)$; τ_{auq} is stored in $\tau_{auq}(i)$ and τ_{aup} in $\tau_{aup}(i)$.

The contents of a on exit are illustrated by the following examples with $nb = 2$:

$m=6, n=5$ ($m>n$)

$$\begin{bmatrix} 1 & 1 & u_1 & u_1 & u_1 \\ v_1 & 1 & 1 & u_2 & u_2 \\ v_1 & v_2 & a & a & a \\ v_1 & v_2 & a & a & a \\ v_1 & v_2 & a & a & a \\ v_1 & v_2 & a & a & a \end{bmatrix}$$

$m=5, n=6$ ($m<n$)

$$\begin{bmatrix} 1 & u_1 & u_1 & u_1 & u_1 & u_1 \\ 1 & 1 & u_2 & u_2 & u_2 & u_2 \\ v_1 & 1 & a & a & a & a \\ v_1 & v_2 & a & a & a & a \\ v_1 & v_2 & a & a & a & a \end{bmatrix}$$

where a denotes an element of the original matrix which is unchanged, v_i denotes an element of the vector defining $H(i)$, and u_i an element of the vector defining $G(i)$.

?lacn2

Estimates the 1-norm of a square matrix, using reverse communication for evaluating matrix-vector products.

Syntax

```
call slacn2( n, v, x, isgn, est, kase, isave )
call dlacn2( n, v, x, isgn, est, kase, isave )
call clacn2( n, v, x, est, kase, isave )
call zlacn2( n, v, x, est, kase, isave )
```

Description

This routine estimates the 1-norm of a square, real or complex matrix A . Reverse communication is used for evaluating matrix-vector products.

Input Parameters

n	INTEGER. The order of the matrix A ($n \geq 1$).
v, x	REAL for <code>slacn2</code> DOUBLE PRECISION for <code>dlacn2</code> COMPLEX for <code>clacn2</code> COMPLEX*16 for <code>zlacn2</code> . Arrays, DIMENSION (n) each. v is a workspace array. x is used as input after an intermediate return.
$isgn$	INTEGER. Workspace array, DIMENSION (n), used with real flavors only.
est	REAL for <code>slacn2/clacn2</code> DOUBLE PRECISION for <code>dlacn2/zlacn2</code> On entry with $kase$ set to 1 or 2, and $isave(1) = 1$, est must be unchanged from the previous call to the routine.
$kase$	INTEGER. On the initial call to the routine, $kase$ must be set to 0.
$isave$	INTEGER. Array, DIMENSION (3). Contains variables from the previous call to the routine.

Output Parameters

est	An estimate (a lower bound) for $\text{norm}(A)$.
$kase$	On an intermediate return, $kase$ is set to 1 or 2, indicating whether x is overwritten by $A*x$ or $A'*x$. On the final return, $kase$ is set to 0.
v	On the final return, $v = A*w$, where $est = \text{norm}(v) / \text{norm}(w)$ (w is not returned).
x	On an intermediate return, x is overwritten by $A*x$, if $kase = 1$, $A'*x$, if $kase = 2$,

isave (A' is the conjugate transpose of A), and the routine must be re-called with all the other parameters unchanged.

This parameter is used to save variables between calls to the routine.

slacon

Estimates the 1-norm of a square matrix, using reverse communication for evaluating matrix-vector products.

Syntax

```
call slacon( n, v, x, isgn, est, kase )
call dlacon( n, v, x, isgn, est, kase )
call clacon( n, v, x, est, kase )
call zlacon( n, v, x, est, kase )
```

Description

This routine estimates the 1-norm of a square, real/complex matrix A. Reverse communication is used for evaluating matrix-vector products.

Input Parameters

<i>n</i>	INTEGER. The order of the matrix A ($n \geq 1$).
<i>v, x</i>	REAL for slacon DOUBLE PRECISION for dlacon COMPLEX for clacon COMPLEX*16 for zlacon. Arrays, DIMENSION (<i>n</i>) each. <i>v</i> is a workspace array. <i>x</i> is used as input after an intermediate return.
<i>isgn</i>	INTEGER. Workspace array, DIMENSION (<i>n</i>), used with real flavors only.
<i>est</i>	REAL for slacon/clacon DOUBLE PRECISION for dlacon/zlacon

An estimate that with $kase=1$ or 2 should be unchanged from the previous call to `?lacon`.

kase INTEGER.
On the initial call to `?lacon`, *kase* should be 0 .

Output Parameters

est REAL for `slacon/clacon`
DOUBLE PRECISION for `dlacon/zlacon`
An estimate (a lower bound) for $\text{norm}(A)$.

kase On an intermediate return, *kase* will be 1 or 2 , indicating whether x should be overwritten by A^*x or A'^*x . On the final return from `?lacon`, *kase* will again be 0 .

v On the final return, $v = A^*w$, where $est = \text{norm}(v) / \text{norm}(w)$ (w is not returned).

x On an intermediate return, x should be overwritten by A^*x , if $kase = 1$,
 A'^*x , if $kase = 2$,
(where for complex flavors A' is the conjugate transpose of A), and `?lacon` must be re-called with all the other parameters unchanged.

?lacpy

Copies all or part of one two-dimensional array to another.

Syntax

```
call slacpy( uplo, m, n, a, lda, b, ldb )
call dlacpy( uplo, m, n, a, lda, b, ldb )
call clacpy( uplo, m, n, a, lda, b, ldb )
call zlacpy( uplo, m, n, a, lda, b, ldb )
```

Description

This routine copies all or part of a two-dimensional matrix A to another matrix B .

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies the part of the matrix <i>A</i> to be copied to <i>B</i>. If <i>uplo</i> = 'U', the upper triangular part of ; if <i>uplo</i> = 'L', the lower triangular part of <i>A</i>. Otherwise, all of the matrix <i>A</i> is copied.
<i>m</i>	INTEGER. The number of rows in the matrix <i>A</i> ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in <i>A</i> ($n \geq 0$).
<i>a</i>	REAL for slacpy DOUBLE PRECISION for dlacpy COMPLEX for clacpy COMPLEX*16 for zlacpy. Array <i>a</i>(<i>lda</i>,*), contains the <i>m</i>-by-<i>n</i> matrix <i>A</i>. The second dimension of <i>a</i> must be at least $\max(1, n)$. If <i>uplo</i> = 'U', only the upper triangle or trapezoid is accessed; if <i>uplo</i> = 'L', only the lower triangle or trapezoid is accessed.
<i>lda</i>	INTEGER. The first dimension of <i>a</i>; $lda \geq \max(1, m)$.
<i>ldb</i>	INTEGER. The first dimension of the output array <i>b</i>; $ldb \geq \max(1, m)$.

Output Parameters

<i>b</i>	REAL for slacpy DOUBLE PRECISION for dlacpy COMPLEX for clacpy COMPLEX*16 for zlacpy. Array <i>b</i>(<i>ldb</i>,*), contains the <i>m</i>-by-<i>n</i> matrix <i>B</i>. The second dimension of <i>b</i> must be at least $\max(1, n)$. On exit, $B = A$ in the locations specified by <i>uplo</i>.

?ladiv

Performs complex division in real arithmetic, avoiding unnecessary overflow.

Syntax

```
call sladiv( a, b, c, d, p, q )
call dladiv( a, b, c, d, p, q )
res = cladiv( x, y )
res = zladiv( x, y )
```

Description

The routines `sladiv/dladiv` perform complex division in real arithmetic as

$$p + iq = \frac{a + ib}{c + id}$$

Complex functions `cladiv/zladiv` compute the result as

$res = x/y$,

where x and y are complex. The computation of x/y will not overflow on an intermediary step unless the results overflows.

Input Parameters

a, b, c, d	REAL for <code>sladiv</code> DOUBLE PRECISION for <code>dladiv</code> The scalars a, b, c , and d in the above expression (for real flavors only).
x, y	COMPLEX for <code>cladiv</code> COMPLEX*16 for <code>zladiv</code> The complex scalars x and y (for complex flavors only).

Output Parameters

p, q	REAL for <code>sladiv</code>

	DOUBLE PRECISION for <code>dladiv</code>
	The scalars p and q in the above expression (for real flavors only).
<code>res</code>	COMPLEX for <code>cladiv</code>
	DOUBLE COMPLEX for <code>zladiv</code>
	Contains the result of division x / y .

?lae2

Computes the eigenvalues of a 2-by-2 symmetric matrix.

Syntax

```
call sla2( a, b, c, rt1, rt2 )
call dla2( a, b, c, rt1, rt2 )
```

Description

The routines `sla2/dla2` compute the eigenvalues of a 2-by-2 symmetric matrix

$$\begin{bmatrix} a & b \\ b & c \end{bmatrix}$$

On return, `rt1` is the eigenvalue of larger absolute value, and `rt2` is the eigenvalue of smaller absolute value.

Input Parameters

a, b, c	REAL for <code>sla2</code>
	DOUBLE PRECISION for <code>dla2</code>
	The elements a , b , and c of the 2-by-2 matrix above.

Output Parameters

<code>rt1, rt2</code>	REAL for <code>sla2</code>
	DOUBLE PRECISION for <code>dla2</code>

The computed eigenvalues of larger and smaller absolute value, respectively.

Application Notes

$rt1$ is accurate to a few ulps barring over/underflow. $rt2$ may be inaccurate if there is massive cancellation in the determinant $a*c-b*b$; higher precision or correctly rounded or correctly truncated arithmetic would be needed to compute $rt2$ accurately in all cases.

Overflow is possible only if $rt1$ is within a factor of 5 of overflow. Underflow is harmless if the input data is 0 or exceeds

$underflow\_threshold / macheps$.

?laebz

Computes the number of eigenvalues of a real symmetric tridiagonal matrix which are less than or equal to a given value, and performs other tasks required by the routine ?stebz.

Syntax

```
call slaebz( ijob, nitmax, n, mmax, minp, nbmin, abstol, reltol, pivmin, d,
e, e2, nval, ab, c, mout, nab, work, iwork, info )
```

```
call dlaebz( ijob, nitmax, n, mmax, minp, nbmin, abstol, reltol, pivmin, d,
e, e2, nval, ab, c, mout, nab, work, iwork, info )
```

Description

The routine ?laebz contains the iteration loops which compute and use the function $n(w)$, which is the count of eigenvalues of a symmetric tridiagonal matrix T less than or equal to its argument w . It performs a choice of two types of loops:

$ijob$	=1, followed by
$ijob$	=2: It takes as input a list of intervals and returns a list of sufficiently small intervals whose union contains the same eigenvalues as the union of the original intervals. The input intervals are $(ab(j,1), ab(j,2)]$, $j=1, \dots, minp$. The output interval $(ab(j,1), ab(j,2)]$ will contain eigenvalues $nab(j,1)+1, \dots, nab(j,2)$, where $1 \leq j \leq mout$.

ijob =3: It performs a binary search in each input interval $(ab(j, 1), ab(j, 2)]$ for a point $w(j)$ such that $n(w(j)) = nval(j)$, and uses $c(j)$ as the starting point of the search. If such a $w(j)$ is found, then on output $ab(j, 1) = ab(j, 2) = w$. If no such $w(j)$ is found, then on output $(ab(j, 1), ab(j, 2)]$ will be a small interval containing the point where $n(w)$ jumps through $nval(j)$, unless that point lies outside the initial interval.

Note that the intervals are in all cases half-open intervals, that is, of the form $(a, b]$, which includes b but not a .

To avoid underflow, the matrix should be scaled so that its largest element is no greater than $overflow^{1/2} * overflow^{1/4}$ in absolute value. To assure the most accurate computation of small eigenvalues, the matrix should be scaled to be not much smaller than that, either.



NOTE. In general, the arguments are not checked for unreasonable values.

Input Parameters

ijob INTEGER. Specifies what is to be done:
 = 1: Compute *nab* for the initial intervals.
 = 2: Perform bisection iteration to find eigenvalues of *T*.
 = 3: Perform bisection iteration to invert $n(w)$, i.e., to find a point which has a specified number of eigenvalues of *T* to its left. Other values will cause ?laebz to return with *info*=-1.

nitmax INTEGER. The maximum number of "levels" of bisection to be performed, i.e., an interval of width *w* will not be made smaller than $2^{-nitmax} * w$. If not all intervals have converged after *nitmax* iterations, then *info* is set to the number of non-converged intervals.

n INTEGER. The dimension *n* of the tridiagonal matrix *T*. It must be at least 1.

mmax INTEGER. The maximum number of intervals. If more than *mmax* intervals are generated, then ?laebz will quit with *info*=*mmax*+1.

<i>minp</i>	INTEGER. The initial number of intervals. It may not be greater than <i>mmax</i> .
<i>nbmin</i>	INTEGER. The smallest number of intervals that should be processed using a vector loop. If zero, then only the scalar loop will be used.
<i>abstol</i>	REAL for slaebz DOUBLE PRECISION for dlaebz. The minimum (absolute) width of an interval. When an interval is narrower than <i>abstol</i> , or than <i>reltol</i> times the larger (in magnitude) endpoint, then it is considered to be sufficiently small, i.e., converged. This must be at least zero.
<i>reltol</i>	REAL for slaebz DOUBLE PRECISION for dlaebz. The minimum relative width of an interval. When an interval is narrower than <i>abstol</i> , or than <i>reltol</i> times the larger (in magnitude) endpoint, then it is considered to be sufficiently small, i.e., converged. Note: this should always be at least <i>radix*machine epsilon</i> .
<i>pivmin</i>	REAL for slaebz DOUBLE PRECISION for dlaebz. The minimum absolute value of a "pivot" in the Sturm sequence loop. This value must be at least ($\max e(j)**2 * safe\_min$) and at least <i>safe_min</i> , where <i>safe_min</i> is at least the smallest number that can divide one without overflow.
<i>d, e, e2</i>	REAL for slaebz DOUBLE PRECISION for dlaebz. Arrays, dimension (<i>n</i>) each. The array <i>d</i> contains the diagonal elements of the tridiagonal matrix <i>T</i> . The array <i>e</i> contains the off-diagonal elements of the tridiagonal matrix <i>T</i> in positions 1 through <i>n</i> -1. <i>e</i> (<i>n</i>) is arbitrary. The array <i>e2</i> contains the squares of the off-diagonal elements of the tridiagonal matrix <i>T</i> . <i>e2</i> (<i>n</i>) is ignored.
<i>nval</i>	INTEGER. Array, dimension (<i>minp</i>). If <i>ijob</i> =1 or 2, not referenced.

	If $ijob=3$, the desired values of $n(w)$.
<i>ab</i>	REAL for slaebz DOUBLE PRECISION for dlaebz. Array, dimension ($mmax, 2$) The endpoints of the intervals. $ab(j,1)$ is $a(j)$, the left endpoint of the j-th interval, and $ab(j,2)$ is $b(j)$, the right endpoint of the j-th interval.
<i>c</i>	REAL for slaebz DOUBLE PRECISION for dlaebz. Array, dimension ($mmax$) If $ijob=1$, ignored. If $ijob=2$, workspace. If $ijob=3$, then on input $c(j)$ should be initialized to the first search point in the binary search.
<i>nab</i>	INTEGER. Array, dimension ($mmax, 2$) If $ijob=2$, then on input, $nab(i,j)$ should be set. It must satisfy the condition: $n(ab(i,1)) \leq nab(i,1) \leq nab(i,2) \leq n(ab(i,2))$, which means that in interval i only eigenvalues $nab(i,1)+1, \dots, nab(i,2)$ are considered. Usually, $nab(i,j)=n(ab(i,j))$, from a previous call to ?laebz with $ijob=1$. If $ijob=3$, normally, nab should be set to some distinctive value(s) before ?laebz is called.
<i>work</i>	REAL for slaebz DOUBLE PRECISION for dlaebz. Workspace array, dimension ($mmax$).
<i>iwork</i>	INTEGER. Workspace array, dimension ($mmax$).

Output Parameters

<i>nval</i>	The elements of <i>nval</i> will be reordered to correspond with the intervals in <i>ab</i>. Thus, $nval(j)$ on output will not, in general be the same as $nval(j)$ on input, but it will correspond with the interval $(ab(j,1), ab(j,2)]$ on output.

<i>ab</i>	The input intervals will, in general, be modified, split, and reordered by the calculation.
<i>mout</i>	INTEGER. If <i>ijob</i> =1, the number of eigenvalues in the intervals. If <i>ijob</i> =2 or 3, the number of intervals output. If <i>ijob</i> =3, <i>mout</i> will equal <i>minp</i> .
<i>nab</i>	If <i>ijob</i> =1, then on output <i>nab</i> (<i>i,j</i>) will be set to $N(ab(i,j))$. If <i>ijob</i> =2, then on output, <i>nab</i> (<i>i,j</i>) will contain $\max(na(k), \min(nb(k), N(ab(i,j))))$, where <i>k</i> is the index of the input interval that the output interval (<i>ab</i> (<i>j</i> ,1), <i>ab</i> (<i>j</i> ,2)] came from, and <i>na</i> (<i>k</i>) and <i>nb</i> (<i>k</i>) are the input values of <i>nab</i> (<i>k</i> ,1) and <i>nab</i> (<i>k</i> ,2). If <i>ijob</i> =3, then on output, <i>nab</i> (<i>i,j</i>) contains $N(ab(i,j))$, unless $N(w) > nval(i)$ for all search points <i>w</i> , in which case <i>nab</i> (<i>i</i> ,1) will not be modified, i.e., the output value will be the same as the input value (modulo reorderings, see <i>nval</i> and <i>ab</i>), or unless $N(w) < nval(i)$ for all search points <i>w</i> , in which case <i>nab</i> (<i>i</i> ,2) will not be modified.
<i>info</i>	INTEGER. If <i>info</i> = 0 - all intervals converged If <i>info</i> = 1-- <i>mmax</i> - the last <i>info</i> interval did not converge. If <i>info</i> = <i>mmax</i> +1 - more than <i>mmax</i> intervals were generated

Application Notes

This routine is intended to be called only by other LAPACK routines, thus the interface is less user-friendly. It is intended for two purposes:

(a) finding eigenvalues. In this case, *?laebz* should have one or more initial intervals set up in *ab*, and *?laebz* should be called with *ijob*=1. This sets up *nab*, and also counts the eigenvalues. Intervals with no eigenvalues would usually be thrown out at this point. Also, if not all the eigenvalues in an interval *i* are desired, *nab*(*i*,1) can be increased or *nab*(*i*,2) decreased. For example, set *nab*(*i*,1)=*nab*(*i*,2)-1 to get the largest eigenvalue. *?laebz* is then called with *ijob*=2 and *mmax* no smaller than the value of *mout* returned by the call with *ijob*=1. After this (*ijob*=2) call, eigenvalues *nab*(*i*,1)+1 through *nab*(*i*,2) are approximately *ab*(*i*,1) (or *ab*(*i*,2)) to the tolerance specified by *abstol* and *reltol*.

(b) finding an interval $(a', b']$ containing eigenvalues $w(f), \dots, w(l)$. In this case, start with a Gershgorin interval (a, b) . Set up ab to contain 2 search intervals, both initially (a, b) . One $nval$ element should contain $f-1$ and the other should contain l , while c should contain a and b , respectively. $nab(i, 1)$ should be -1 and $nab(i, 2)$ should be $n+1$, to flag an error if the desired interval does not lie in (a, b) . `?laebz` is then called with $ijob=3$. On exit, if $w(f-1) < w(f)$, then one of the intervals -- j -- will have $ab(j, 1)=ab(j, 2)$ and $nab(j, 1)=nab(j, 2)=f-1$, while if, to the specified tolerance, $w(f-k) = \dots = w(f+r)$, $k > 0$ and $r \geq 0$, then the interval will have $n(ab(j, 1))=nab(j, 1)=f-k$ and $n(ab(j, 2))=nab(j, 2)=f+r$. The cases $w(l) < w(l+1)$ and $w(l-r) = \dots = w(l+k)$ are handled similarly.

?laed0

Used by ?stedc. Computes all eigenvalues and corresponding eigenvectors of an unreduced symmetric tridiagonal matrix using the divide and conquer method.

Syntax

```
call slaed0( icompg, qsiz, n, d, e, q, ldq, qstore, ldqs, work, iwork, info
)
call dlaed0( icompg, qsiz, n, d, e, q, ldq, qstore, ldqs, work, iwork, info
)
call claed0( qsiz, n, d, e, q, ldq, qstore, ldqs, rwork, iwork, info )
call zlaed0( qsiz, n, d, e, q, ldq, qstore, ldqs, rwork, iwork, info )
```

Description

Real flavors of this routine compute all eigenvalues and (optionally) corresponding eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method.

Complex flavors `claed0/zlaed0` compute all eigenvalues of a symmetric tridiagonal matrix which is one diagonal block of those from reducing a dense or band Hermitian matrix and corresponding eigenvectors of the dense or band matrix.

Input Parameters

<i>icompg</i>	INTEGER. Used with real flavors only. If <i>icompg</i> = 0, compute eigenvalues only.

If $icompg = 1$, compute eigenvectors of original dense symmetric matrix also.
 On entry, the array q must contain the orthogonal matrix used to reduce the original matrix to tridiagonal form.
 If $icompg = 2$, compute eigenvalues and eigenvectors of the tridiagonal matrix.

$qsiz$ INTEGER.
 The dimension of the orthogonal/unitary matrix used to reduce the full matrix to tridiagonal form; $qsiz \geq n$ (for real flavors, $qsiz \geq n$ if $icompg = 1$).

n INTEGER. The dimension of the symmetric tridiagonal matrix ($n \geq 0$).

$d, e, rwork$ REAL for single-precision flavors
 DOUBLE PRECISION for double-precision flavors. Arrays:
 $d(*)$ contains the main diagonal of the tridiagonal matrix. The dimension of d must be at least $\max(1, n)$.
 $e(*)$ contains the off-diagonal elements of the tridiagonal matrix. The dimension of e must be at least $\max(1, n-1)$.
 $rwork(*)$ is a workspace array used in complex flavors only. The dimension of $rwork$ must be at least $(1 + 3n + 2n \lg(n) + 3n^2)$, where $\lg(n)$ = smallest integer k such that $2^k \geq n$.

$q, qstore$ REAL for slaed0
 DOUBLE PRECISION for dlaed0
 COMPLEX for claed0
 COMPLEX*16 for zlaed0.
 Arrays: $q(ldq, *)$, $qstore(ldqs, *)$. The second dimension of these arrays must be at least $\max(1, n)$.
For real flavors:
 If $icompg = 0$, array q is not referenced.
 If $icompg = 1$, on entry, q is a subset of the columns of the orthogonal matrix used to reduce the full matrix to tridiagonal form corresponding to the subset of the full matrix which is being decomposed at this time.

If $icompq = 2$, on entry, q will be the identity matrix. The array $qstore$ is a workspace array referenced only when $icompq = 1$. Used to store parts of the eigenvector matrix when the updating matrix multiplies take place.

For complex flavors:

On entry, q must contain an $qsiz$ -by- n matrix whose columns are unitarily orthonormal. It is a part of the unitary matrix that reduces the full dense Hermitian matrix to a (reducible) symmetric tridiagonal matrix. The array $qstore$ is a workspace array used to store parts of the eigenvector matrix when the updating matrix multiplies take place.

ldq	INTEGER. The first dimension of the array q ; $ldq \geq \max(1, n)$.
$ldqs$	INTEGER. The first dimension of the array $qstore$; $ldqs \geq \max(1, n)$.
$work$	REAL for slaed0 DOUBLE PRECISION for dlaed0. Workspace array, used in real flavors only. If $icompq = 0$ or 1 , the dimension of $work$ must be at least $(1 + 3n + 2n \lg(n) + 2n^2)$, where $\lg(n) =$ smallest integer k such that $2^k \geq n$. If $icompq = 2$, the dimension of $work$ must be at least $(4n + n^2)$.
$iwork$	INTEGER. Workspace array. For real flavors, if $icompq = 0$ or 1 , and for complex flavors, the dimension of $iwork$ must be at least $(6 + 6n + 5n \lg(n))$. For real flavors, if $icompq = 2$, the dimension of $iwork$ must be at least $(3 + 5n)$.

Output Parameters

d	On exit, contains eigenvalues in ascending order.
e	On exit, the array is destroyed.
q	If $icompq = 2$, on exit, q contains the eigenvectors of the tridiagonal matrix.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.
 If *info* = *i* > 0, the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns *i*/(*n*+1) through mod(*i*, *n*+1).

?laed1

Used by sstedc/dstedc. Computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. Used when the original matrix is tridiagonal.

Syntax

```
call slaed1( n, d, q, ldq, indxq, rho, cutpnt, work, iwork, info )
call dlaed1( n, d, q, ldq, indxq, rho, cutpnt, work, iwork, info )
```

Description

The routine ?laed1 computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. This routine is used only for the eigenproblem which requires all eigenvalues and eigenvectors of a tridiagonal matrix. ?laed7 handles the case in which eigenvalues only or eigenvalues and eigenvectors of a full symmetric matrix (which was reduced to tridiagonal form) are desired.

$$T = Q(\text{in}) * (D(\text{in}) + \text{rho} * Z * Z') * Q'(\text{in}) = Q(\text{out}) * D(\text{out}) * Q'(\text{out})$$

where $Z = Q'u$, u is a vector of length n with ones in the *cutpnt* and (*cutpnt*+1) -th elements and zeros elsewhere. The eigenvectors of the original matrix are stored in Q , and the eigenvalues are in D . The algorithm consists of three stages:

The first stage consists of deflating the size of the problem when there are multiple eigenvalues or if there is a zero in the z vector. For each such occurrence the dimension of the secular equation problem is reduced by one. This stage is performed by the routine ?laed2.

The second stage consists of calculating the updated eigenvalues. This is done by finding the roots of the secular equation via the routine ?laed4 (as called by ?laed3). This routine also calculates the eigenvectors of the current problem.

The final stage consists of computing the updated eigenvectors directly using the updated eigenvalues. The eigenvectors for the current problem are multiplied with the eigenvectors from the overall problem.

Input Parameters

<i>n</i>	INTEGER. The dimension of the symmetric tridiagonal matrix ($n \geq 0$).
<i>d, q, work</i>	REAL for slaed1 DOUBLE PRECISION for dlaed1. Arrays: <i>d</i> (*) contains the eigenvalues of the rank-1-perturbed matrix. The dimension of <i>d</i> must be at least $\max(1, n)$. <i>q</i> (ldq, *) contains the eigenvectors of the rank-1-perturbed matrix. The second dimension of <i>q</i> must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array, dimension at least $(4n+n^2)$.
<i>ldq</i>	INTEGER. The first dimension of the array <i>q</i> ; $ldq \geq \max(1, n)$.
<i>indxq</i>	INTEGER. Array, dimension (<i>n</i>). On entry, the permutation which separately sorts the two subproblems in <i>d</i> into ascending order.
<i>rho</i>	REAL for slaed1 DOUBLE PRECISION for dlaed1. The subdiagonal entry used to create the rank-1 modification. This parameter can be modified by ?laed2, where it is input/output.
<i>cutpnt</i>	INTEGER. The location of the last eigenvalue in the leading sub-matrix. $\min(1, n) \leq cutpnt \leq n/2$.
<i>iwork</i>	INTEGER. Workspace array, dimension $(4n)$.

Output Parameters

<i>d</i>	On exit, contains the eigenvalues of the repaired matrix.

<i>q</i>	On exit, <i>q</i> contains the eigenvectors of the repaired tridiagonal matrix.
<i>indxq</i>	On exit, contains the permutation which will reintegrate the subproblems back into sorted order, that is, <i>d</i> (<i>indxq</i> (<i>i</i> = 1, <i>n</i>)) will be in ascending order.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = 1, an eigenvalue did not converge.

?laed2

Used by sstedc/dstedc. Merges eigenvalues and deflates secular equation. Used when the original matrix is tridiagonal.

Syntax

```
call slaed2( k, n, n1, d, q, ldq, indxq, rho, z, dlamda, w, q2, indx, indxc,
            indxpr, coltyp, info )
```

```
call dlaed2( k, n, n1, d, q, ldq, indxq, rho, z, dlamda, w, q2, indx, indxc,
            indxpr, coltyp, info )
```

Description

The routine ?laed2 merges the two sets of eigenvalues together into a single sorted set. Then it tries to deflate the size of the problem. There are two ways in which deflation can occur: when two or more eigenvalues are close together or if there is a tiny entry in the *z* vector. For each such occurrence the order of the related secular equation problem is reduced by one.

Input Parameters

<i>k</i>	INTEGER. The number of non-deflated eigenvalues, and the order of the related secular equation ($0 \leq k \leq n$).
<i>n</i>	INTEGER. The dimension of the symmetric tridiagonal matrix ($n \geq 0$).
<i>n1</i>	INTEGER. The location of the last eigenvalue in the leading sub-matrix; $\min(1, n) \leq n1 \leq n/2$.

<i>d, q, z</i>	REAL for slaed2 DOUBLE PRECISION for dlaed2. Arrays: <i>d</i>(*) contains the eigenvalues of the two submatrices to be combined. The dimension of <i>d</i> must be at least $\max(1, n)$. <i>q</i>(ldq, *) contains the eigenvectors of the two submatrices in the two square blocks with corners at (1,1), (n1,n1) and (n1+1,n1+1), (n,n). The second dimension of <i>q</i> must be at least $\max(1, n)$. <i>z</i>(*) contains the updating vector (the last row of the first sub-eigenvector matrix and the first row of the second sub-eigenvector matrix).
<i>ldq</i>	INTEGER. The first dimension of the array <i>q</i>; $ldq \geq \max(1, n)$.
<i>indxq</i>	INTEGER. Array, dimension (n). On entry, the permutation which separately sorts the two subproblems in <i>d</i> into ascending order. Note that elements in the second half of this permutation must first have <i>n1</i> added to their values.
<i>rho</i>	REAL for slaed2 DOUBLE PRECISION for dlaed2. On entry, the off-diagonal element associated with the rank-1 cut which originally split the two submatrices which are now being recombined.
<i>indx, indxp</i>	INTEGER. Workspace arrays, dimension (n) each. Array <i>indx</i> contains the permutation used to sort the contents of <i>dlamda</i> into ascending order. Array <i>indxp</i> contains the permutation used to place deflated values of <i>d</i> at the end of the array. <i>indxp</i>(1:k) points to the nondeflated <i>d</i>-values and <i>indxp</i>(k+1:n) points to the deflated eigenvalues.
<i>coltyp</i>	INTEGER. Workspace array, dimension (n). During execution, a label which will indicate which of the following types a column in the <i>q2</i> matrix is: 1 : non-zero in the upper half only;

2 : dense;
 3 : non-zero in the lower half only;
 4 : deflated.

Output Parameters

d On exit, *d* contains the trailing $(n-k)$ updated eigenvalues (those which were deflated) sorted into increasing order.

q On exit, *q* contains the trailing $(n-k)$ updated eigenvectors (those which were deflated) in its last $n-k$ columns.

z On exit, *z* content is destroyed by the updating process.

indxq Destroyed on exit.

rho On exit, *rho* has been modified to the value required by ?laed3.

dlamda, *w*, *q2* REAL for slaed2
 DOUBLE PRECISION for dlaed2.
 Arrays: *dlamda*(*n*), *w*(*n*), *q2*($n1^2 + (n-n1)^2$).
 The array *dlamda* contains a copy of the first *k* eigenvalues which is used by ?laed3 to form the secular equation.
 The array *w* contains the first *k* values of the final deflation-altered *z*-vector which is passed to ?laed3.
 The array *q2* contains a copy of the first *k* eigenvectors which is used by ?laed3 in a matrix multiply (sgemm/dgemm) to solve for the new eigenvectors.

indx INTEGER. Array, dimension (*n*).
 The permutation used to arrange the columns of the deflated *q* matrix into three groups: the first group contains non-zero elements only at and above *n1*, the second contains non-zero elements only below *n1*, and the third is dense.

coltyp On exit, *coltyp*(*i*) is the number of columns of type *i*, for *i*=1 to 4 only (see the definition of types in the description of *coltyp* in *Input Parameters*).

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

?laed3

Used by sstedc/dstedc. Finds the roots of the secular equation and updates the eigenvectors. Used when the original matrix is tridiagonal.

Syntax

```
call slaed3( k, n, n1, d, q, ldq, rho, dlamda, q2, indx, ctot, w, s, info )
call dlaed3( k, n, n1, d, q, ldq, rho, dlamda, q2, indx, ctot, w, s, info )
```

Description

The routine ?laed3 finds the roots of the secular equation, as defined by the values in *d*, *w*, and *rho*, between 1 and *k*.

It makes the appropriate calls to ?laed4 and then updates the eigenvectors by multiplying the matrix of eigenvectors of the pair of eigensystems being combined by the matrix of eigenvectors of the *k*-by-*k* system which is solved here.

This code makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but none are known.

Input Parameters

<i>k</i>	INTEGER. The number of terms in the rational function to be solved by ?laed4 ($k \geq 0$).
<i>n</i>	INTEGER. The number of rows and columns in the <i>q</i> matrix. $n \geq k$ (deflation may result in $n > k$).
<i>n1</i>	INTEGER. The location of the last eigenvalue in the leading sub-matrix; $\min(1, n) \leq n1 \leq n/2$.
<i>q</i>	REAL for slaed3 DOUBLE PRECISION for dlaed3. Array <i>q</i> (<i>ldq</i> , *). The second dimension of <i>q</i> must be at least $\max(1, n)$. Initially, the first <i>k</i> columns of this array are used as workspace.

<i>ldq</i>	INTEGER. The first dimension of the array <i>q</i> ; $ldq \geq \max(1, n)$.
<i>rho</i>	REAL for slaed3 DOUBLE PRECISION for dlaed3. The value of the parameter in the rank one update equation. $\rho \geq 0$ required.
<i>dlambda, q2, w</i>	REAL for slaed3 DOUBLE PRECISION for dlaed3. Arrays: <i>dlambda(k)</i> , <i>q2(ldq2, *)</i> , <i>w(k)</i> . The first <i>k</i> elements of the array <i>dlambda</i> contain the old roots of the deflated updating problem. These are the poles of the secular equation. The first <i>k</i> columns of the array <i>q2</i> contain the non-deflated eigenvectors for the split problem. The second dimension of <i>q2</i> must be at least $\max(1, n)$. The first <i>k</i> elements of the array <i>w</i> contain the components of the deflation-adjusted updating vector.
<i>indx</i>	INTEGER. Array, dimension (<i>n</i>). The permutation used to arrange the columns of the deflated <i>q</i> matrix into three groups (see ?laed2). The rows of the eigenvectors found by ?laed4 must be likewise permuted before the matrix multiply can take place.
<i>ctot</i>	INTEGER. Array, dimension (4). A count of the total number of the various types of columns in <i>q</i> , as described in <i>indx</i> . The fourth column type is any column which has been deflated.
<i>s</i>	REAL for slaed3 DOUBLE PRECISION for dlaed3. Workspace array, dimension $(n1+1)*k$. Will contain the eigenvectors of the repaired matrix which will be multiplied by the previously accumulated eigenvectors to update the system.

Output Parameters

<i>d</i>	REAL for slaed3 DOUBLE PRECISION for dlaed3.

	Array, dimension at least $\max(1, n)$.
	$d(i)$ contains the updated eigenvalues for $1 \leq i \leq k$.
q	On exit, the columns 1 to k of q contain the updated eigenvectors.
$d\lambda mda$	May be changed on output by having lowest order bit set to zero on Cray X-MP, Cray Y-MP, Cray-2, or Cray C-90, as described above.
w	Destroyed on exit.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value. If $info = 1$, an eigenvalue did not converge.

?laed4

Used by sstedc/dstedc. Finds a single root of the secular equation.

Syntax

```
call slaed4( n, i, d, z, delta, rho, dlam, info )
call dlaed4( n, i, d, z, delta, rho, dlam, info )
```

Description

This subroutine computes the i -th updated eigenvalue of a symmetric rank-one modification to a diagonal matrix whose elements are given in the array d , and that

$$D(i) < D(j) \text{ for } i < j$$

and that $\rho > 0$. This is arranged by the calling routine, and is no loss in generality. The rank-one modified system is thus

$$\text{diag}(D) + \rho * Z * \text{transpose}(Z).$$

where we assume the Euclidean norm of Z is 1.

The method consists of approximating the rational functions in the secular equation by simpler interpolating rational functions.

Input Parameters

<i>n</i>	INTEGER. The length of all arrays.
<i>i</i>	INTEGER. The index of the eigenvalue to be computed; $1 \leq i \leq n$.
<i>d</i> , <i>z</i>	REAL for slaed4 DOUBLE PRECISION for dlaed4 Arrays, dimension (<i>n</i>) each. The array <i>d</i> contains the original eigenvalues. It is assumed that they are in order, $d(i) < d(j)$ for $i < j$. The array <i>z</i> contains the components of the updating vector <i>Z</i> .
<i>rho</i>	REAL for slaed4 DOUBLE PRECISION for dlaed4 The scalar in the symmetric updating formula.

Output Parameters

<i>delta</i>	REAL for slaed4 DOUBLE PRECISION for dlaed4 Array, dimension (<i>n</i>). If $n \neq 1$, <i>delta</i> contains $(d(j) - \lambda_i)$ in its <i>j</i> -th component. If $n = 1$, then $delta(1) = 1$. The vector <i>delta</i> contains the information necessary to construct the eigenvectors.
<i>diam</i>	REAL for slaed4 DOUBLE PRECISION for dlaed4 The computed λ_i , the <i>i</i> -th updated eigenvalue.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = 1, the updating process failed.

slaed5

Used by sstedc/dstedc. Solves the 2-by-2 secular equation.

Syntax

```
call slaed5( i, d, z, delta, rho, dlam )
call dlaed5( i, d, z, delta, rho, dlam )
```

Description

This subroutine computes the i -th eigenvalue of a symmetric rank-one modification of a 2-by-2 diagonal matrix

$$\text{diag}(D) + \rho * Z * \text{transpose}(Z).$$

The diagonal elements in the array D are assumed to satisfy

$$D(i) < D(j) \text{ for } i < j.$$

We also assume $\rho > 0$ and that the Euclidean norm of the vector Z is one.

Input Parameters

i	INTEGER. The index of the eigenvalue to be computed; $1 \leq i \leq 2$.
d, z	REAL for slaed5 DOUBLE PRECISION for dlaed5 Arrays, dimension (2) each. The array d contains the original eigenvalues. It is assumed that $d(1) < d(2)$. The array z contains the components of the updating vector.
ρ	REAL for slaed5 DOUBLE PRECISION for dlaed5 The scalar in the symmetric updating formula.

Output Parameters

δ	REAL for slaed5 DOUBLE PRECISION for dlaed5 Array, dimension (2).

The vector *delta* contains the information necessary to construct the eigenvectors.

diam REAL for slaed5
 DOUBLE PRECISION for dlaed5
 The computed *lambda_i*, the *i*-th updated eigenvalue.

?laed6

Used by sstedc/dstedc. Computes one Newton step in solution of the secular equation.

Syntax

```
call slaed6( kniter, orgati, rho, d, z, finit, tau, info )
call dlaed6( kniter, orgati, rho, d, z, finit, tau, info )
```

Description

This routine computes the positive or negative root (closest to the origin) of

$$f(x) = rho + \frac{z(1)}{d(1) - x} + \frac{z(2)}{d(2) - x} + \frac{z(3)}{d(3) - x}$$

It is assumed that if *orgati* = .TRUE. the root is between *d*(2) and *d*(3); otherwise it is between *d*(1) and *d*(2). This routine is called by ?laed4 when necessary. In most cases, the root sought is the smallest in magnitude, though it might not be in some extremely rare situations.

Input Parameters

kniter INTEGER.
 Refer to ?laed4 for its significance.

orgati LOGICAL.
 If *orgati* = .TRUE., the needed root is between *d*(2) and *d*(3); otherwise it is between *d*(1) and *d*(2). See ?laed4 for further details.

rho REAL for slaed6
 DOUBLE PRECISION for dlaed6

<i>d, z</i>	Refer to the equation for $f(x)$ above. REAL for slaed6 DOUBLE PRECISION for dlaed6 Arrays, dimension (3) each. The array <i>d</i> satisfies $d(1) < d(2) < d(3)$. Each of the elements in the array <i>z</i> must be positive.
<i>finit</i>	REAL for slaed6 DOUBLE PRECISION for dlaed6 The value of $f(x)$ at 0. It is more accurate than the one evaluated inside this routine (if someone wants to do so).

Output Parameters

<i>tau</i>	REAL for slaed6 DOUBLE PRECISION for dlaed6 The root of the equation for $f(x)$.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = 1, failure to converge.

?laed7

Used by ?stedc. Computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. Used when the original matrix is dense.

Syntax

```
call slaed7( icompg, n, qsiz, tlvl, curlvl, curpbm, d, q, ldq, indxq, rho,
cutpnt, qstore, qptr, prmptr, perm, givptr, givcol, givnum, work, iwork, info
)

call dlaed7( icompg, n, qsiz, tlvl, curlvl, curpbm, d, q, ldq, indxq, rho,
cutpnt, qstore, qptr, prmptr, perm, givptr, givcol, givnum, work, iwork, info
)

call claed7( n, cutpnt, qsiz, tlvl, curlvl, curpbm, d, q, ldq, rho, indxq,
qstore, qptr, prmptr, perm, givptr, givcol, givnum, work, rwork, iwork, info
)

call zlaed7( n, cutpnt, qsiz, tlvl, curlvl, curpbm, d, q, ldq, rho, indxq,
qstore, qptr, prmptr, perm, givptr, givcol, givnum, work, rwork, iwork, info
)
```

Description

The routine ?laed7 computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. This routine is used only for the eigenproblem which requires all eigenvalues and optionally eigenvectors of a dense symmetric/Hermitian matrix that has been reduced to tridiagonal form. For real flavors, slaed1/dlaed1 handles the case in which all eigenvalues and eigenvectors of a symmetric tridiagonal matrix are desired.

$$T = Q(\text{in}) * (D(\text{in}) + \rho * Z * Z') * Q'(\text{in}) = Q(\text{out}) * D(\text{out}) * Q'(\text{out})$$

where $Z = Q' * u$, u is a vector of length n with ones in the cutpnt and $(\text{cutpnt} + 1)$ -th elements and zeros elsewhere. The eigenvectors of the original matrix are stored in Q , and the eigenvalues are in D . The algorithm consists of three stages:

The first stage consists of deflating the size of the problem when there are multiple eigenvalues or if there is a zero in the z vector. For each such occurrence the dimension of the secular equation problem is reduced by one. This stage is performed by the routine slaed8/dlaed8 (for real flavors) or by the routine slaed2/dlaed2 (for complex flavors).

The second stage consists of calculating the updated eigenvalues. This is done by finding the roots of the secular equation via the routine ?laed4 (as called by ?laed9 or ?laed3). This routine also calculates the eigenvectors of the current problem.

The final stage consists of computing the updated eigenvectors directly using the updated eigenvalues. The eigenvectors for the current problem are multiplied with the eigenvectors from the overall problem.

Input Parameters

<i>icompg</i>	INTEGER. Used with real flavors only. If <i>icompg</i> = 0, compute eigenvalues only. If <i>icompg</i> = 1, compute eigenvectors of original dense symmetric matrix also. On entry, the array <i>q</i> must contain the orthogonal matrix used to reduce the original matrix to tridiagonal form.
<i>n</i>	INTEGER. The dimension of the symmetric tridiagonal matrix ($n \geq 0$).
<i>cutpnt</i>	INTEGER. The location of the last eigenvalue in the leading sub-matrix. $\min(1, n) \leq \text{cutpnt} \leq n$.
<i>qsiz</i>	INTEGER. The dimension of the orthogonal/unitary matrix used to reduce the full matrix to tridiagonal form; $qsiz \geq n$ (for real flavors, $qsiz \geq n$ if <i>icompg</i> = 1).
<i>tlvls</i>	INTEGER. The total number of merging levels in the overall divide and conquer tree.
<i>curlvl</i>	INTEGER. The current level in the overall merge routine, $0 \leq \text{curlvl} \leq \text{tlvls}$.
<i>curpbm</i>	INTEGER. The current problem in the current level in the overall merge routine (counting from upper left to lower right).
<i>d</i>	REAL for slaed7/claed7 DOUBLE PRECISION for dlaed7/zlaed7. Array, dimension at least $\max(1, n)$. Array <i>d</i>(*) contains the eigenvalues of the rank-1-perturbed matrix.

<i>q, work</i>	REAL for slaed7 DOUBLE PRECISION for dlaed7 COMPLEX for claed7 COMPLEX*16 for zlaed7. Arrays: <i>q(ldq, *)</i> contains the the eigenvectors of the rank-1-perturbed matrix. The second dimension of <i>q</i> must be at least $\max(1, n)$. <i>work(*)</i> is a workspace array, dimension at least $(3n+qsiz*n)$ for real flavors and at least $(qsiz*n)$ for complex flavors.
<i>ldq</i>	INTEGER. The first dimension of the array <i>q</i> ; $ldq \geq \max(1, n)$.
<i>indxq</i>	INTEGER. Array, dimension (<i>n</i>). Contains the permutation that separately sorts the two sub-problems in <i>d</i> into ascending order.
<i>rho</i>	REAL for slaed7 /claed7 DOUBLE PRECISION for dlaed7/zlaed7. The subdiagonal element used to create the rank-1 modification.
<i>qstore</i>	REAL for slaed7/claed7 DOUBLE PRECISION for dlaed7/zlaed7. Array, dimension (n^2+1). Serves also as output parameter. Stores eigenvectors of submatrices encountered during divide and conquer, packed together. <i>qp</i><i>tr</i> points to beginning of the submatrices.
<i>qp</i> <i>tr</i>	INTEGER. Array, dimension $(n+2)$. Serves also as output parameter. List of indices pointing to beginning of submatrices stored in <i>qstore</i>. The submatrices are numbered starting at the bottom left of the divide and conquer tree, from left to right and bottom to top.
<i>prmp</i> <i>tr</i> , <i>perm</i> , <i>givp</i> <i>tr</i>	INTEGER. Arrays, dimension $(n \lg n)$ each. The array <i>prmp</i><i>tr</i>(*) contains a list of pointers which indicate where in <i>perm</i> a level's permutation is stored. <i>prmp</i><i>tr</i>(<i>i</i>+1) - <i>prmp</i><i>tr</i>(<i>i</i>) indicates the size of the permutation and also the size of the full, non-deflated problem.

The array *perm*(*) contains the permutations (from deflation and sorting) to be applied to each eigenblock. This parameter can be modified by *?laed8*, where it is output. The array *givptr*(*) contains a list of pointers which indicate where in *givcol* a level's Givens rotations are stored. *givptr*(i+1) - *givptr*(i) indicates the number of Givens rotations.

<i>givcol</i>	INTEGER. Array, dimension (2, <i>n lg n</i>). Each pair of numbers indicates a pair of columns to take place in a Givens rotation.
<i>givnum</i>	REAL for <i>slaed7/claed7</i> DOUBLE PRECISION for <i>dlaed7/zlaed7</i> . Array, dimension (2, <i>n lg n</i>). Each number indicates the <i>s</i> value to be used in the corresponding Givens rotation.
<i>iwork</i>	INTEGER. Workspace array, dimension (4 <i>n</i>).
<i>rwork</i>	REAL for <i>claed7</i> DOUBLE PRECISION for <i>zlaed7</i> . Workspace array, dimension (3 <i>n</i> +2 <i>qsiz</i> * <i>n</i>). Used in complex flavors only.

Output Parameters

<i>d</i>	On exit, contains the eigenvalues of the repaired matrix.
<i>q</i>	On exit, <i>q</i> contains the eigenvectors of the repaired tridiagonal matrix.
<i>indxq</i>	INTEGER. Array, dimension (<i>n</i>). Contains the permutation that reintegrates the subproblems back into a sorted order, that is, <i>d</i> (<i>indxq</i> (i = 1, <i>n</i>)) will be in the ascending order.
<i>rho</i>	This parameter can be modified by <i>?laed8</i> , where it is input/output.
<i>prmptr, perm, givptr</i>	INTEGER. Arrays, dimension (<i>n lg n</i>) each. The array <i>prmptr</i> contains an updated list of pointers. The array <i>perm</i> contains an updated permutation. The array <i>givptr</i> contains an updated list of pointers.

<i>givcol</i>	This parameter can be modified by <code>?laed8</code> , where it is output.
<i>givnum</i>	This parameter can be modified by <code>?laed8</code> , where it is output.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = 1, an eigenvalue did not converge.

?laed8

Used by ?stedc. Merges eigenvalues and deflates secular equation. Used when the original matrix is dense.

Syntax

```
call slaed8( icompg, k, n, qsiz, d, q, ldq, indxq, rho, cutpnt, z, dlamda,
q2, ldq2, w, perm, givptr, givcol, givnum, indxp, indx, info )
call dlaed8( icompg, k, n, qsiz, d, q, ldq, indxq, rho, cutpnt, z, dlamda,
q2, ldq2, w, perm, givptr, givcol, givnum, indxp, indx, info )
call claed8( k, n, qsiz, q, ldq, d, rho, cutpnt, z, dlamda, q2, ldq2, w, indxp,
indx, indxq, perm, givptr, givcol, givnum, info )
call zlaed8( k, n, qsiz, q, ldq, d, rho, cutpnt, z, dlamda, q2, ldq2, w, indxp,
indx, indxq, perm, givptr, givcol, givnum, info )
```

Description

This routine merges the two sets of eigenvalues together into a single sorted set. Then it tries to deflate the size of the problem. There are two ways in which deflation can occur: when two or more eigenvalues are close together or if there is a tiny element in the *z* vector. For each such occurrence the order of the related secular equation problem is reduced by one.

Input Parameters

<i>icompg</i>	INTEGER. Used with real flavors only. If <i>icompg</i> = 0, compute eigenvalues only. If <i>icompg</i> = 1, compute eigenvectors of original dense symmetric matrix also.

	On entry, the array q must contain the orthogonal matrix used to reduce the original matrix to tridiagonal form.
n	INTEGER. The dimension of the symmetric tridiagonal matrix ($n \geq 0$).
$cutpnt$	INTEGER. The location of the last eigenvalue in the leading sub-matrix. $\min(1, n) \leq cutpnt \leq n$.
$qsiz$	INTEGER. The dimension of the orthogonal/unitary matrix used to reduce the full matrix to tridiagonal form; $qsiz \geq n$ (for real flavors, $qsiz \geq n$ if $icompg = 1$).
d, z	REAL for slaed8/claed8 DOUBLE PRECISION for dlaed8/zlaed8. Arrays, dimension at least $\max(1, n)$ each. The array $d(*)$ contains the eigenvalues of the two submatrices to be combined. On entry, $z(*)$ contains the updating vector (the last row of the first sub-eigenvector matrix and the first row of the second sub-eigenvector matrix). The contents of z are destroyed by the updating process.
q	REAL for slaed8 DOUBLE PRECISION for dlaed8 COMPLEX for claed8 COMPLEX*16 for zlaed8. Array $q(ldq, *)$. The second dimension of q must be at least $\max(1, n)$. On entry, q contains the eigenvectors of the partially solved system which has been previously updated in matrix multiplies with other partially solved eigensystems. For real flavors, If $icompg = 0$, q is not referenced.
ldq	INTEGER. The first dimension of the array q ; $ldq \geq \max(1, n)$.
$ldq2$	INTEGER. The first dimension of the output array $q2$; $ldq2 \geq \max(1, n)$.
$indxq$	INTEGER. Array, dimension (n).

The permutation that separately sorts the two sub-problems in d into ascending order. Note that elements in the second half of this permutation must first have *cutpnt* added to their values in order to be accurate.

rho

REAL for slaed8/claed8

DOUBLE PRECISION for dlaed8/zlaed8.

On entry, the off-diagonal element associated with the rank-1 cut which originally split the two submatrices which are now being recombined.

Output Parameters

k

INTEGER. The number of non-deflated eigenvalues, and the order of the related secular equation.

d

On exit, contains the trailing $(n-k)$ updated eigenvalues (those which were deflated) sorted into increasing order.

z

On exit, the updating process destroys the contents of *z*.

q

On exit, *q* contains the trailing $(n-k)$ updated eigenvectors (those which were deflated) in its last $(n-k)$ columns.

indxq

INTEGER. Array, dimension (n) .

The permutation of merged eigenvalues set.

rho

On exit, *rho* has been modified to the value required by ?laed3.

dlamda, w

REAL for slaed8/claed8

DOUBLE PRECISION for dlaed8/zlaed8.

Arrays, dimension (n) each. The array *dlamda*(*) contains a copy of the first k eigenvalues which will be used by ?laed3 to form the secular equation.

The array *w*(*) will hold the first k values of the final deflation-altered *z*-vector and will be passed to ?laed3.

q2

REAL for slaed8

DOUBLE PRECISION for dlaed8

COMPLEX for claed8

COMPLEX*16 for zlaed8.

Array

q2(*ldq2*, *). The second dimension of *q2* must be at least $\max(1, n)$.

	Contains a copy of the first k eigenvectors which will be used by <code>slaed7/dlaed7</code> in a matrix multiply (<code>sgemm/dgemm</code>) to update the new eigenvectors. For real flavors, If <code>icompg</code> = 0, <code>q2</code> is not referenced.
<code>indxp, indx</code>	INTEGER. Workspace arrays, dimension (n) each. The array <code>indxp(*)</code> will contain the permutation used to place deflated values of d at the end of the array. On output, <code>indxp(1:k)</code> points to the nondeflated d -values and <code>indxp(k+1:n)</code> points to the deflated eigenvalues. The array <code>indx(*)</code> will contain the permutation used to sort the contents of d into ascending order.
<code>perm</code>	INTEGER. Array, dimension (n). Contains the permutations (from deflation and sorting) to be applied to each eigenblock.
<code>givptr</code>	INTEGER. Contains the number of Givens rotations which took place in this subproblem.
<code>givcol</code>	INTEGER. Array, dimension (2, n). Each pair of numbers indicates a pair of columns to take place in a Givens rotation.
<code>givnum</code>	REAL for <code>slaed8/claed8</code> DOUBLE PRECISION for <code>dlaed8/zlaed8</code> . Array, dimension (2, n). Each number indicates the s value to be used in the corresponding Givens rotation.
<code>info</code>	INTEGER. If <code>info</code> = 0, the execution is successful. If <code>info</code> = $-i$, the i -th parameter had an illegal value.

slaed9

Used by sstedc/dstedc. Finds the roots of the secular equation and updates the eigenvectors. Used when the original matrix is dense.

Syntax

```
call slaed9( k, kstart, kstop, n, d, q, ldq, rho, dlamda, w, s, lds, info )
call dlaed9( k, kstart, kstop, n, d, q, ldq, rho, dlamda, w, s, lds, info )
```

Description

This routine finds the roots of the secular equation, as defined by the values in d , z , and ρ , between $kstart$ and $kstop$. It makes the appropriate calls to slaed4/dlaed4 and then stores the new matrix of eigenvectors for use in calculating the next level of z vectors.

Input Parameters

k	INTEGER. The number of terms in the rational function to be solved by slaed4/dlaed4 ($k \geq 0$).
$kstart, kstop$	INTEGER. The updated eigenvalues $\lambda(i)$, $kstart \leq i \leq kstop$ are to be computed. $1 \leq kstart \leq kstop \leq k$.
n	INTEGER. The number of rows and columns in the Q matrix. $n \geq k$ (deflation may result in $n > k$).
q	REAL for slaed9 DOUBLE PRECISION for dlaed9. Workspace array, dimension ($ldq, *$). The second dimension of q must be at least $\max(1, n)$.
ldq	INTEGER. The first dimension of the array q ; $ldq \geq \max(1, n)$.
ρ	REAL for slaed9 DOUBLE PRECISION for dlaed9 The value of the parameter in the rank one update equation. $\rho \geq 0$ required.
$dlamda, w$	REAL for slaed9

DOUBLE PRECISION for dlaed9

Arrays, dimension (k) each. The first k elements of the array $d\lambda mda(*)$ contain the old roots of the deflated updating problem. These are the poles of the secular equation. The first k elements of the array $w(*)$ contain the components of the deflation-adjusted updating vector.

lds

INTEGER. The first dimension of the output array s ; $lds \geq \max(1, k)$.

Output Parameters

d

REAL for slaed9

DOUBLE PRECISION for dlaed9

Array, dimension (n). Elements in $d(i)$ are not referenced for $1 \leq i < kstart$ or $kstop < i \leq n$.

s

REAL for slaed9

DOUBLE PRECISION for dlaed9.

Array, dimension ($lds, *$).

The second dimension of s must be at least $\max(1, k)$. Will contain the eigenvectors of the repaired matrix which will be stored for subsequent z vector calculation and multiplied by the previously accumulated eigenvectors to update the system.

$d\lambda mda$

On exit, the value is modified to make sure all $d\lambda mda(i) - d\lambda mda(j)$ can be computed with high relative accuracy, barring overflow and underflow.

w

Destroyed on exit.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value. If $info = 1$, the eigenvalue did not converge.

?laeda

Used by ?stedc. Computes the Z vector determining the rank-one modification of the diagonal matrix. Used when the original matrix is dense.

Syntax

```
call slaeda( n, tlvl, curlvl, curpbm, prmptr, perm, givptr, givcol, givnum,
q, qptr, z, ztemp, info )
call dlaeda( n, tlvl, curlvl, curpbm, prmptr, perm, givptr, givcol, givnum,
q, qptr, z, ztemp, info )
```

Description

The routine ?laeda computes the Z vector corresponding to the merge step in the *curlvl*-th step of the merge process with *tlvl* steps for the *curpbm*-th problem.

Input Parameters

<i>n</i>	INTEGER. The dimension of the symmetric tridiagonal matrix ($n \geq 0$).
<i>tlvl</i>	INTEGER. The total number of merging levels in the overall divide and conquer tree.
<i>curlvl</i>	INTEGER. The current level in the overall merge routine, $0 \leq \text{curlvl} \leq \text{tlvl}$.
<i>curpbm</i>	INTEGER. The current problem in the current level in the overall merge routine (counting from upper left to lower right).
<i>prmptr</i> , <i>perm</i> , <i>givptr</i>	INTEGER. Arrays, dimension $(n \lg n)$ each. The array <i>prmptr</i> (*) contains a list of pointers which indicate where in <i>perm</i> a level's permutation is stored. <i>prmptr</i> (<i>i</i> +1) - <i>prmptr</i> (<i>i</i>) indicates the size of the permutation and also the size of the full, non-deflated problem. The array <i>perm</i> (*) contains the permutations (from deflation and sorting) to be applied to each eigenblock.

	The array <i>givptr</i>(*) contains a list of pointers which indicate where in <i>givcol</i> a level's Givens rotations are stored. <i>givptr</i>(i+1) - <i>givptr</i>(i) indicates the number of Givens rotations.
<i>givcol</i>	INTEGER. Array, dimension (2, <i>n lg n</i>). Each pair of numbers indicates a pair of columns to take place in a Givens rotation.
<i>givnum</i>	REAL for slaeda DOUBLE PRECISION for dlaeda. Array, dimension (2, <i>n lg n</i>). Each number indicates the <i>s</i> value to be used in the corresponding Givens rotation.
<i>q</i>	REAL for slaeda DOUBLE PRECISION for dlaeda. Array, dimension (<i>n</i>²). Contains the square eigenblocks from previous levels, the starting positions for blocks are given by <i>qptr</i>.
<i>qptr</i>	INTEGER. Array, dimension (<i>n</i>+2). Contains a list of pointers which indicate where in <i>q</i> an eigenblock is stored. <i>sqrt</i>(<i>qptr</i>(i+1) - <i>qptr</i>(i)) indicates the size of the block.
<i>ztemp</i>	REAL for slaeda DOUBLE PRECISION for dlaeda. Workspace array, dimension (<i>n</i>).

Output Parameters

<i>z</i>	REAL for slaeda DOUBLE PRECISION for dlaeda. Array, dimension (<i>n</i>). Contains the updating vector (the last row of the first sub-eigenvector matrix and the first row of the second sub-eigenvector matrix).
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.

?laein

Computes a specified right or left eigenvector of an upper Hessenberg matrix by inverse iteration.

Syntax

```
call slaein( rightv, noinit, n, h, ldh, wr, wi, vr, vi, b, ldb, work, eps3,
smlnum, bignum, info )

call dlaein( rightv, noinit, n, h, ldh, wr, wi, vr, vi, b, ldb, work, eps3,
smlnum, bignum, info )

call claein( rightv, noinit, n, h, ldh, w, v, b, ldb, rwork, eps3, smlnum,
info )

call zlaein( rightv, noinit, n, h, ldh, w, v, b, ldb, rwork, eps3, smlnum,
info )
```

Description

The routine ?laein uses inverse iteration to find a right or left eigenvector corresponding to the eigenvalue (wr, wi) of a real upper Hessenberg matrix H (for real flavors slaein/dlaein) or to the eigenvalue w of a complex upper Hessenberg matrix H (for complex flavors claein/zlaein).

Input Parameters

<i>rightv</i>	LOGICAL. If <i>rightv</i> = .TRUE., compute right eigenvector; if <i>rightv</i> = .FALSE., compute left eigenvector.
<i>noinit</i>	LOGICAL. If <i>noinit</i> = .TRUE., no initial vector is supplied in (<i>vr</i> , <i>vi</i>) or in <i>v</i> (for complex flavors); if <i>noinit</i> = .FALSE., initial vector is supplied in (<i>vr</i> , <i>vi</i>) or in <i>v</i> (for complex flavors).
<i>n</i>	INTEGER. The order of the matrix H ($n \geq 0$).
<i>h</i>	REAL for slaein DOUBLE PRECISION for dlaein COMPLEX for claein COMPLEX*16 for zlaein.

	Array $h(ldh, *)$. The second dimension of h must be at least $\max(1, n)$. Contains the upper Hessenberg matrix H.
ldh	INTEGER. The first dimension of the array h; $ldh \geq \max(1, n)$.
wr, wi	REAL for <code>slaein</code> DOUBLE PRECISION for <code>dlaein</code>. The real and imaginary parts of the eigenvalue of H whose corresponding right or left eigenvector is to be computed (for real flavors of the routine).
w	COMPLEX for <code>claein</code> COMPLEX*16 for <code>zlaein</code>. The eigenvalue of H whose corresponding right or left eigenvector is to be computed (for complex flavors of the routine).
vr, vi	REAL for <code>slaein</code> DOUBLE PRECISION for <code>dlaein</code>. Arrays, dimension (n) each. Used for real flavors only. On entry, if <code>noinit</code> = <code>.FALSE.</code> and $wi = 0.0$, vr must contain a real starting vector for inverse iteration using the real eigenvalue wr; if <code>noinit</code> = <code>.FALSE.</code> and $wi \neq 0.0$, vr and vi must contain the real and imaginary parts of a complex starting vector for inverse iteration using the complex eigenvalue (wr, wi); otherwise vr and vi need not be set.
v	COMPLEX for <code>claein</code> COMPLEX*16 for <code>zlaein</code>. Array, dimension (n). Used for complex flavors only. On entry, if <code>noinit</code> = <code>.FALSE.</code>, v must contain a starting vector for inverse iteration; otherwise v need not be set.
b	REAL for <code>slaein</code> DOUBLE PRECISION for <code>dlaein</code> COMPLEX for <code>claein</code> COMPLEX*16 for <code>zlaein</code>. Workspace array $b(l db, *)$. The second dimension of b must be at least $\max(1, n)$.

<i>ldb</i>	INTEGER. The first dimension of the array <i>b</i>; $ldb \geq n+1$ for real flavors; $ldb \geq \max(1, n)$ for complex flavors.
<i>work</i>	REAL for slaein DOUBLE PRECISION for dlaein. Workspace array, dimension (<i>n</i>). Used for real flavors only.
<i>rwork</i>	REAL for claein DOUBLE PRECISION for zlaein. Workspace array, dimension (<i>n</i>). Used for complex flavors only.
<i>eps3, smlnum</i>	REAL for slaein/claein DOUBLE PRECISION for dlaein/zlaein. <i>eps3</i> is a small machine-dependent value which is used to perturb close eigenvalues, and to replace zero pivots. <i>smlnum</i> is a machine-dependent value close to underflow threshold.
<i>bignum</i>	REAL for slaein DOUBLE PRECISION for dlaein. <i>bignum</i> is a machine-dependent value close to overflow threshold. Used for real flavors only.

Output Parameters

<i>vr, vi</i>	On exit, if $w_i = 0.0$ (real eigenvalue), <i>vr</i> contains the computed real eigenvector; if $w_i \neq 0.0$ (complex eigenvalue), <i>vr</i> and <i>vi</i> contain the real and imaginary parts of the computed complex eigenvector. The eigenvector is normalized so that the component of largest magnitude has magnitude 1; here the magnitude of a complex number (<i>x,y</i>) is taken to be $x + y$. <i>vi</i> is not referenced if $w_i = 0.0$.
<i>v</i>	On exit, <i>v</i> contains the computed eigenvector, normalized so that the component of largest magnitude has magnitude 1; here the magnitude of a complex number (<i>x,y</i>) is taken to be $x + y$.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = 1, inverse iteration did not converge. For real flavors, *vr* is set to the last iterate, and so is *vi*, if *wi* ≠ 0.0. For complex flavors, *v* is set to the last iterate.

?laev2

Computes the eigenvalues and eigenvectors of a 2-by-2 symmetric/Hermitian matrix.

Syntax

call slaev2(*a*, *b*, *c*, *rt1*, *rt2*, *cs1*, *sn1*)

call dlaev2(*a*, *b*, *c*, *rt1*, *rt2*, *cs1*, *sn1*)

call claev2(*a*, *b*, *c*, *rt1*, *rt2*, *cs1*, *sn1*)

call zlaev2(*a*, *b*, *c*, *rt1*, *rt2*, *cs1*, *sn1*)

Description

This routine performs the eigendecomposition of a 2-by-2 symmetric matrix

$$\begin{bmatrix} a & b \\ b & c \end{bmatrix} \text{ (for slaev2/dlaev2) or Hermitian matrix } \begin{bmatrix} a & b \\ \text{conjg}(b) & c \end{bmatrix}$$

(for claev2/zlaev2).

On return, *rt1* is the eigenvalue of larger absolute value, *rt2* of smaller absolute value, and (*cs1*, *sn1*) is the unit right eigenvector for *rt1*, giving the decomposition

$$\begin{bmatrix} cs1 & sn1 \\ -sn1 & cs1 \end{bmatrix} \cdot \begin{bmatrix} a & b \\ b & c \end{bmatrix} \cdot \begin{bmatrix} cs1 & sn1 \\ -sn1 & cs1 \end{bmatrix} = \begin{bmatrix} rt1 & 0 \\ 0 & rt2 \end{bmatrix}$$

(for slaev2/dlaev2),

or

$$\begin{bmatrix} cs1 & \text{conjg}(sn1) \\ -sn1 & cs1 \end{bmatrix} \cdot \begin{bmatrix} a & b \\ \text{conjg}(b) & c \end{bmatrix} \cdot \begin{bmatrix} cs1 & -\text{conjg}(sn1) \\ sn1 & cs1 \end{bmatrix} = \begin{bmatrix} rt1 & 0 \\ 0 & rt2 \end{bmatrix}$$

(for `claev2/zlaev2`).

Input Parameters

a, b, c REAL for `slaev2`
 DOUBLE PRECISION for `dlaev2`
 COMPLEX for `claev2`
 COMPLEX*16 for `zlaev2`.
 Elements of the input matrix.

Output Parameters

rt1, rt2 REAL for `slaev2/claev2`
 DOUBLE PRECISION for `dlaev2/zlaev2`.
 Eigenvalues of larger and smaller absolute value,
 respectively.

cs1 REAL for `slaev2/claev2`
 DOUBLE PRECISION for `dlaev2/zlaev2`.

sn1 REAL for `slaev2`
 DOUBLE PRECISION for `dlaev2`
 COMPLEX for `claev2`
 COMPLEX*16 for `zlaev2`.
 The vector (*cs1, sn1*) is the unit right eigenvector for *rt1*.

Application Notes

rt1 is accurate to a few ulps barring over/underflow. *rt2* may be inaccurate if there is massive cancellation in the determinant $a*c-b*b$; higher precision or correctly rounded or correctly truncated arithmetic would be needed to compute *rt2* accurately in all cases. *cs1* and *sn1* are accurate to a few ulps barring over/underflow. Overflow is possible only if *rt1* is within a factor of 5 of overflow. Underflow is harmless if the input data is 0 or exceeds *underflow_threshold* / *macheps*.

?laexc

Swaps adjacent diagonal blocks of a real upper quasi-triangular matrix in Schur canonical form, by an orthogonal similarity transformation.

Syntax

```
call slaexc( wantq, n, t, ldt, q, ldq, j1, n1, n2, work, info )
call dlaexc( wantq, n, t, ldt, q, ldq, j1, n1, n2, work, info )
```

Description

This routine swaps adjacent diagonal blocks T_{11} and T_{22} of order 1 or 2 in an upper quasi-triangular matrix T by an orthogonal similarity transformation.

T must be in Schur canonical form, that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks; each 2-by-2 diagonal block has its diagonal elements equal and its off-diagonal elements of opposite sign.

Input Parameters

<i>wantq</i>	LOGICAL. If <i>wantq</i> = .TRUE., accumulate the transformation in the matrix Q ; If <i>wantq</i> = .FALSE., do not accumulate the transformation.
<i>n</i>	INTEGER. The order of the matrix T ($n \geq 0$).
<i>t, q</i>	REAL for slaexc DOUBLE PRECISION for dlaexc Arrays: <i>t</i> (<i>ldt</i> ,*) contains on entry the upper quasi-triangular matrix T , in Schur canonical form. The second dimension of <i>t</i> must be at least $\max(1, n)$. <i>q</i> (<i>ldq</i> ,*) contains on entry, if <i>wantq</i> = .TRUE., the orthogonal matrix Q . If <i>wantq</i> = .FALSE., <i>q</i> is not referenced. The second dimension of <i>q</i> must be at least $\max(1, n)$.
<i>ldt</i>	INTEGER. The first dimension of <i>t</i> ; at least $\max(1, n)$.
<i>ldq</i>	INTEGER. The first dimension of <i>q</i> ;

If `wantq = .FALSE.`, then $ldq \geq 1$.
 If `wantq = .TRUE.`, then $ldq \geq \max(1, n)$.

`j1` INTEGER. The index of the first row of the first block T_{11} .

`n1` INTEGER. The order of the first block T_{11}
 ($n1 = 0, 1$, or 2).

`n2` INTEGER. The order of the second block T_{22}
 ($n2 = 0, 1$, or 2).

`work` REAL for slaexc;
 DOUBLE PRECISION for dlaexc.
 Workspace array, DIMENSION (n).

Output Parameters

`t` On exit, the updated matrix T , again in Schur canonical form.

`q` On exit, if `wantq = .TRUE.`, the updated matrix Q .

`info` INTEGER.
 If `info = 0`, the execution is successful.
 If `info = 1`, the transformed matrix T would be too far from Schur form; the blocks are not swapped and T and Q are unchanged.

?lag2

Computes the eigenvalues of a 2-by-2 generalized eigenvalue problem, with scaling as necessary to avoid over-/underflow.

Syntax

```
call slag2( a, lda, b, ldb, safmin, scale1, scale2, wr1, wr2, wi )
call dlag2( a, lda, b, ldb, safmin, scale1, scale2, wr1, wr2, wi )
```

Description

This routine computes the eigenvalues of a 2×2 generalized eigenvalue problem $A - w * B$, with scaling as necessary to avoid over-/underflow. The scaling factor, s , results in a modified eigenvalue equation

$$s^*A - w^*B,$$

where s is a non-negative scaling factor chosen so that w , w^*B , and s^*A do not overflow and, if possible, do not underflow, either.

Input Parameters

a, b	REAL for <code>slag2</code> DOUBLE PRECISION for <code>dlag2</code> Arrays: $a(lda, 2)$ contains, on entry, the 2 x 2 matrix A. It is assumed that its 1-norm is less than $1/safmin$. Entries less than $\text{sqrt}(safmin) * \text{norm}(A)$ are subject to being treated as zero. $b(l db, 2)$ contains, on entry, the 2 x 2 upper triangular matrix B. It is assumed that the one-norm of B is less than $1/safmin$. The diagonals should be at least $\text{sqrt}(safmin)$ times the largest element of B (in absolute value); if a diagonal is smaller than that, then $\pm \text{sqrt}(safmin)$ will be used instead of that diagonal.
lda	INTEGER. The first dimension of a ; $lda \geq 2$.
ldb	INTEGER. The first dimension of b ; $ldb \geq 2$.
$safmin$	REAL for <code>slag2</code>; DOUBLE PRECISION for <code>dlag2</code>. The smallest positive number such that $1/safmin$ does not overflow. (This should always be <code>?lamch('S')</code> - it is an argument in order to avoid having to call <code>?lamch</code> frequently.)

Output Parameters

$scale1$	REAL for <code>slag2</code>; DOUBLE PRECISION for <code>dlag2</code>. A scaling factor used to avoid over-/underflow in the eigenvalue equation which defines the first eigenvalue. If the eigenvalues are complex, then the eigenvalues are $(wr1 \pm wii)/scale1$ (which may lie outside the exponent range of the machine), $scale1 = scale2$, and $scale1$ will always be positive.

If the eigenvalues are real, then the first (real) eigenvalue is $wr1/scale1$, but this may overflow or underflow, and in fact, $scale1$ may be zero or less than the underflow threshold if the exact eigenvalue is sufficiently large.

scale2

REAL for *slag2*;

DOUBLE PRECISION for *dlag2*.

A scaling factor used to avoid over-/underflow in the eigenvalue equation which defines the second eigenvalue. If the eigenvalues are complex, then $scale2=scale1$. If the eigenvalues are real, then the second (real) eigenvalue is $wr2/scale2$, but this may overflow or underflow, and in fact, $scale2$ may be zero or less than the underflow threshold if the exact eigenvalue is sufficiently large.

wr1

REAL for *slag2*;

DOUBLE PRECISION for *dlag2*.

If the eigenvalue is real, then $wr1$ is $scale1$ times the eigenvalue closest to the (2,2) element of $A \cdot \text{inv}(B)$. If the eigenvalue is complex, then $wr1=wr2$ is $scale1$ times the real part of the eigenvalues.

wr2

REAL for *slag2*;

DOUBLE PRECISION for *dlag2*.

If the eigenvalue is real, then $wr2$ is $scale2$ times the other eigenvalue. If the eigenvalue is complex, then $wr1=wr2$ is $scale1$ times the real part of the eigenvalues.

wi

REAL for *slag2*;

DOUBLE PRECISION for *dlag2*.

If the eigenvalue is real, then wi is zero. If the eigenvalue is complex, then wi is $scale1$ times the imaginary part of the eigenvalues. wi will always be non-negative.

slags2

Computes 2-by-2 orthogonal matrices U , V , and Q , and applies them to matrices A and B such that the rows of the transformed A and B are parallel.

Syntax

```
call slags2( upper, a1, a2, a3, b1, b2, b3, csu, snu, csv, snv, csq, snq )
call dlags2( upper, a1, a2, a3, b1, b2, b3, csu, snu, csv, snv, csq, snq )
```

Description

This routine computes 2-by-2 orthogonal matrices U , V and Q , such that if $upper = .TRUE.$, then

$$U^* A Q = U^* \begin{bmatrix} A_1 & A_2 \\ 0 & A_3 \end{bmatrix} * Q = \begin{bmatrix} x & 0 \\ x & x \end{bmatrix}$$

and

$$V^* B Q = V^* \begin{bmatrix} B_1 & B_2 \\ 0 & B_3 \end{bmatrix} * Q = \begin{bmatrix} x & 0 \\ x & x \end{bmatrix}$$

or if $upper = .FALSE.$, then

$$U^* A Q = U^* \begin{bmatrix} A_1 & 0 \\ A_2 & A_3 \end{bmatrix} * Q = \begin{bmatrix} x & x \\ 0 & x \end{bmatrix}$$

and

$$V^* B Q = U^* \begin{bmatrix} B_1 & 0 \\ B_2 & B_3 \end{bmatrix} Q = \begin{bmatrix} x & x \\ 0 & x \end{bmatrix}$$

The rows of the transformed A and B are parallel, where

$$U = \begin{bmatrix} csu & snu \\ -snu & csu \end{bmatrix}, V = \begin{bmatrix} csv & snv \\ -snv & csv \end{bmatrix}, Q = \begin{bmatrix} csq & snq \\ -snq & csq \end{bmatrix}$$

Here Z' denotes the transpose of Z .

Input Parameters

<i>upper</i>	LOGICAL. If <i>upper</i> = .TRUE., the input matrices A and B are upper triangular; If <i>upper</i> = .FALSE., the input matrices A and B are lower triangular.
<i>a1, a2, a3</i>	REAL for slags2 DOUBLE PRECISION for dlags2 On entry, <i>a1</i> , <i>a2</i> and <i>a3</i> are elements of the input 2-by-2 upper (lower) triangular matrix A .
<i>b1, b2, b3</i>	REAL for slags2 DOUBLE PRECISION for dlags2 On entry, <i>b1</i> , <i>b2</i> and <i>b3</i> are elements of the input 2-by-2 upper (lower) triangular matrix B .

Output Parameters

<i>csu, snu</i>	REAL for slags2 DOUBLE PRECISION for dlags2 The desired orthogonal matrix U .
<i>csv, snv</i>	REAL for slags2 DOUBLE PRECISION for dlags2 The desired orthogonal matrix V .
<i>csq, snq</i>	REAL for slags2 DOUBLE PRECISION for dlags2

The desired orthogonal matrix Q .

?lagtf

Computes an LU factorization of a matrix $T - \lambda * I$, where T is a general tridiagonal matrix, and λ is a scalar, using partial pivoting with row interchanges.

Syntax

```
call slagtf( n, a, lambda, b, c, tol, d, in, info )
call dlagtf( n, a, lambda, b, c, tol, d, in, info )
```

Description

This routine factorizes the matrix $(T - \lambda * I)$, where T is an n -by- n tridiagonal matrix and λ is a scalar, as

$$T - \lambda * I = P * L * U,$$

where P is a permutation matrix, L is a unit lower tridiagonal matrix with at most one non-zero sub-diagonal elements per column and U is an upper triangular matrix with at most two non-zero super-diagonal elements per column. The factorization is obtained by Gaussian elimination with partial pivoting and implicit row scaling. The parameter λ is included in the routine so that ?lagtf may be used, in conjunction with ?lagts, to obtain eigenvectors of T by inverse iteration.

Input Parameters

n	INTEGER. The order of the matrix T ($n \geq 0$).
a, b, c	REAL for slagtf DOUBLE PRECISION for dlagtf Arrays, dimension $a(n), b(n-1), c(n-1)$: On entry, $a(*)$ must contain the diagonal elements of the matrix T . On entry, $b(*)$ must contain the $(n-1)$ super-diagonal elements of T . On entry, $c(*)$ must contain the $(n-1)$ sub-diagonal elements of T .
tol	REAL for slagtf

DOUBLE PRECISION for `dlagtf`

On entry, a relative tolerance used to indicate whether or not the matrix $(T - \lambda I)$ is nearly singular. `tol` should normally be chosen as approximately the largest relative error in the elements of T . For example, if the elements of T are correct to about 4 significant figures, then `tol` should be set to about 5×10^{-4} . If `tol` is supplied as less than `eps`, where `eps` is the relative machine precision, then the value `eps` is used in place of `tol`.

Output Parameters

<i>a</i>	On exit, <i>a</i> is overwritten by the <i>n</i> diagonal elements of the upper triangular matrix <i>U</i> of the factorization of <i>T</i> .
<i>b</i>	On exit, <i>b</i> is overwritten by the <i>n</i> -1 super-diagonal elements of the matrix <i>U</i> of the factorization of <i>T</i> .
<i>c</i>	On exit, <i>c</i> is overwritten by the <i>n</i> -1 sub-diagonal elements of the matrix <i>L</i> of the factorization of <i>T</i> .
<i>d</i>	REAL for <code>slagtf</code> DOUBLE PRECISION for <code>dlagtf</code> Array, dimension (<i>n</i> -2). On exit, <i>d</i> is overwritten by the <i>n</i> -2 second super-diagonal elements of the matrix <i>U</i> of the factorization of <i>T</i> .
<i>in</i>	INTEGER. Array, dimension (<i>n</i>). On exit, <i>in</i> contains details of the permutation matrix <i>p</i> . If an interchange occurred at the <i>k</i> -th step of the elimination, then <i>in</i> (<i>k</i>) = 1, otherwise <i>in</i> (<i>k</i>) = 0. The element <i>in</i> (<i>n</i>) returns the smallest positive integer <i>j</i> such that $\text{abs}(u(j,j)) \leq \text{norm}((T - \lambda I)(j)) * \text{tol},$ where $\text{norm}(A(j))$ denotes the sum of the absolute values of the <i>j</i> -th row of the matrix <i>A</i> . If no such <i>j</i> exists then <i>in</i> (<i>n</i>) is returned as zero. If <i>in</i> (<i>n</i>) is returned as positive, then a diagonal element of <i>U</i> is small, indicating that $(T - \lambda I)$ is singular or nearly singular.
<i>info</i>	INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*k*, the *k*-th parameter had an illegal value.

?lagtm

Performs a matrix-matrix product of the form $C = \alpha * A * B + \beta * C$, where A is a tridiagonal matrix, B and C are rectangular matrices, and α and β are scalars, which may be 0, 1, or -1.

Syntax

```
call slagtm( trans, n, nrhs, alpha, dl, d, du, x, ldx, beta, b, ldb )
call dlagtm( trans, n, nrhs, alpha, dl, d, du, x, ldx, beta, b, ldb )
call clagtm( trans, n, nrhs, alpha, dl, d, du, x, ldx, beta, b, ldb )
call zlagtm( trans, n, nrhs, alpha, dl, d, du, x, ldx, beta, b, ldb )
```

Description

This routine performs a matrix-vector product of the form:

$$B := \alpha * A * X + \beta * B$$

where A is a tridiagonal matrix of order n , B and X are n -by- $nrhs$ matrices, and α and β are real scalars, each of which may be 0., 1., or -1.

Input Parameters

<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', then $B := \alpha * A * X + \beta * B$ (no transpose); If <i>trans</i> = 'T', then $B := \alpha * A^T * X + \beta * B$ (transpose); If <i>trans</i> = 'C', then $B := \alpha * A^H * X + \beta * B$ (conjugate transpose)
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).

<i>nrhs</i>	INTEGER. The number of right-hand sides, i.e., the number of columns in X and B ($nrhs \geq 0$).
<i>alpha, beta</i>	REAL for <code>slagtm/clagtm</code> DOUBLE PRECISION for <code>dlagtm/zlagtm</code> Specify the scalars <i>alpha</i> and <i>beta</i> respectively. <i>alpha</i> must be 0., 1., or -1.; otherwise, it is assumed to be 0. <i>beta</i> must be 0., 1., or -1.; otherwise, it is assumed to be 1.
<i>dl, d, du</i>	REAL for <code>slagtm</code> DOUBLE PRECISION for <code>dlagtm</code> COMPLEX for <code>clagtm</code> COMPLEX*16 for <code>zlagtm</code> . Arrays: $dl(n-1)$, $d(n)$, $du(n-1)$. The array <i>dl</i> contains the $(n-1)$ sub-diagonal elements of T . The array <i>d</i> contains the n diagonal elements of T . The array <i>du</i> contains the $(n-1)$ super-diagonal elements of T .
<i>x, b</i>	REAL for <code>slagtm</code> DOUBLE PRECISION for <code>dlagtm</code> COMPLEX for <code>clagtm</code> COMPLEX*16 for <code>zlagtm</code> . Arrays: $x(ldx,*)$ contains the n -by- $nrhs$ matrix X . The second dimension of x must be at least $\max(1, nrhs)$. $b(l db,*)$ contains the n -by- $nrhs$ matrix B . The second dimension of b must be at least $\max(1, nrhs)$.
<i>ldx</i>	INTEGER. The leading dimension of the array x ; $ldx \geq \max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of the array b ; $ldb \geq \max(1, n)$.

Output Parameters

<i>b</i>	Overwritten by the matrix expression $B := \alpha A^* X + \beta B$

?lagts

Solves the system of equations $(T - \lambda I)x = y$ or $(T - \lambda I)^T x = y$, where T is a general tridiagonal matrix and λ is a scalar, using the LU factorization computed by ?lagtf.

Syntax

```
call slagts( job, n, a, b, c, d, in, y, tol, info )
call dlagts( job, n, a, b, c, d, in, y, tol, info )
```

Description

This routine may be used to solve for x one of the systems of equations:

$$(T - \lambda I)x = y \quad \text{or} \quad (T - \lambda I)^T x = y,$$

where T is an n -by- n tridiagonal matrix, following the factorization of $(T - \lambda I)$ as

$$T - \lambda I = P * L * U,$$

computed by the routine ?lagtf.

The choice of equation to be solved is controlled by the argument *job*, and in each case there is an option to perturb zero or very small diagonal elements of U , this option being intended for use in applications such as inverse iteration.

Input Parameters

<i>job</i>	INTEGER. Specifies the job to be performed by ?lagts as follows: = 1: The equations $(T - \lambda I)x = y$ are to be solved, but diagonal elements of U are not to be perturbed. = -1: The equations $(T - \lambda I)x = y$ are to be solved and, if overflow would otherwise occur, the diagonal elements of U are to be perturbed. See argument <i>tol</i> below. = 2: The equations $(T - \lambda I)^T x = y$ are to be solved, but diagonal elements of U are not to be perturbed. = -2: The equations $(T - \lambda I)^T x = y$ are to be solved and, if overflow would otherwise occur, the diagonal elements of U are to be perturbed. See argument <i>tol</i> below.

n INTEGER. The order of the matrix T ($n \geq 0$).

a, b, c, d REAL for `slagts`
DOUBLE PRECISION for `dlagts`
Arrays, dimension $a(n)$, $b(n-1)$, $c(n-1)$, $d(n-2)$:
On entry, $a(*)$ must contain the diagonal elements of U as returned from `?lagtf`.
On entry, $b(*)$ must contain the first super-diagonal elements of U as returned from `?lagtf`.
On entry, $c(*)$ must contain the sub-diagonal elements of L as returned from `?lagtf`.
On entry, $d(*)$ must contain the second super-diagonal elements of U as returned from `?lagtf`.

in INTEGER.
Array, dimension (n) .
On entry, $in(*)$ must contain details of the matrix p as returned from `?lagtf`.

y REAL for `slagts`
DOUBLE PRECISION for `dlagts`
Array, dimension (n) . On entry, the right hand side vector y .

tol REAL for `slagtf`
DOUBLE PRECISION for `dlagtf`.
On entry, with $job < 0$, *tol* should be the minimum perturbation to be made to very small diagonal elements of U . *tol* should normally be chosen as about $eps * \text{norm}(U)$, where eps is the relative machine precision, but if *tol* is supplied as non-positive, then it is reset to $eps * \max(|u(i,j)|)$. If $job > 0$ then *tol* is not referenced.

Output Parameters

y On exit, y is overwritten by the solution vector x .

tol On exit, *tol* is changed as described in *Input Parameters* section above, only if *tol* is non-positive on entry. Otherwise *tol* is unchanged.

info INTEGER.
If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value. If $info = i > 0$, overflow would occur when computing the i th element of the solution vector x . This can only occur when job is supplied as positive and either means that a diagonal element of U is very small, or that the elements of the right-hand side vector y are very large.

?lagv2

Computes the Generalized Schur factorization of a real 2-by-2 matrix pencil (A,B) where B is upper triangular.

Syntax

```
call slagv2( a, lda, b, ldb, alphas, alphai, beta, cs1, sn1, csr, snr )
call dlagv2( a, lda, b, ldb, alphas, alphai, beta, cs1, sn1, csr, snr )
```

Description

This routine computes the Generalized Schur factorization of a real 2-by-2 matrix pencil (A,B) where B is upper triangular. The routine computes orthogonal (rotation) matrices given by $cs1$, $sn1$ and csr , snr such that:

- 1) if the pencil (A,B) has two real eigenvalues (include 0/0 or 1/0 types), then

$$\begin{bmatrix} a_{11} & a_{12} \\ 0 & a_{22} \end{bmatrix} = \begin{bmatrix} cs1 & sn1 \\ -sn1 & cs1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} csr & -snr \\ snr & csr \end{bmatrix}$$

$$\begin{bmatrix} b_{11} & b_{12} \\ 0 & b_{22} \end{bmatrix} = \begin{bmatrix} cs1 & sn1 \\ -sn1 & cs1 \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ 0 & b_{22} \end{bmatrix} \begin{bmatrix} csr & -snr \\ snr & csr \end{bmatrix}$$

- 2) if the pencil (A,B) has a pair of complex conjugate eigenvalues, then

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} cs1 & sn1 \\ -sn1 & cs1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} csr & -snr \\ snr & csr \end{bmatrix}$$

$$\begin{bmatrix} b_{11} & 0 \\ 0 & b_{22} \end{bmatrix} = \begin{bmatrix} cs1 & sn1 \\ -sn1 & cs1 \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ 0 & b_{22} \end{bmatrix} \begin{bmatrix} csr & -snr \\ snr & csr \end{bmatrix}$$

where $b_{11} \geq b_{22} > 0$.

Input Parameters

a, b REAL for `slagv2`
DOUBLE PRECISION for `dlagv2`
Arrays:
a(l da,2) contains the 2-by-2 matrix *A*;
b(l db,2) contains the upper triangular 2-by-2 matrix *B*.
lda INTEGER. The leading dimension of the array *a*;
lda ≥ 2.
ldb INTEGER. The leading dimension of the array *b*;
ldb ≥ 2.

Output Parameters

a On exit, *a* is overwritten by the "A-part" of the generalized Schur form.
b On exit, *b* is overwritten by the "B-part" of the generalized Schur form.
alphan, alphai, beta REAL for `slagv2`
DOUBLE PRECISION for `dlagv2`.
Arrays, dimension (2) each.
(*alphan(k) + i*alphai(k)*)/*beta(k)* are the eigenvalues of the pencil (*A,B*), *k*=1,2 and *i* = `sqrt(-1)`.
Note that *beta(k)* may be zero.
cs1, sn1 REAL for `slagv2`

	DOUBLE PRECISION for dlagv2
	The cosine and sine of the left rotation matrix, respectively.
<i>csr, snr</i>	REAL for slagv2
	DOUBLE PRECISION for dlagv2
	The cosine and sine of the right rotation matrix, respectively.

?lahqr

Computes the eigenvalues and Schur factorization of an upper Hessenberg matrix, using the double-shift/single-shift QR algorithm.

Syntax

```
call slahqr( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi, iloz, ihiz, z, ldz,
info )

call dlahqr( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi, iloz, ihiz, z, ldz,
info )

call clahqr( wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, info
)

call zlahqr( wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, info
)
```

Description

This routine is an auxiliary routine called by [?hseqr](#) to update the eigenvalues and Schur decomposition already computed by [?hseqr](#), by dealing with the Hessenberg submatrix in rows and columns *ilo* to *ihi*.

Input Parameters

<i>wantt</i>	LOGICAL. If <i>wantt</i> = .TRUE., the full Schur form <i>T</i> is required; If <i>wantt</i> = .FALSE., eigenvalues only are required.
<i>wantz</i>	LOGICAL. If <i>wantz</i> = .TRUE., the matrix of Schur vectors <i>Z</i> is required; If <i>wantz</i> = .FALSE., Schur vectors are not required.

n INTEGER. The order of the matrix H ($n \geq 0$).

ilo, ihi INTEGER.
It is assumed that h is already upper quasi-triangular in rows and columns $ihi+1:n$, and that $h(ilo, ilo-1) = 0$ (unless $ilo = 1$). The routine ?lahqr works primarily with the Hessenberg submatrix in rows and columns ilo to ihi , but applies transformations to all of h if $wantt = .TRUE..$
Constraints:
 $1 \leq ilo \leq \max(1, ihi); ihi \leq n$.

h, z REAL for slahqr
DOUBLE PRECISION for dlahqr
COMPLEX for clahqr
COMPLEX*16 for zlahqr.
Arrays:
 $h(ldh,*)$ contains the upper Hessenberg matrix h .
The second dimension of h must be at least $\max(1, n)$.
 $z(ldz,*)$
If $wantz = .TRUE.$, then, on entry, z must contain the current matrix z of transformations accumulated by ?hseqr.
If $wantz = .FALSE.$, then z is not referenced. The second dimension of z must be at least $\max(1, n)$.

ldh INTEGER. The first dimension of h ; at least $\max(1, n)$.

ldz INTEGER. The first dimension of z ; at least $\max(1, n)$.

iloz, ihiz INTEGER. Specify the rows of z to which transformations must be applied if $wantz = .TRUE..$
 $1 \leq iloz \leq ilo; ihi \leq ihiz \leq n$.

Output Parameters

h On exit, if $info = 0$ and $wantt = .TRUE.$, then,

- for slahqr/dlahqr, h is upper quasi-triangular in rows and columns $ilo:ihi$ with any 2-by-2 diagonal blocks in standard form.
- for clahqr/zlahqr, h is upper triangular in rows and columns $ilo:ihi$.

If *info* = 0 and *wantt* = .FALSE., the contents of *h* are unspecified on exit. If *info* is positive, see description of *info* for the output state of *h*.

wr, wi REAL for *slahqr*
 DOUBLE PRECISION for *dlahqr*
 Arrays, DIMENSION at least $\max(1, n)$ each. Used with real flavors only. The real and imaginary parts, respectively, of the computed eigenvalues *ilo* to *ihi* are stored in the corresponding elements of *wr* and *wi*. If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of *wr* and *wi*, say the *i*-th and (*i*+1)-th, with *wi*(*i*) > 0 and *wi*(*i*+1) < 0.
 If *wantt* = .TRUE., the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *h*, with *wr*(*i*) = *h*(*i*,*i*), and, if *h*(*i*:*i*+1, *i*:*i*+1) is a 2-by-2 diagonal block, *wi*(*i*) = $\sqrt{h(i+1,i)*h(i,i+1)}$ and *wi*(*i*+1) = -*wi*(*i*).

w COMPLEX for *clahqr*
 COMPLEX*16 for *zlahqr*.
 Array, DIMENSION at least $\max(1, n)$. Used with complex flavors only. The computed eigenvalues *ilo* to *ihi* are stored in the corresponding elements of *w*.
 If *wantt* = .TRUE., the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *h*, with *w*(*i*) = *h*(*i*,*i*).

z If *wantz* = .TRUE., then, on exit *z* has been updated; transformations are applied only to the submatrix *z*(*iloz*:*ihiz*, *ilo*:*ihi*).

info INTEGER.
 If *info* = 0, the execution is successful.
 With *info* > 0,

- if *info* = *i*, ?*lahqr* failed to compute all the eigenvalues *ilo* to *ihi* in a total of 30 iterations per eigenvalue; elements *i*+1:*ihi* of *wr* and *wi* (for *slahqr*/*dlahqr*) or *w* (for *clahqr*/*zlahqr*) contain those eigenvalues which have been successfully computed.

- if `wantt` is `.FALSE.`, then on exit the remaining unconverged eigenvalues are the eigenvalues of the upper Hessenberg matrix rows and columns `ilo` through `info` of the final output value of `h`.
- if `wantt` is `.TRUE.`, then on exit

(initial value of h)* u = u *(final value of h),
(*)

where u is an orthogonal matrix. The final value of h is upper Hessenberg and triangular in rows and columns `info+1` through `ihi`.

- if `wantz` is `.TRUE.`, then on exit

(final value of z) = (initial value of z)* u ,

where u is an orthogonal matrix in (*) regardless of the value of `wantt`.

?lahrd

Reduces the first `nb` columns of a general rectangular matrix A so that elements below the k -th subdiagonal are zero, and returns auxiliary matrices which are needed to apply the transformation to the unreduced part of A .

Syntax

```
call slahrd( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call dlahrd( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call clahrd( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call zlahrd( n, k, nb, a, lda, tau, t, ldt, y, ldy )
```

Description

The routine reduces the first nb columns of a real/complex general n -by- $(n-k+1)$ matrix A so that elements below the k -th subdiagonal are zero. The reduction is performed by an orthogonal/unitary similarity transformation $Q^* A Q$. The routine returns the matrices V and T which determine Q as a block reflector $I - V^* T^* V$, and also the matrix $Y = A^* V^* T$.

The matrix Q is represented as products of nb elementary reflectors:

$$Q = H(1) * H(2) * \dots * H(nb)$$

Each $H(i)$ has the form

$$H(i) = I - \tau v v^*$$

where τ is a real/complex scalar, and v is a real/complex vector.

This is an obsolete auxiliary routine. Please use the new routine `zlahr2` instead.

Input Parameters

n	INTEGER. The order of the matrix A ($n \geq 0$).
k	INTEGER. The offset for the reduction. Elements below the k -th subdiagonal in the first nb columns are reduced to zero.
nb	INTEGER. The number of columns to be reduced.
a	REAL for <code>slahrd</code> DOUBLE PRECISION for <code>dlahrd</code> COMPLEX for <code>clahrd</code> COMPLEX*16 for <code>zlahrd</code> . Array $a(lda, n-k+1)$ contains the n -by- $(n-k+1)$ general matrix A to be reduced.
lda	INTEGER. The first dimension of a ; at least $\max(1, n)$.
ldt	INTEGER. The first dimension of the output array t ; must be at least $\max(1, nb)$.
ldy	INTEGER. The first dimension of the output array y ; must be at least $\max(1, n)$.

Output Parameters

<i>a</i>	On exit, the elements on and above the <i>k</i> -th subdiagonal in the first <i>nb</i> columns are overwritten with the corresponding elements of the reduced matrix; the elements below the <i>k</i> -th subdiagonal, with the array <i>tau</i> , represent the matrix <i>Q</i> as a product of elementary reflectors. The other columns of <i>a</i> are unchanged. See <i>Application Notes</i> below.
<i>tau</i>	REAL for slahrd DOUBLE PRECISION for dlahrd COMPLEX for clahrd COMPLEX*16 for zlahrd. Array, DIMENSION (<i>nb</i>). Contains scalar factors of the elementary reflectors.
<i>t</i> , <i>y</i>	REAL for slahrd DOUBLE PRECISION for dlahrd COMPLEX for clahrd COMPLEX*16 for zlahrd. Arrays, dimension <i>t</i> (<i>ldt</i> , <i>nb</i>), <i>y</i> (<i>ldy</i> , <i>nb</i>). The array <i>t</i> contains upper triangular matrix <i>T</i> . The array <i>y</i> contains the <i>n</i> -by- <i>nb</i> matrix <i>Y</i> .

Application Notes

For the elementary reflector $H(i)$,

$v(1:i+k-1) = 0$, $v(i+k) = 1$; $v(i+k+1:n)$ is stored on exit in $a(i+k+1:n, i)$ and *tau* is stored in *tau*(*i*).

The elements of the vectors *v* together form the (*n-k+1*)-by-*nb* matrix *V* which is needed, with *T* and *Y*, to apply the transformation to the unreduced part of the matrix, using an update of the form:

$$A := (I - V^* T^* V') * (A - Y^* V').$$

The contents of *A* on exit are illustrated by the following example with $n = 7$, $k = 3$ and $nb = 2$:

$$\begin{bmatrix} a & h & a & a & a \\ a & h & a & a & a \\ a & h & a & a & a \\ h & h & a & a & a \\ v_1 & h & a & a & a \\ v_1 & v_2 & a & a & a \\ v_1 & v_2 & a & a & a \end{bmatrix}$$

where a denotes an element of the original matrix A , h denotes a modified element of the upper Hessenberg matrix H , and v_i denotes an element of the vector defining $H(i)$.

slahr2

Reduces the specified number of first columns of a general rectangular matrix A so that elements below the specified subdiagonal are zero, and returns auxiliary matrices which are needed to apply the transformation to the unreduced part of A .

Syntax

```
call slahr2( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call dlahr2( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call clahr2( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call zlahr2( n, k, nb, a, lda, tau, t, ldt, y, ldy )
```

Description

The routine reduces the first nb columns of a real/complex general n -by- $(n-k+1)$ matrix A so that elements below the k -th subdiagonal are zero. The reduction is performed by an orthogonal/unitary similarity transformation Q'^*A*Q . The routine returns the matrices V and T which determine Q as a block reflector $I - V*T*V'$, and also the matrix $Y = A*V*T$.

The matrix Q is represented as products of nb elementary reflectors:

$$Q = H(1) * H(2) * \dots * H(nb)$$

Each $H(i)$ has the form

$$H(i) = I - \tau v v'$$

where τ is a real/complex scalar, and v is a real/complex vector.

This is an auxiliary routine called by `?gehrd`.

Input Parameters

n	INTEGER. The order of the matrix A ($n \geq 0$).
k	INTEGER. The offset for the reduction. Elements below the k -th subdiagonal in the first nb columns are reduced to zero ($k < n$).
nb	INTEGER. The number of columns to be reduced.
a	REAL for <code>slahr2</code> DOUBLE PRECISION for <code>dlahr2</code> COMPLEX for <code>clahr2</code> COMPLEX*16 for <code>zlahr2</code> . Array, DIMENSION ($lda, n-k+1$) contains the n -by- $(n-k+1)$ general matrix A to be reduced.
lda	INTEGER. The first dimension of the array a ; $lda \geq \max(1, n)$.
ldt	INTEGER. The first dimension of the output array t ; $ldt \geq nb$.
ldy	INTEGER. The first dimension of the output array y ; $ldy \geq n$.

Output Parameters

a	On exit, the elements on and above the k -th subdiagonal in the first nb columns are overwritten with the corresponding elements of the reduced matrix; the elements below the k -th subdiagonal, with the array τ , represent the matrix Q as a product of elementary reflectors. The other columns of a are unchanged. See <i>Application Notes</i> below.

tau REAL for slahr2
DOUBLE PRECISION for dlahr2
COMPLEX for clahr2
COMPLEX*16 for zlahr2.
Array, DIMENSION (*nb*).
Contains scalar factors of the elementary reflectors.

t, y REAL for slahr2
DOUBLE PRECISION for dlahr2
COMPLEX for clahr2
COMPLEX*16 for zlahr2.
Arrays, dimension $t(ldt, nb)$, $y(ldy, nb)$.
The array *t* contains upper triangular matrix *T*.
The array *y* contains the *n*-by-*nb* matrix *Y*.

Application Notes

For the elementary reflector $H(i)$,

$v(1:i+k-1) = 0$, $v(i+k) = 1$; $v(i+k+1:n)$ is stored on exit in $a(i+k+1:n, i)$ and *tau* is stored in *tau*(*i*).

The elements of the vectors *v* together form the $(n-k+1)$ -by-*nb* matrix *V* which is needed, with *T* and *Y*, to apply the transformation to the unreduced part of the matrix, using an update of the form:

$$A := (I - V * T * V') * (A - Y * V').$$

The contents of *A* on exit are illustrated by the following example with $n = 7$, $k = 3$ and *nb* = 2:

$$\begin{bmatrix} a & a & a & a & a \\ a & a & a & a & a \\ a & a & a & a & a \\ h & h & a & a & a \\ v_1 & h & a & a & a \\ v_1 & v_2 & a & a & a \\ v_1 & v_2 & a & a & a \end{bmatrix}$$

where a denotes an element of the original matrix A , h denotes a modified element of the upper Hessenberg matrix H , and v_i denotes an element of the vector defining $H(i)$.

?laic1

Applies one step of incremental condition estimation.

Syntax

```
call slaic1( job, j, x, sest, w, gamma, sestpr, s, c )
call dlaic1( job, j, x, sest, w, gamma, sestpr, s, c )
call claic1( job, j, x, sest, w, gamma, sestpr, s, c )
call zlaic1( job, j, x, sest, w, gamma, sestpr, s, c )
```

Description

The routine ?laic1 applies one step of incremental condition estimation in its simplest version.

Let x , $\|x\|_2 = 1$ (where $\|a\|_2$ denotes the 2-norm of a), be an approximate singular vector of an j -by- j lower triangular matrix L , such that

$$\|L^*x\|_2 = \text{sest}$$

Then ?laic1 computes sestpr , s , c such that the vector

$$\text{xhat} = \begin{bmatrix} s^*x \\ c \end{bmatrix}$$

is an approximate singular vector of

$$\text{Lhat} = \begin{bmatrix} L & 0 \\ w^t & \text{gamma} \end{bmatrix}$$

in the sense that

$||Lhat*xhat||_2 = sestpr.$

Depending on *job*, an estimate for the largest or smallest singular value is computed.

Note that $[s \ c]^T$ and $sestpr^2$ is an eigenpair of the system (for slaic1/claic)

$$\text{diag}(sest*sest, 0) + [\alpha \ \gamma] * \begin{bmatrix} \alpha \\ \gamma \end{bmatrix}$$

where $\alpha = x'*w$;

or of the system (for claic1/zlaic)

$$\text{diag}(sest*sest, 0) + [\alpha \ \gamma] * \begin{bmatrix} \text{conjg}(\alpha) \\ \text{conjg}(\gamma) \end{bmatrix}$$

where $\alpha = \text{conjg}(x)^T * w$.

Input Parameters

<i>job</i>	INTEGER. If <i>job</i> =1, an estimate for the largest singular value is computed; If <i>job</i> =2, an estimate for the smallest singular value is computed;
<i>j</i>	INTEGER. Length of <i>x</i> and <i>w</i> .
<i>x, w</i>	REAL for slaic1 DOUBLE PRECISION for dlaic1 COMPLEX for claic1 COMPLEX*16 for zlaic1. Arrays, dimension (<i>j</i>) each. Contain vectors <i>x</i> and <i>w</i> , respectively.
<i>sest</i>	REAL for slaic1/claic1; DOUBLE PRECISION for dlaic1/zlaic1. Estimated singular value of <i>j</i> -by- <i>j</i> matrix <i>L</i> .
<i>gamma</i>	REAL for slaic1

DOUBLE PRECISION for `dlaic1`
 COMPLEX for `claic1`
 COMPLEX*16 for `zlaic1`.
 The diagonal element *gamma*.

Output Parameters

sestpr REAL for `slaic1/claic1`;
 DOUBLE PRECISION for `dlaic1/zlaic1`.
 Estimated singular value of $(j+1)$ -by- $(j+1)$ matrix *Lhat*.

s, c REAL for `slaic1`
 DOUBLE PRECISION for `dlaic1`
 COMPLEX for `claic1`
 COMPLEX*16 for `zlaic1`.
 Sine and cosine needed in forming *xhat*.

?laln2

Solves a 1-by-1 or 2-by-2 linear system of equations of the specified form.

Syntax

```
call slaln2( ltrans, na, nw, smin, ca, a, lda, d1, d2, b, ldb, wr, wi, x, ldx,
scale, xnorm, info )
```

```
call dlaln2( ltrans, na, nw, smin, ca, a, lda, d1, d2, b, ldb, wr, wi, x, ldx,
scale, xnorm, info )
```

Description

The routine solves a system of the form

$$(ca*A - w*D)*X = s*B, \text{ or } (ca*A' - w*D)*X = s*B$$

with possible scaling (*s*) and perturbation of *A* (*A'* means *A*-transpose.)

A is an *na*-by-*na* real matrix, *ca* is a real scalar, *D* is an *na*-by-*na* real diagonal matrix, *w* is a real or complex value, and *X* and *B* are *na*-by-1 matrices: real if *w* is real, complex if *w* is complex. The parameter *na* may be 1 or 2.

If *w* is complex, *X* and *B* are represented as *na*-by-2 matrices, the first column of each being the real part and the second being the imaginary part.

The routine computes the scaling factor s (≤ 1) so chosen that x can be computed without overflow. x is further scaled if necessary to assure that $\text{norm}(ca*A - w*D) * \text{norm}(X)$ is less than overflow.

If both singular values of $(ca*A - w*D)$ are less than $smin$, $smin*I$ (where I stands for identity) will be used instead of $(ca*A - w*D)$. If only one singular value is less than $smin$, one element of $(ca*A - w*D)$ will be perturbed enough to make the smallest singular value roughly $smin$.

If both singular values are at least $smin$, $(ca*A - w*D)$ will not be perturbed. In any case, the perturbation will be at most some small multiple of $\max(smin, ulp * \text{norm}(ca*A - w*D))$.

The singular values are computed by infinity-norm approximations, and thus will only be correct to a factor of 2 or so.



NOTE. All input quantities are assumed to be smaller than overflow by a reasonable factor (see *bignum*).

Input Parameters

<i>trans</i>	LOGICAL. If <i>trans</i> = .TRUE., <i>A</i> - transpose will be used. If <i>trans</i> = .FALSE., <i>A</i> will be used (not transposed.)
<i>na</i>	INTEGER. The size of the matrix <i>A</i> , possible values 1 or 2.
<i>nw</i>	INTEGER. This parameter must be 1 if <i>w</i> is real, and 2 if <i>w</i> is complex. Possible values 1 or 2.
<i>smin</i>	REAL for slaln2 DOUBLE PRECISION for dlaln2. The desired lower bound on the singular values of <i>A</i> . This should be a safe distance away from underflow or overflow, for example, between $(\text{underflow}/\text{machine\_precision})$ and $(\text{machine\_precision} * \text{overflow})$. (See <i>bignum</i> and <i>ulp</i>).
<i>ca</i>	REAL for slaln2 DOUBLE PRECISION for dlaln2. The coefficient by which <i>A</i> is multiplied.
<i>a</i>	REAL for slaln2 DOUBLE PRECISION for dlaln2.

Array, DIMENSION (lda, na).
The na -by- na matrix A .

lda INTEGER. The leading dimension of a . Must be at least na .

$d1, d2$ REAL for slaln2
DOUBLE PRECISION for dlaln2.
The (1,1) and (2,2) elements in the diagonal matrix D , respectively. $d2$ is not used if $nw = 1$.

b REAL for slaln2
DOUBLE PRECISION for dlaln2.
Array, DIMENSION (ldb, nw). The na -by- nw matrix B (right-hand side). If $nw = 2$ (w is complex), column 1 contains the real part of B and column 2 contains the imaginary part.

ldb INTEGER. The leading dimension of b . Must be at least na .

wr, wi REAL for slaln2
DOUBLE PRECISION for dlaln2.
The real and imaginary part of the scalar w , respectively. wi is not used if $nw = 1$.

ldx INTEGER. The leading dimension of the output array x . Must be at least na .

Output Parameters

x REAL for slaln2
DOUBLE PRECISION for dlaln2.
Array, DIMENSION (ldx, nw). The na -by- nw matrix X (unknowns), as computed by the routine. If $nw = 2$ (w is complex), on exit, column 1 will contain the real part of x and column 2 will contain the imaginary part.

$scale$ REAL for slaln2
DOUBLE PRECISION for dlaln2.
The scale factor that B must be multiplied by to insure that overflow does not occur when computing X . Thus $(ca * A - w * D) X$ will be $scale * B$, not B (ignoring perturbations of A .) It will be at most 1.

$xnorm$ REAL for slaln2
DOUBLE PRECISION for dlaln2.

info

The infinity-norm of X , when X is regarded as an na -by- nw real matrix.

INTEGER.

An error flag. It will be zero if no error occurs, a negative number if an argument is in error, or a positive number if $(ca*A - w*D)$ had to be perturbed. The possible values are:

If $info = 0$: no error occurred, and $(ca*A - w*D)$ did not have to be perturbed.

If $info = 1$: $(ca*A - w*D)$ had to be perturbed to make its smallest (or only) singular value greater than $smin$.



NOTE. For higher speed, this routine does not check the inputs for errors.

?lals0

Applies back multiplying factors in solving the least squares problem using divide and conquer SVD approach. Used by ?gelsd.

Syntax

```
call slals0 ( icalpq, nl, nr, sqre, nrhs, b, ldb, bx, ldbx, perm, givptr,
             givcol, ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, work, info )
call dlals0 ( icalpq, nl, nr, sqre, nrhs, b, ldb, bx, ldbx, perm, givptr,
             givcol, ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, work, info )
call clals0( icalpq, nl, nr, sqre, nrhs, b, ldb, bx, ldbx, perm, givptr,
             givcol, ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, rwork, info )
call zlals0( icalpq, nl, nr, sqre, nrhs, b, ldb, bx, ldbx, perm, givptr,
             givcol, ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, rwork, info )
```

Description

The routine applies back the multiplying factors of either the left or right singular vector matrix of a diagonal matrix appended by a row to the right hand side matrix B in solving the least squares problem using the divide-and-conquer SVD approach.

For the left singular vector matrix, three types of orthogonal matrices are involved:

(1L) Givens rotations: the number of such rotations is *givptr*; the pairs of columns/rows they were applied to are stored in *givcol*; and the *c*- and *s*-values of these rotations are stored in *givnum*.

(2L) Permutation. The (*nl*+1)-st row of *B* is to be moved to the first row, and for *j*=2:*n*, *perm*(*j*)-th row of *B* is to be moved to the *j*-th row.

(3L) The left singular vector matrix of the remaining matrix.

For the right singular vector matrix, four types of orthogonal matrices are involved:

(1R) The right singular vector matrix of the remaining matrix.

(2R) If *sqre* = 1, one extra Givens rotation to generate the right null space.

(3R) The inverse transformation of (2L).

(4R) The inverse transformation of (1L).

Input Parameters

<i>icompg</i>	INTEGER. Specifies whether singular vectors are to be computed in factored form: If <i>icompg</i> = 0: Left singular vector matrix. If <i>icompg</i> = 1: Right singular vector matrix.
<i>nl</i>	INTEGER. The row dimension of the upper block. $nl \geq 1$.
<i>nr</i>	INTEGER. The row dimension of the lower block. $nr \geq 1$.
<i>sqre</i>	INTEGER. If <i>sqre</i> = 0: the lower block is an <i>nr</i> -by- <i>nr</i> square matrix. If <i>sqre</i> = 1: the lower block is an <i>nr</i> -by-(<i>nr</i> +1) rectangular matrix. The bidiagonal matrix has row dimension $n = nl + nr + 1$, and column dimension $m = n + sqre$.
<i>nrhs</i>	INTEGER. The number of columns of <i>B</i> and <i>bx</i> . Must be at least 1.
<i>b</i>	REAL for slals0 DOUBLE PRECISION for dlals0 COMPLEX for clals0 COMPLEX*16 for zlals0. Array, DIMENSION (<i>ldb</i> , <i>nrhs</i>).

	Contains the right hand sides of the least squares problem in rows 1 through <i>m</i> .
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> . Must be at least $\max(1, \max(m, n))$.
<i>bx</i>	REAL for slals0 DOUBLE PRECISION for dlals0 COMPLEX for clals0 COMPLEX*16 for zlals0. Workspace array, DIMENSION (<i>ldbx</i> , <i>nrhs</i>).
<i>ldbx</i>	INTEGER. The leading dimension of <i>bx</i> .
<i>perm</i>	INTEGER. Array, DIMENSION (<i>n</i>). The permutations (from deflation and sorting) applied to the two blocks.
<i>givptr</i>	INTEGER. The number of Givens rotations which took place in this subproblem.
<i>givcol</i>	INTEGER. Array, DIMENSION (<i>ldgcol</i> , 2). Each pair of numbers indicates a pair of rows/columns involved in a Givens rotation.
<i>ldgcol</i>	INTEGER. The leading dimension of <i>givcol</i> , must be at least <i>n</i> .
<i>givnum</i>	REAL for slals0/clals0 DOUBLE PRECISION for dlals0/zlals0 Array, DIMENSION (<i>ldgnum</i> , 2). Each number indicates the <i>c</i> or <i>s</i> value used in the corresponding Givens rotation.
<i>ldgnum</i>	INTEGER. The leading dimension of arrays <i>difr</i> , <i>poles</i> and <i>givnum</i> , must be at least <i>k</i> .
<i>poles</i>	REAL for slals0/clals0 DOUBLE PRECISION for dlals0/zlals0 Array, DIMENSION (<i>ldgnum</i> , 2). On entry, <i>poles</i> (1: <i>k</i> , 1) contains the new singular values obtained from solving the secular equation, and <i>poles</i> (1: <i>k</i> , 2) is an array containing the poles in the secular equation.
<i>difl</i>	REAL for slals0/clals0 DOUBLE PRECISION for dlals0/zlals0

	Array, DIMENSION (k). On entry, $difl(i)$ is the distance between i -th updated (undeflated) singular value and the i -th (undeflated) old singular value.
<i>difr</i>	REAL for slals0/clals0 DOUBLE PRECISION for dlals0/zlals0 Array, DIMENSION ($ldgnum, 2$). On entry, $difr(i, 1)$ contains the distances between i -th updated (undeflated) singular value and the $i+1$ -th (undeflated) old singular value. And $difr(i, 2)$ is the normalizing factor for the i -th right singular vector.
<i>z</i>	REAL for slals0/clals0 DOUBLE PRECISION for dlals0/zlals0 Array, DIMENSION (k). Contains the components of the deflation-adjusted updating row vector.
<i>K</i>	INTEGER. Contains the dimension of the non-deflated matrix. This is the order of the related secular equation. $1 \leq k \leq n$.
<i>c</i>	REAL for slals0/clals0 DOUBLE PRECISION for dlals0/zlals0 Contains garbage if $sqr = 0$ and the c value of a Givens rotation related to the right null space if $sqr = 1$.
<i>s</i>	REAL for slals0/clals0 DOUBLE PRECISION for dlals0/zlals0 Contains garbage if $sqr = 0$ and the s value of a Givens rotation related to the right null space if $sqr = 1$.
<i>work</i>	REAL for slals0 DOUBLE PRECISION for dlals0 Workspace array, DIMENSION (k). Used with real flavors only.
<i>rwork</i>	REAL for clals0 DOUBLE PRECISION for zlals0 Workspace array, DIMENSION ($k*(1+nrhs) + 2*nrhs$). Used with complex flavors only.

Output Parameters

<i>b</i>	On exit, contains the solution x in rows 1 through n .

info INTEGER.
 If *info* = 0: successful exit.
 If *info* = -*i* < 0, the *i*-th argument had an illegal value.

?lalsa

Computes the SVD of the coefficient matrix in compact form. Used by ?gelsd.

Syntax

```
call slalsa( icalmpq, smlsiz, n, nrhs, b, ldb, bx, ldbx, u, ldu, vt, k, difl,
difr, z, poles, givptr, givcol, ldgcol, perm, givnum, c, s, work, iwork, info
)

call dlalsa( icalmpq, smlsiz, n, nrhs, b, ldb, bx, ldbx, u, ldu, vt, k, difl,
difr, z, poles, givptr, givcol, ldgcol, perm, givnum, c, s, work, iwork, info
)

call clalsa( icalmpq, smlsiz, n, nrhs, b, ldb, bx, ldbx, u, ldu, vt, k, difl,
difr, z, poles, givptr, givcol, ldgcol, perm, givnum, c, s, rwork, iwork,
info )

call zlalsa( icalmpq, smlsiz, n, nrhs, b, ldb, bx, ldbx, u, ldu, vt, k, difl,
difr, z, poles, givptr, givcol, ldgcol, perm, givnum, c, s, rwork, iwork,
info )
```

Description

The routine is an intermediate step in solving the least squares problem by computing the SVD of the coefficient matrix in compact form. The singular vectors are computed as products of simple orthogonal matrices.

If *icalmpq* = 0, ?lalsa applies the inverse of the left singular vector matrix of an upper bidiagonal matrix to the right hand side; and if *icalmpq* = 1, the routine applies the right singular vector matrix to the right hand side. The singular vector matrices were generated in the compact form by ?lalsa.

Input Parameters

icalmpq INTEGER. Specifies whether the left or the right singular vector matrix is involved. If *icalmpq* = 0: left singular vector matrix is used

	If <i>icompg</i> = 1: right singular vector matrix is used.
<i>smlsiz</i>	INTEGER. The maximum size of the subproblems at the bottom of the computation tree.
<i>n</i>	INTEGER. The row and column dimensions of the upper bidiagonal matrix.
<i>nrhs</i>	INTEGER. The number of columns of <i>b</i> and <i>bx</i> . Must be at least 1.
<i>b</i>	REAL for slalsa DOUBLE PRECISION for dlalsa COMPLEX for clalsa COMPLEX*16 for zlalsa Array, DIMENSION (<i>ldb</i> , <i>nrhs</i>). Contains the right hand sides of the least squares problem in rows 1 through <i>m</i> .
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> in the calling subprogram. Must be at least $\max(1, \max(m, n))$.
<i>ldb_x</i>	INTEGER. The leading dimension of the output array <i>bx</i> .
<i>u</i>	REAL for slalsa/clalsa DOUBLE PRECISION for dlalsa/zlalsa Array, DIMENSION (<i>ldu</i> , <i>smlsiz</i>). On entry, <i>u</i> contains the left singular vector matrices of all subproblems at the bottom level.
<i>ldu</i>	INTEGER, <i>ldu</i> ≥ <i>n</i> . The leading dimension of arrays <i>u</i> , <i>vt</i> , <i>difl</i> , <i>difr</i> , <i>poles</i> , <i>givnum</i> , and <i>z</i> .
<i>vt</i>	REAL for slalsa/clalsa DOUBLE PRECISION for dlalsa/zlalsa Array, DIMENSION (<i>ldu</i> , <i>smlsiz</i> + 1). On entry, contains the right singular vector matrices of all subproblems at the bottom level.
<i>k</i>	INTEGER array, DIMENSION (<i>n</i>).
<i>difl</i>	REAL for slalsa/clalsa DOUBLE PRECISION for dlalsa/zlalsa Array, DIMENSION (<i>ldu</i> , <i>nlvl</i>), where $nlvl = \text{int}(\log_2(n / (smlsiz + 1))) + 1$.
<i>difr</i>	REAL for slalsa/clalsa DOUBLE PRECISION for dlalsa/zlalsa

	Array, DIMENSION (<i>ldu</i>, 2*<i>nlvl</i>). On entry, <i>difl</i>(*, <i>i</i>) and <i>difr</i>(*, 2<i>i</i> -1) record distances between singular values on the <i>i</i>-th level and singular values on the (<i>i</i> -1)-th level, and <i>difr</i>(*, 2<i>i</i>) record the normalizing factors of the right singular vectors matrices of subproblems on <i>i</i>-th level.
<i>z</i>	REAL for slalsa/clalsa DOUBLE PRECISION for dlalsa/zlalsa Array, DIMENSION (<i>ldu</i>, <i>nlvl</i>). On entry, <i>z</i>(1, <i>i</i>) contains the components of the deflation- adjusted updating the row vector for subproblems on the <i>i</i>-th level.
<i>poles</i>	REAL for slalsa/clalsa DOUBLE PRECISION for dlalsa/zlalsa Array, DIMENSION (<i>ldu</i>, 2*<i>nlvl</i>). On entry, <i>poles</i>(*, 2<i>i</i>-1: 2<i>i</i>) contains the new and old singular values involved in the secular equations on the <i>i</i>-th level.
<i>givptr</i>	INTEGER. Array, DIMENSION (<i>n</i>). On entry, <i>givptr</i>(<i>i</i>) records the number of Givens rotations performed on the <i>i</i>-th problem on the computation tree.
<i>givcol</i>	INTEGER. Array, DIMENSION (<i>ldgcol</i>, 2*<i>nlvl</i>). On entry, for each <i>i</i>, <i>givcol</i>(*, 2<i>i</i>-1: 2<i>i</i>) records the locations of Givens rotations performed on the <i>i</i>-th level on the computation tree.
<i>ldgcol</i>	INTEGER, <i>ldgcol</i> ≥ <i>n</i>. The leading dimension of arrays <i>givcol</i> and <i>perm</i>.
<i>perm</i>	INTEGER. Array, DIMENSION (<i>ldgcol</i>, <i>nlvl</i>). On entry, <i>perm</i>(*, <i>i</i>) records permutations done on the <i>i</i>-th level of the computation tree.
<i>givnum</i>	REAL for slalsa/clalsa DOUBLE PRECISION for dlalsa/zlalsa Array, DIMENSION (<i>ldu</i>, 2*<i>nlvl</i>). On entry, <i>givnum</i>(*, 2<i>i</i>-1 : 2<i>i</i>) records the <i>c</i> and <i>s</i> values of Givens rotations performed on the <i>i</i>-th level on the computation tree.
<i>c</i>	REAL for slalsa/clalsa DOUBLE PRECISION for dlalsa/zlalsa

	Array, DIMENSION (n). On entry, if the i -th subproblem is not square, $c(i)$ contains the c value of a Givens rotation related to the right null space of the i -th subproblem.
s	REAL for slalsa/clalsa DOUBLE PRECISION for dlalsa/zlalsa Array, DIMENSION (n). On entry, if the i -th subproblem is not square, $s(i)$ contains the s -value of a Givens rotation related to the right null space of the i -th subproblem.
$work$	REAL for slalsa DOUBLE PRECISION for dlalsa Workspace array, DIMENSION at least (n). Used with real flavors only.
$rwork$	REAL for clalsa DOUBLE PRECISION for zlalsa Workspace array, DIMENSION at least $\max(n, (smlsz+1)*nrhs*3)$. Used with complex flavors only.
$iwork$	INTEGER. Workspace array, DIMENSION at least ($3n$).

Output Parameters

b	On exit, contains the solution x in rows 1 through n .
bx	REAL for slalsa DOUBLE PRECISION for dlalsa COMPLEX for clalsa COMPLEX*16 for zlalsa Array, DIMENSION ($ldbx, nrhs$). On exit, the result of applying the left or right singular vector matrix to b .
$info$	INTEGER. If $info = 0$: successful exit If $info = -i < 0$, the i -th argument had an illegal value.

zlalsd

Uses the singular value decomposition of A to solve the least squares problem.

Syntax

```
call slalsd( uplo, smlsiz, n, nrhs, d, e, b, ldb, rcond, rank, work, iwork,
info )

call dlalsd( uplo, smlsiz, n, nrhs, d, e, b, ldb, rcond, rank, work, iwork,
info )

call clalsd( uplo, smlsiz, n, nrhs, d, e, b, ldb, rcond, rank, work, rwork,
iwork, info )

call zlalsd( uplo, smlsiz, n, nrhs, d, e, b, ldb, rcond, rank, work, rwork,
iwork, info )
```

Description

The routine uses the singular value decomposition of A to solve the least squares problem of finding x to minimize the Euclidean norm of each column of A^*X-B , where A is n -by- n upper bidiagonal, and X and B are n -by- $nrhs$. The solution X overwrites B .

The singular values of A smaller than $rcond$ times the largest singular value are treated as zero in solving the least squares problem; in this case a minimum norm solution is returned. The actual singular values are returned in d in ascending order.

This code makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray XMP, Cray YMP, Cray C 90, or Cray 2.

It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

Input Parameters

<code>uplo</code>	CHARACTER*1. If <code>uplo = 'U'</code> , d and e define an upper bidiagonal matrix. If <code>uplo = 'L'</code> , d and e define a lower bidiagonal matrix.
<code>smlsiz</code>	INTEGER. The maximum size of the subproblems at the bottom of the computation tree.
<code>n</code>	INTEGER. The dimension of the bidiagonal matrix.

	$n \geq 0$.
<i>nrhs</i>	INTEGER. The number of columns of <i>B</i> . Must be at least 1.
<i>d</i>	REAL for slalsd/clalsd DOUBLE PRECISION for dlalsd/zlalsd Array, DIMENSION (<i>n</i>). On entry, <i>d</i> contains the main diagonal of the bidiagonal matrix.
<i>e</i>	REAL for slalsd/clalsd DOUBLE PRECISION for dlalsd/zlalsd Array, DIMENSION (<i>n</i> -1). Contains the super-diagonal entries of the bidiagonal matrix. On exit, <i>e</i> is destroyed.
<i>b</i>	REAL for slalsd DOUBLE PRECISION for dlalsd COMPLEX for clalsd COMPLEX*16 for zlalsd Array, DIMENSION (<i>ldb</i> , <i>nrhs</i>). On input, <i>b</i> contains the right hand sides of the least squares problem. On output, <i>b</i> contains the solution X.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> in the calling subprogram. Must be at least $\max(1, n)$.
<i>rcond</i>	REAL for slalsd/clalsd DOUBLE PRECISION for dlalsd/zlalsd The singular values of <i>A</i> less than or equal to <i>rcond</i> times the largest singular value are treated as zero in solving the least squares problem. If <i>rcond</i> is negative, machine precision is used instead. For example, for the least squares problem $\text{diag}(S) * X = B$, where $\text{diag}(S)$ is a diagonal matrix of singular values, the solution is $X(i) = B(i) / S(i)$ if $S(i)$ is greater than $rcond * \max(S)$, and $X(i) = 0$ if $S(i)$ is less than or equal to $rcond * \max(S)$.
<i>rank</i>	INTEGER. The number of singular values of <i>A</i> greater than <i>rcond</i> times the largest singular value.
<i>work</i>	REAL for slalsd DOUBLE PRECISION for dlalsd COMPLEX for clalsd COMPLEX*16 for zlalsd Workspace array.

work DIMENSION for real flavors at least
 $(9n+2n*smlsiz+8n*nlvl+n*nrhs+(smlsiz+1)^2),$
 where
 $nlvl = \max(0, \text{int}(\log_2(n/(smlsiz+1))) + 1).$
 DIMENSION for complex flavors is $(n*nrhs).$

rwork REAL for clalsd
 DOUBLE PRECISION for zlalsd
 Workspace array, used with complex flavors only.
 DIMENSION at least $(9n + 2n*smlsiz + 8n*nlvl + 3*smlsiz*nrhs + (smlsiz+1)^2),$
 where
 $nlvl = \max(0, \text{int}(\log_2(\min(m,n)/(smlsiz+1))) + 1).$

iwork INTEGER.
 Workspace array of DIMENSION $(3n*nlvl + 11n).$

Output Parameters

d On exit, if *info* = 0, *d* contains singular values of the bidiagonal matrix.

e On exit, destroyed.

b On exit, *b* contains the solution *x*.

info INTEGER.
 If *info* = 0: successful exit.
 If *info* = -*i* < 0, the *i*-th argument had an illegal value.
 If *info* > 0: The algorithm failed to compute a singular value while working on the submatrix lying in rows and columns *info*/(*n*+1) through mod(*info*, *n*+1).

zlamrg

Creates a permutation list to merge the entries of two independently sorted sets into a single set sorted in ascending order.

Syntax

```
call slamrg( n1, n2, a, strd1, strd2, index )
call dlamrg( n1, n2, a, strd1, strd2, index )
```

Description

The routine creates a permutation list which will merge the elements of *a* (which is composed of two independently sorted sets) into a single set which is sorted in ascending order.

Input Parameters

<i>n1, n2</i>	INTEGER. These arguments contain the respective lengths of the two sorted lists to be merged.
<i>a</i>	REAL for slamrg DOUBLE PRECISION for dlamrg. Array, DIMENSION (<i>n1+n2</i>). The first <i>n1</i> elements of <i>a</i> contain a list of numbers which are sorted in either ascending or descending order. Likewise for the final <i>n2</i> elements.
<i>strd1, strd2</i>	INTEGER. These are the strides to be taken through the array <i>a</i> . Allowable strides are 1 and -1. They indicate whether a subset of <i>a</i> is sorted in ascending (<i>strdx</i> = 1) or descending (<i>strdx</i> = -1) order.

Output Parameters

<i>index</i>	INTEGER. Array, DIMENSION (<i>n1+n2</i>). On exit, this array will contain a permutation such that if $b(i) = a(index(i))$ for $i=1, n1+n2$, then <i>b</i> will be sorted in ascending order.

?laneg

Computes the Sturm count, the number of negative pivots encountered while factoring tridiagonal

$$T\text{-}\sigma I = L^*D^*L^T.$$

Syntax

```
value = slaneg( n, d, lld, sigma, pivmin, r )
```

```
value = dlaneg( n, d, lld, sigma, pivmin, r )
```

Description

The routine computes the Sturm count, the number of negative pivots encountered while factoring tridiagonal $T\text{-}\sigma I = L^*D^*L^T$. This implementation works directly on the factors without forming the tridiagonal matrix T . The Sturm count is also the number of eigenvalues of T less than σ . This routine is called from ?larb. The current routine does not use the *pivmin* parameter but rather requires IEEE-754 propagation of infinities and NaNs (NaN stands for 'Not A Number'). This routine also has no input range restrictions but does require default exception handling such that $x/0$ produces Inf when x is non-zero, and Inf/Inf produces NaN. (For more information see [Marques06]).

Input Parameters

<i>n</i>	INTEGER. The order of the matrix.
<i>d</i>	REAL for slaneg DOUBLE PRECISION for dlaneg Array, DIMENSION (<i>n</i>). Contains <i>n</i> diagonal elements of the matrix <i>D</i> .
<i>lld</i>	REAL for slaneg DOUBLE PRECISION for dlaneg Array, DIMENSION (<i>n</i> -1). Contains (<i>n</i> -1) elements $L(i)^*L(i)^*D(i)$.
<i>sigma</i>	REAL for slaneg DOUBLE PRECISION for dlaneg Shift amount in $T\text{-}\sigma I = L^*D^*L^*T$.
<i>pivmin</i>	REAL for slaneg DOUBLE PRECISION for dlaneg

The minimum pivot in the Sturm sequence. May be used when zero pivots are encountered on non-IEEE-754 architectures.

r INTEGER.
The twist index for the twisted factorization that is used for the negcount.

Output Parameters

value INTEGER. The number of negative pivots encountered while factoring.

?langb

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of general band matrix.

Syntax

val = slangb(*norm*, *n*, *kl*, *ku*, *ab*, *ldab*, *work*)

val = dlangb(*norm*, *n*, *kl*, *ku*, *ab*, *ldab*, *work*)

val = clangb(*norm*, *n*, *kl*, *ku*, *ab*, *ldab*, *work*)

val = zlangb(*norm*, *n*, *kl*, *ku*, *ab*, *ldab*, *work*)

Description

The function returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an n -by- n band matrix A , with kl sub-diagonals and ku super-diagonals.

Input Parameters

norm CHARACTER*1. Specifies the value to be returned by the routine:
 = 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A .
 = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum),

= 'I' or 'i': *val* = normI(*A*), infinity norm of the matrix *A* (maximum row sum),
 = 'F', 'f', 'E' or 'e': *val* = normF(*A*), Frobenius norm of the matrix *A* (square root of sum of squares).

n INTEGER. The order of the matrix *A*. $n \geq 0$. When $n = 0$, ?langb is set to zero.

kl INTEGER. The number of sub-diagonals of the matrix *A*. $kl \geq 0$.

ku INTEGER. The number of super-diagonals of the matrix *A*. $ku \geq 0$.

ab REAL for slangb
 DOUBLE PRECISION for dlangb
 COMPLEX for clangb
 COMPLEX*16 for zlangb
 Array, DIMENSION (*ldab*,*n*).
 The band matrix *A*, stored in rows 1 to $kl+ku+1$. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:
 $ab(ku+1+i-j, j) = a(i, j)$
 for $\max(1, j-ku) \leq i \leq \min(n, j+kl)$.

ldab INTEGER. The leading dimension of the array *ab*.
 $ldab \geq kl+ku+1$.

work REAL for slangb/clangb
 DOUBLE PRECISION for dlangb/zlangb
 Workspace array, DIMENSION ($\max(1, lwork)$), where
 $lwork \geq n$ when *norm* = 'I'; otherwise, *work* is not referenced.

Output Parameters

val REAL for slangb/clangb
 DOUBLE PRECISION for dlangb/zlangb
 Value returned by the function.

?lange

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a general rectangular matrix.

Syntax

```
val = slange( norm, m, n, a, lda, work )
val = dlange( norm, m, n, a, lda, work )
val = clange( norm, m, n, a, lda, work )
val = zlange( norm, m, n, a, lda, work )
```

Description

The function ?lange returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex matrix *A*.

Input Parameters

<i>norm</i>	CHARACTER*1. Specifies the value to be returned by the routine: = 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix <i>A</i> . = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix <i>A</i> (maximum column sum), = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix <i>A</i> (maximum row sum), = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix <i>A</i> (square root of sum of squares).
<i>m</i>	INTEGER. The number of rows of the matrix <i>A</i> . $m \geq 0$. When $m = 0$, ?lange is set to zero.
<i>n</i>	INTEGER. The number of columns of the matrix <i>A</i> . $n \geq 0$. When $n = 0$, ?lange is set to zero.
<i>a</i>	REAL for slange DOUBLE PRECISION for dlange COMPLEX for clange COMPLEX*16 for zlange

	Array, DIMENSION (<i>lda</i> , <i>n</i>).
	The <i>m</i> -by- <i>n</i> matrix <i>A</i> .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . <i>lda</i> ≥ max(<i>m</i> ,1).
<i>work</i>	REAL for <i>srange</i> and <i>clrange</i> . DOUBLE PRECISION for <i>drange</i> and <i>zrange</i> . Workspace array, DIMENSION max(1, <i>lwork</i>), where <i>lwork</i> ≥ <i>m</i> when <i>norm</i> = 'I'; otherwise, <i>work</i> is not referenced.

Output Parameters

<i>val</i>	REAL for <i>srange</i> / <i>clrange</i> DOUBLE PRECISION for <i>drange</i> / <i>zrange</i> Value returned by the function.

?langt

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a general tridiagonal matrix.

Syntax

```
val = slangt( norm, n, dl, d, du )
val = dlangt( norm, n, dl, d, du )
val = clangt( norm, n, dl, d, du )
val = zlangt( norm, n, dl, d, du )
```

Description

The routine returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex tridiagonal matrix *A*.

Input Parameters

<i>norm</i>	CHARACTER*1. Specifies the value to be returned by the routine:

= 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A .
 = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum),
 = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum),
 = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).

n
 INTEGER. The order of the matrix A . $n \geq 0$. When $n = 0$, **?langt** is set to zero.

dl, d, du
 REAL for slangt
 DOUBLE PRECISION for dlangt
 COMPLEX for clangt
 COMPLEX*16 for zlangt
Arrays: $dl(n-1)$, $d(n)$, $du(n-1)$.
 The array dl contains the $(n-1)$ sub-diagonal elements of A .
 The array d contains the diagonal elements of A .
 The array du contains the $(n-1)$ super-diagonal elements of A .

Output Parameters

val
 REAL for slangt/clangt
 DOUBLE PRECISION for dlangt/zlangt
 Value returned by the function.

?lanhs

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of an upper Hessenberg matrix.

Syntax

```
val = slanh( norm, n, a, lda, work )
val = dlanhs( norm, n, a, lda, work )
val = clanh( norm, n, a, lda, work )
val = zlanhs( norm, n, a, lda, work )
```

Description

The function `?lanhs` returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a Hessenberg matrix A .

The value `val` returned by the function is:

```
val = max(abs(Aij)), if norm = 'M' or 'm'
= norm1(A), if norm = '1' or 'O' or 'o'
= normI(A), if norm = 'I' or 'i'
= normF(A), if norm = 'F', 'f', 'E' or 'e'
```

where `norm1` denotes the 1-norm of a matrix (maximum column sum), `normI` denotes the infinity norm of a matrix (maximum row sum) and `normF` denotes the Frobenius norm of a matrix (square root of sum of squares). Note that `max(abs(Aij))` is not a consistent matrix norm.

Input Parameters

<code>norm</code>	CHARACTER*1. Specifies the value to be returned by the routine as described above.
<code>n</code>	INTEGER. The order of the matrix A . $n \geq 0$. When $n = 0$, <code>?lanhs</code> is set to zero.
<code>a</code>	REAL for <code>slanh</code> DOUBLE PRECISION for <code>dlanhs</code> COMPLEX for <code>clanh</code>

COMPLEX*16 for zlanhs
 Array, DIMENSION (lda, n). The n -by- n upper Hessenberg matrix A ; the part of A below the first sub-diagonal is not referenced.

lda INTEGER. The leading dimension of the array a .
 $lda \geq \max(n, 1)$.

$work$ REAL for slanhb and clanhb.
 DOUBLE PRECISION for dlange and zlange.
 Workspace array, DIMENSION ($\max(1, lwork)$), where
 $lwork \geq n$ when $norm = 'I'$; otherwise, $work$ is not referenced.

Output Parameters

val REAL for slanhb/clanhb
 DOUBLE PRECISION for dlanhb/zlanhb
 Value returned by the function.

?lansb

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric band matrix.

Syntax

```
val = slansb( norm, uplo, n, k, ab, ldab, work )
val = dlansb( norm, uplo, n, k, ab, ldab, work )
val = clansb( norm, uplo, n, k, ab, ldab, work )
val = zlansb( norm, uplo, n, k, ab, ldab, work )
```

Description

The function ?lansb returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an n -by- n real/complex symmetric band matrix A , with k super-diagonals.

Input Parameters

<i>norm</i>	CHARACTER*1. Specifies the value to be returned by the routine: = 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix <i>A</i>. = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix <i>A</i> (maximum column sum), = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix <i>A</i> (maximum row sum), = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix <i>A</i> (square root of sum of squares).
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the band matrix <i>A</i> is supplied. If <i>uplo</i> = 'U': upper triangular part is supplied; If <i>uplo</i> = 'L': lower triangular part is supplied.
<i>n</i>	INTEGER. The order of the matrix <i>A</i>. $n \geq 0$. When $n = 0$, ?lansb is set to zero.
<i>k</i>	INTEGER. The number of super-diagonals or sub-diagonals of the band matrix <i>A</i>. $k \geq 0$.
<i>ab</i>	REAL for slansb DOUBLE PRECISION for dlansb COMPLEX for clansb COMPLEX*16 for zlansb Array, DIMENSION (<i>ldab</i>,<i>n</i>). The upper or lower triangle of the symmetric band matrix <i>A</i>, stored in the first <i>k</i>+1 rows of <i>ab</i>. The <i>j</i>-th column of <i>A</i> is stored in the <i>j</i>-th column of the array <i>ab</i> as follows: if <i>uplo</i> = 'U', $ab(k+1+i-j, j) = a(i, j)$ for $\max(1, j-k) \leq i \leq j$; if <i>uplo</i> = 'L', $ab(1+i-j, j) = a(i, j)$ for $j \leq i \leq \min(n, j+k)$.
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i>. $ldab \geq k+1$.

work REAL for slansb and clansb.
 DOUBLE PRECISION for dlansb and zlansb.
 Workspace array, DIMENSION (max(1, *lwork*)), where
lwork ≥ *n* when *norm* = 'I' or '1' or 'O'; otherwise,
work is not referenced.

Output Parameters

val REAL for slansb/clansb
 DOUBLE PRECISION for dlansb/zlansb
 Value returned by the function.

?lanhb

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a Hermitian band matrix.

Syntax

```
val = clanhb( norm, uplo, n, k, ab, ldab, work )
val = zlanhb( norm, uplo, n, k, ab, ldab, work )
```

Description

The routine returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an *n*-by-*n* Hermitian band matrix *A*, with *k* super-diagonals.

Input Parameters

norm CHARACTER*1. Specifies the value to be returned by the routine:
 = 'M' or 'm': *val* = max(abs(*A*_{*ij*})), largest absolute value of the matrix *A*.
 = '1' or 'O' or 'o': *val* = norm1(*A*), 1-norm of the matrix *A* (maximum column sum),
 = 'I' or 'i': *val* = normI(*A*), infinity norm of the matrix *A* (maximum row sum),

	= 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the band matrix A is supplied. If <i>uplo</i> = 'U': upper triangular part is supplied; If <i>uplo</i> = 'L': lower triangular part is supplied.
<i>n</i>	INTEGER. The order of the matrix A. $n \geq 0$. When $n = 0$, <i>lanhb</i> is set to zero.
<i>k</i>	INTEGER. The number of super-diagonals or sub-diagonals of the band matrix A. $k \geq 0$.
<i>ab</i>	COMPLEX for <i>clanhb</i>. COMPLEX*16 for <i>zlanhb</i>. Array, DIMENSION (<i>ldaB</i>,<i>n</i>). The upper or lower triangle of the Hermitian band matrix A, stored in the first $k+1$ rows of <i>ab</i>. The j-th column of A is stored in the j-th column of the array <i>ab</i> as follows: if <i>uplo</i> = 'U', $ab(k+1+i-j, j) = a(i, j)$ for $\max(1, j-k) \leq i \leq j$; if <i>uplo</i> = 'L', $ab(1+i-j, j) = a(i, j)$ for $j \leq i \leq \min(n, j+k)$. Note that the imaginary parts of the diagonal elements need not be set and are assumed to be zero.
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i>. $ldab \geq k+1$.
<i>work</i>	REAL for <i>clanhb</i>. DOUBLE PRECISION for <i>zlanhb</i>. Workspace array, DIMENSION $\max(1, lwork)$, where $lwork \geq n$ when <i>norm</i> = 'I' or '1' or 'O'; otherwise, <i>work</i> is not referenced.

Output Parameters

<i>val</i>	REAL for <i>slanhb</i> / <i>clanhb</i>

DOUBLE PRECISION for dlanhb/zlanhb
Value returned by the function.

?lanzp

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric matrix supplied in packed form.

Syntax

```
val = slansp( norm, uplo, n, ap, work )
val = dlansp( norm, uplo, n, ap, work )
val = clansp( norm, uplo, n, ap, work )
val = zlansp( norm, uplo, n, ap, work )
```

Description

The function ?lanzp returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex symmetric matrix *A*, supplied in packed form.

Input Parameters

<i>norm</i>	CHARACTER*1. Specifies the value to be returned by the routine: = 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix <i>A</i>. = '1' or 'O' or 'o': $val = \text{norml}(A)$, 1-norm of the matrix <i>A</i> (maximum column sum), = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix <i>A</i> (maximum row sum), = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix <i>A</i> (square root of sum of squares).
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix <i>A</i> is supplied. If <i>uplo</i> = 'U': Upper triangular part of <i>A</i> is supplied

If *uplo* = 'L': Lower triangular part of *A* is supplied.

n INTEGER. The order of the matrix *A*. $n \geq 0$. When $n = 0$, *zlan*sp is set to zero.

ap REAL for slansp
DOUBLE PRECISION for dlansp
COMPLEX for clansp
COMPLEX*16 for zlan

Array, DIMENSION $(n(n+1)/2)$.
The upper or lower triangle of the symmetric matrix *A*, packed columnwise in a linear array. The *j*-th column of *A* is stored in the array *ap* as follows:

if *uplo* = 'U', $ap(i + (j-1)j/2) = A(i, j)$ for $1 \leq i \leq j$;
if *uplo* = 'L', $ap(i + (j-1)(2n-j)/2) = A(i, j)$ for $j \leq i \leq n$.

work REAL for slansp and clansp.
DOUBLE PRECISION for dlansp and zlan

Workspace array, DIMENSION $(\max(1, lwork))$, where $lwork \geq n$ when *norm* = 'I' or '1' or 'O'; otherwise, *work* is not referenced.

Output Parameters

val REAL for slansp/clansp
DOUBLE PRECISION for dlansp/zlan

Value returned by the function.

?lanhp

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix supplied in packed form.

Syntax

```
val = clanhp( norm, uplo, n, ap, work )
val = zlanhp( norm, uplo, n, ap, work )
```

Description

The function ?lanhp returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix A , supplied in packed form.

Input Parameters

<i>norm</i>	CHARACTER*1. Specifies the value to be returned by the routine: = 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A. = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum), = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum), = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian matrix A is supplied. If <i>uplo</i> = 'U': Upper triangular part of A is supplied If <i>uplo</i> = 'L': Lower triangular part of A is supplied.
<i>n</i>	INTEGER. The order of the matrix A. $n \geq 0$. When $n = 0$, ?lanhp is set to zero.
<i>ap</i>	COMPLEX for clanhp. COMPLEX*16 for zlanhp.

Array, DIMENSION $(n(n+1)/2)$. The upper or lower triangle of the Hermitian matrix A , packed columnwise in a linear array. The j -th column of A is stored in the array ap as follows:

if $uplo = 'U'$, $ap(i + (j-1)j/2) = A(i, j)$ for $1 \leq i \leq j$;
 if $uplo = 'L'$, $ap(i + (j-1)(2n-j)/2) = A(i, j)$ for $j \leq i \leq n$.

work

REAL for `clanhp`.

DOUBLE PRECISION for `zlanhp`.

Workspace array, DIMENSION $(\max(1, lwork))$, where

$lwork \geq n$ when $norm = 'I'$ or $'1'$ or $'O'$; otherwise, *work* is not referenced.

Output Parameters

val

REAL for `clanhp`.

DOUBLE PRECISION for `zlanhp`.

Value returned by the function.

?lanst/?lanht

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real symmetric or complex Hermitian tridiagonal matrix.

Syntax

`val = slanst( norm, n, d, e )`

`val = dlanst( norm, n, d, e )`

`val = clanht( norm, n, d, e )`

`val = zlanht( norm, n, d, e )`

Description

The functions `?lanst/?lanht` return the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real symmetric or a complex Hermitian tridiagonal matrix A .

Input Parameters

<i>norm</i>	CHARACTER*1. Specifies the value to be returned by the routine: = 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A. = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum), = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum), = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<i>n</i>	INTEGER. The order of the matrix A. $n \geq 0$. When $n = 0$, <code>?lanst/?lanht</code> is set to zero.
<i>d</i>	REAL for <code>slanst/clanht</code> DOUBLE PRECISION for <code>dlanst/zlanht</code> Array, DIMENSION (n). The diagonal elements of A.
<i>e</i>	REAL for <code>slanst</code> DOUBLE PRECISION for <code>dlanst</code> COMPLEX for <code>clanht</code> COMPLEX*16 for <code>zlanht</code> Array, DIMENSION ($n-1$). The ($n-1$) sub-diagonal or super-diagonal elements of A.

Output Parameters

<i>val</i>	REAL for <code>slanst/clanht</code> DOUBLE PRECISION for <code>dlanst/zlanht</code> Value returned by the function.

?lansy

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex symmetric matrix.

Syntax

```
val = slansy( norm, uplo, n, a, lda, work )
val = dlansy( norm, uplo, n, a, lda, work )
val = clansy( norm, uplo, n, a, lda, work )
val = zlansy( norm, uplo, n, a, lda, work )
```

Description

The function ?lansy returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex symmetric matrix *A*.

Input Parameters

<i>norm</i>	CHARACTER*1. Specifies the value to be returned by the routine: = 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix <i>A</i>. = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix <i>A</i> (maximum column sum), = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix <i>A</i> (maximum row sum), = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix <i>A</i> (square root of sum of squares).
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix <i>A</i> is to be referenced. = 'U': Upper triangular part of <i>A</i> is referenced. = 'L': Lower triangular part of <i>A</i> is referenced
<i>n</i>	INTEGER. The order of the matrix <i>A</i>. $n \geq 0$. When $n = 0$, ?lansy is set to zero.

a REAL for slansy
 DOUBLE PRECISION for dlansy
 COMPLEX for clansy
 COMPLEX*16 for zlansy
 Array, DIMENSION (*lda*,*n*). The symmetric matrix *A*.
 If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced.
 If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda INTEGER. The leading dimension of the array *a*.
lda ≥ max(*n*,1).

work REAL for slansy and clansy.
 DOUBLE PRECISION for dlansy and zlansy.
 Workspace array, DIMENSION (max(1, *lwork*)), where
lwork ≥ *n* when *norm* = 'I' or '1' or 'O'; otherwise,
work is not referenced.

Output Parameters

val REAL for slansy/clansy
 DOUBLE PRECISION for dlansy/zlansy
 Value returned by the function.

?lanhe

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix.

Syntax

```
val = clanhe( norm, uplo, n, a, lda, work )
val = zlanhe( norm, uplo, n, a, lda, work )
```

Description

The function `?lanhe` returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix *A*.

Input Parameters

<i>norm</i>	CHARACTER*1. Specifies the value to be returned by the routine: = 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix <i>A</i>. = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix <i>A</i> (maximum column sum), = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix <i>A</i> (maximum row sum), = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix <i>A</i> (square root of sum of squares).
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian matrix <i>A</i> is to be referenced. = 'U': Upper triangular part of <i>A</i> is referenced. = 'L': Lower triangular part of <i>A</i> is referenced
<i>n</i>	INTEGER. The order of the matrix <i>A</i>. $n \geq 0$. When $n = 0$, <code>?lanhe</code> is set to zero.
<i>a</i>	COMPLEX for <code>clanhe</code>. COMPLEX*16 for <code>zlanhe</code>. Array, DIMENSION (<i>lda</i>,<i>n</i>). The Hermitian matrix <i>A</i>. If <i>uplo</i> = 'U', the leading <i>n</i>-by-<i>n</i> upper triangular part of <i>a</i> contains the upper triangular part of the matrix <i>A</i>, and the strictly lower triangular part of <i>a</i> is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i>-by-<i>n</i> lower triangular part of <i>a</i> contains the lower triangular part of the matrix <i>A</i>, and the strictly upper triangular part of <i>a</i> is not referenced.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i>. $lda \geq \max(n, 1)$.
<i>work</i>	REAL for <code>clanhe</code>. DOUBLE PRECISION for <code>zlanhe</code>.

Workspace array, DIMENSION (max(1, lwork)) , where
 $lwork \geq n$ when $norm = 'I'$ or $'1'$ or $'O'$; otherwise,
 $work$ is not referenced.

Output Parameters

val REAL for `clanhe`.
 DOUBLE PRECISION for `zlanhe`.
 Value returned by the function.

?lantb

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular band matrix.

Syntax

```
val = slantb( norm, uplo, diag, n, k, ab, ldab, work )
val = dlantb( norm, uplo, diag, n, k, ab, ldab, work )
val = clantb( norm, uplo, diag, n, k, ab, ldab, work )
val = zlantb( norm, uplo, diag, n, k, ab, ldab, work )
```

Description

The function `?lantb` returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an n -by- n triangular band matrix A , with $(k + 1)$ diagonals.

Input Parameters

$norm$ CHARACTER*1. Specifies the value to be returned by the routine:
 = 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A .
 = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum),
 = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum),

	= 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<i>uplo</i>	CHARACTER*1. Specifies whether the matrix A is upper or lower triangular. = 'U': Upper triangular = 'L': Lower triangular.
<i>diag</i>	CHARACTER*1. Specifies whether or not the matrix A is unit triangular. = 'N': Non-unit triangular = 'U': Unit triangular.
<i>n</i>	INTEGER. The order of the matrix A. $n \geq 0$. When $n = 0$, <i>?lantb</i> is set to zero.
<i>k</i>	INTEGER. The number of super-diagonals of the matrix A if <i>uplo</i> = 'U', or the number of sub-diagonals of the matrix A if <i>uplo</i> = 'L'. $k \geq 0$.
<i>ab</i>	REAL for <i>slantb</i> DOUBLE PRECISION for <i>dlantb</i> COMPLEX for <i>clantb</i> COMPLEX*16 for <i>zlantb</i> Array, DIMENSION (<i>ldab</i>,<i>n</i>). The upper or lower triangular band matrix A, stored in the first $k+1$ rows of <i>ab</i>. The j-th column of A is stored in the j-th column of the array <i>ab</i> as follows: if <i>uplo</i> = 'U', $ab(k+1+i-j, j) = a(i, j)$ for $\max(1, j-k) \leq i \leq j$; if <i>uplo</i> = 'L', $ab(1+i-j, j) = a(i, j)$ for $j \leq i \leq \min(n, j+k)$. Note that when <i>diag</i> = 'U', the elements of the array <i>ab</i> corresponding to the diagonal elements of the matrix A are not referenced, but are assumed to be one.
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i>. $ldab \geq k+1$.
<i>work</i>	REAL for <i>slantb</i> and <i>clantb</i>. DOUBLE PRECISION for <i>dlantb</i> and <i>zlantb</i>. Workspace array, DIMENSION ($\max(1, lwork)$), where

$lwork \geq n$ when $norm = 'I'$; otherwise, $work$ is not referenced.

Output Parameters

val REAL for slantb/clantb.
DOUBLE PRECISION for dlantb/zlantb.
Value returned by the function.

?lantp

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular matrix supplied in packed form.

Syntax

```
val = slantp( norm, uplo, diag, n, ap, work )
val = dlantp( norm, uplo, diag, n, ap, work )
val = clantp( norm, uplo, diag, n, ap, work )
val = zlantp( norm, uplo, diag, n, ap, work )
```

Description

The function ?lantp returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular matrix A , supplied in packed form.

Input Parameters

$norm$ CHARACTER*1. Specifies the value to be returned by the routine:
= 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A .
= '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum),
= 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum),

	= 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<i>uplo</i>	CHARACTER*1. Specifies whether the matrix A is upper or lower triangular. = 'U': Upper triangular = 'L': Lower triangular.
<i>diag</i>	CHARACTER*1. Specifies whether or not the matrix A is unit triangular. = 'N': Non-unit triangular = 'U': Unit triangular.
<i>n</i>	INTEGER. The order of the matrix A. $n \geq 0$. When $n = 0$, <i>slantp</i> is set to zero.
<i>ap</i>	REAL for <i>slantp</i> DOUBLE PRECISION for <i>dlantp</i> COMPLEX for <i>clantp</i> COMPLEX*16 for <i>zlantp</i> Array, DIMENSION $(n(n+1)/2)$. The upper or lower triangular matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array <i>ap</i> as follows: if <i>uplo</i> = 'U', $ap(i + (j-1)j/2) = a(i, j)$ for $1 \leq i \leq j$; if <i>uplo</i> = 'L', $ap(i + (j-1)(2n-j)/2) = a(i, j)$ for $j \leq i \leq n$. Note that when <i>diag</i> = 'U', the elements of the array <i>ap</i> corresponding to the diagonal elements of the matrix A are not referenced, but are assumed to be one.
<i>work</i>	REAL for <i>slantp</i> and <i>clantp</i>. DOUBLE PRECISION for <i>dlantp</i> and <i>zlantp</i>. Workspace array, DIMENSION $(\max(1, lwork))$, where $lwork \geq n$ when <i>norm</i> = 'I' ; otherwise, <i>work</i> is not referenced.

Output Parameters

<i>val</i>	REAL for <i>slantp</i> / <i>clantp</i> .

DOUBLE PRECISION for dlantr/zlantr.
Value returned by the function.

?lantr

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular matrix.

Syntax

```
val = slantr( norm, uplo, diag, m, n, a, lda, work )
val = dlantr( norm, uplo, diag, m, n, a, lda, work )
val = clantr( norm, uplo, diag, m, n, a, lda, work )
val = zlantr( norm, uplo, diag, m, n, a, lda, work )
```

Description

The function ?lantr returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular matrix *A*.

Input Parameters

<i>norm</i>	CHARACTER*1. Specifies the value to be returned by the routine: = 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix <i>A</i>. = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix <i>A</i> (maximum column sum), = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix <i>A</i> (maximum row sum), = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix <i>A</i> (square root of sum of squares).
<i>uplo</i>	CHARACTER*1. Specifies whether the matrix <i>A</i> is upper or lower trapezoidal. = 'U': Upper trapezoidal = 'L': Lower trapezoidal. Note that <i>A</i> is triangular instead of trapezoidal if $m = n$.

<i>diag</i>	CHARACTER*1. Specifies whether or not the matrix <i>A</i> has unit diagonal. = 'N': Non-unit diagonal = 'U': Unit diagonal.
<i>m</i>	INTEGER. The number of rows of the matrix <i>A</i>. $m \geq 0$, and if <i>uplo</i> = 'U', $m \leq n$. When $m = 0$, <i>?lantr</i> is set to zero.
<i>n</i>	INTEGER. The number of columns of the matrix <i>A</i>. $n \geq 0$, and if <i>uplo</i> = 'L', $n \leq m$. When $n = 0$, <i>?lantr</i> is set to zero.
<i>a</i>	REAL for <i>slantr</i> DOUBLE PRECISION for <i>dlantr</i> COMPLEX for <i>clantr</i> COMPLEX*16 for <i>zlantr</i> Array, DIMENSION (<i>lda</i>,<i>n</i>). The trapezoidal matrix <i>A</i> (<i>A</i> is triangular if $m = n$). If <i>uplo</i> = 'U', the leading <i>m</i>-by-<i>n</i> upper trapezoidal part of the array <i>a</i> contains the upper trapezoidal matrix, and the strictly lower triangular part of <i>A</i> is not referenced. If <i>uplo</i> = 'L', the leading <i>m</i>-by-<i>n</i> lower trapezoidal part of the array <i>a</i> contains the lower trapezoidal matrix, and the strictly upper triangular part of <i>A</i> is not referenced. Note that when <i>diag</i> = 'U', the diagonal elements of <i>A</i> are not referenced and are assumed to be one.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i>. $lda \geq \max(m, 1)$.
<i>work</i>	REAL for <i>slantr/clantrp</i>. DOUBLE PRECISION for <i>dlantr/zlantr</i>. Workspace array, DIMENSION ($\max(1, lwork)$), where $lwork \geq m$ when <i>norm</i> = 'I' ; otherwise, <i>work</i> is not referenced.

Output Parameters

<i>val</i>	REAL for <i>slantr/clantrp</i> .

DOUBLE PRECISION for dlantr/zlantr.
Value returned by the function.

?lanv2

Computes the Schur factorization of a real 2-by-2 nonsymmetric matrix in standard form.

Syntax

```
call slanv2( a, b, c, d, rt1r, rt1i, rt2r, rt2i, cs, sn )
call dlanv2( a, b, c, d, rt1r, rt1i, rt2r, rt2i, cs, sn )
```

Description

The routine computes the Schur factorization of a real 2-by-2 nonsymmetric matrix in standard form:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} cs & -sn \\ sn & cs \end{bmatrix} \begin{bmatrix} aa & bb \\ cc & dd \end{bmatrix} \begin{bmatrix} cs & sn \\ -sn & cs \end{bmatrix}$$

where either

- 1.** $cc = 0$ so that aa and dd are real eigenvalues of the matrix, or
- 2.** $aa = dd$ and $bb*cc < 0$, so that $aa \pm \sqrt{bb*cc}$ are complex conjugate eigenvalues.

The routine was adjusted to reduce the risk of cancellation errors, when computing real eigenvalues, and to ensure, if possible, that $abs(rt1r) \geq abs(rt2r)$.

Input Parameters

a, b, c, d REAL for slanv2
 DOUBLE PRECISION for dlanv2.
 On entry, elements of the input matrix.

Output Parameters

a, b, c, d On exit, overwritten by the elements of the standardized Schur form.

rt1r, rt1i, rt2r, rt2i REAL for slantv2
 DOUBLE PRECISION for dlanv2.
 The real and imaginary parts of the eigenvalues.
 If the eigenvalues are a complex conjugate pair, *rt1i* > 0.

cs, sn REAL for slantv2
 DOUBLE PRECISION for dlanv2.
 Parameters of the rotation matrix.

?lapll

Measures the linear dependence of two vectors.

Syntax

```
call slapll( n, x, incx, Y, incy, ssmín )
call dlapll( n, x, incx, Y, incy, ssmín )
call clapll( n, x, incx, Y, incy, ssmín )
call zlapll( n, x, incx, Y, incy, ssmín )
```

Description

Given two column vectors *x* and *y* of length *n*, let

$A = (x \ y)$ be the *n*-by-2 matrix.

The routine ?lapll first computes the *QR* factorization of *A* as $A = Q \cdot R$ and then computes the SVD of the 2-by-2 upper triangular matrix *R*. The smaller singular value of *R* is returned in *ssmin*, which is used as the measurement of the linear dependency of the vectors *x* and *y*.

Input Parameters

n INTEGER. The length of the vectors *x* and *y*.

x REAL for slapll
 DOUBLE PRECISION for dlapll
 COMPLEX for clapll
 COMPLEX*16 for zlapll
 Array, DIMENSION (1+(*n*-1) *incx*).
 On entry, *x* contains the *n*-vector *x*.

y REAL for slapll
 DOUBLE PRECISION for dlapll
 COMPLEX for clapll
 COMPLEX*16 for zlapll
 Array, DIMENSION (1+(*n*-1)*incy*).
 On entry, *y* contains the *n*-vector *y*.

incx INTEGER. The increment between successive elements of *x*; *incx* > 0.

incy INTEGER. The increment between successive elements of *y*; *incy* > 0.

Output Parameters

x On exit, *x* is overwritten.

y On exit, *y* is overwritten.

ssmin REAL for slapll/clapll
 DOUBLE PRECISION for dlapll/zlapll
 The smallest singular value of the *n*-by-2 matrix $A = \begin{pmatrix} x & y \end{pmatrix}$.

?lapmt

Performs a forward or backward permutation of the columns of a matrix.

Syntax

```
call slapmt( forwr, m, n, x, ldx, k )
call dlapmt( forwr, m, n, x, ldx, k )
call clapmt( forwr, m, n, x, ldx, k )
call zlapmt( forwr, m, n, x, ldx, k )
```

Description

The routine ?lapmt rearranges the columns of the *m*-by-*n* matrix *X* as specified by the permutation $k(1), k(2), \dots, k(n)$ of the integers $1, \dots, n$.

If *forwr* = .TRUE., forward permutation:

$X(*, k(j))$ is moved to $X(*, j)$ for $j=1, 2, \dots, n$.

If `forwrđ = .FALSE.`, backward permutation:

$X(*, j)$ is moved to $X(*, k(j))$ for $j = 1, 2, \dots, n$.

Input Parameters

<code>forwrđ</code>	LOGICAL. If <code>forwrđ = .TRUE.</code> , forward permutation If <code>forwrđ = .FALSE.</code> , backward permutation
<code>m</code>	INTEGER. The number of rows of the matrix X . $m \geq 0$.
<code>n</code>	INTEGER. The number of columns of the matrix X . $n \geq 0$.
<code>x</code>	REAL for <code>slapmt</code> DOUBLE PRECISION for <code>dlapmt</code> COMPLEX for <code>clapmt</code> COMPLEX*16 for <code>zlapmt</code> Array, DIMENSION (<code>ldx</code> , <code>n</code>). On entry, the m -by- n matrix X .
<code>ldx</code>	INTEGER. The leading dimension of the array X , $ldx \geq \max(1, m)$.
<code>k</code>	INTEGER. Array, DIMENSION (<code>n</code>). On entry, <code>k</code> contains the permutation vector and is used as internal workspace.

Output Parameters

<code>x</code>	On exit, <code>x</code> contains the permuted matrix X .
<code>k</code>	On exit, <code>k</code> is reset to its original value.

?lapy2

Returns $\sqrt{x^2 + y^2}$.

Syntax

`val = slap2( x, y )`

`val = dlapy2( x, y )`

Description

The function `?lapy2` returns $\sqrt{x^2 + y^2}$, avoiding unnecessary overflow or harmful underflow.

Input Parameters

x, y REAL for slapy2
DOUBLE PRECISION for dlapy2
Specify the input values x and y .

Output Parameters

val REAL for slapy2
DOUBLE PRECISION for dlapy2.
Value returned by the function.

?lapy3

Returns $\text{sqrt}(x^2+y^2+z^2)$.

Syntax

```
val = slapy3( x, y, z )  
val = dlapy3( x, y, z )
```

Description

The function ?lapy3 returns $\text{sqrt}(x^2+y^2+z^2)$, avoiding unnecessary overflow or harmful underflow.

Input Parameters

x, y, z REAL for slapy3
DOUBLE PRECISION for dlapy3
Specify the input values x, y and z .

Output Parameters

val REAL for slapy3
DOUBLE PRECISION for dlapy3.
Value returned by the function.

?laqgb

Scales a general band matrix, using row and column scaling factors computed by ?gbequ.

Syntax

```
call slaqgb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, equed )
call dlaqgb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, equed )
call claqgb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, equed )
call zlaqgb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, equed )
```

Description

The routine equilibrates a general m -by- n band matrix A with kl subdiagonals and ku superdiagonals using the row and column scaling factors in the vectors r and c .

Input Parameters

m	INTEGER. The number of rows of the matrix A . $m \geq 0$.
n	INTEGER. The number of columns of the matrix A . $n \geq 0$.
kl	INTEGER. The number of subdiagonals within the band of A . $kl \geq 0$.
ku	INTEGER. The number of superdiagonals within the band of A . $ku \geq 0$.
ab	REAL for slaqgb DOUBLE PRECISION for dlaqgb COMPLEX for claqgb COMPLEX*16 for zlaqgb Array, DIMENSION ($ldab, n$). On entry, the matrix A in band storage, in rows 1 to $kl+ku+1$. The j -th column of A is stored in the j -th column of the array ab as follows: $ab(ku+1+i-j, j) = A(i, j)$ for $\max(1, j-ku) \leq i \leq \min(m, j+kl)$.
$ldab$	INTEGER. The leading dimension of the array ab . $lda \geq kl+ku+1$.

<i>amax</i>	REAL for slaqgb/claqgb DOUBLE PRECISION for dlaqgb/zlaqgb Absolute value of largest matrix entry.
<i>r, c</i>	REAL for slaqgb/claqgb DOUBLE PRECISION for dlaqgb/zlaqgb Arrays $r(m)$, $c(n)$. Contain the row and column scale factors for A , respectively.
<i>rowcnd</i>	REAL for slaqgb/claqgb DOUBLE PRECISION for dlaqgb/zlaqgb Ratio of the smallest $r(i)$ to the largest $r(i)$.
<i>colcnd</i>	REAL for slaqgb/claqgb DOUBLE PRECISION for dlaqgb/zlaqgb Ratio of the smallest $c(i)$ to the largest $c(i)$.

Output Parameters

<i>ab</i>	On exit, the equilibrated matrix, in the same storage format as A . See <i>equed</i> for the form of the equilibrated matrix.
<i>equed</i>	CHARACTER*1. Specifies the form of equilibration that was done. If <i>equed</i> = 'N': No equilibration If <i>equed</i> = 'R': Row equilibration, that is, A has been premultiplied by $\text{diag}(r)$. If <i>equed</i> = 'C': Column equilibration, that is, A has been postmultiplied by $\text{diag}(c)$. If <i>equed</i> = 'B': Both row and column equilibration, that is, A has been replaced by $\text{diag}(r) * A * \text{diag}(c)$.

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if row or column scaling should be done based on the ratio of the row or column scaling factors. If $\text{rowcnd} < \text{thresh}$, row scaling is done, and if $\text{colcnd} < \text{thresh}$, column scaling is done. *large* and *small* are threshold values used to decide if row scaling should be done based on the absolute size of the largest matrix element. If $\text{amax} > \text{large}$ or $\text{amax} < \text{small}$, row scaling is done.

?laqge

Scales a general rectangular matrix, using row and column scaling factors computed by ?geequ.

Syntax

```
call slaqge( m, n, a, lda, r, c, rowcnd, colcnd, amax, equed )
call dlaqge( m, n, a, lda, r, c, rowcnd, colcnd, amax, equed )
call claqge( m, n, a, lda, r, c, rowcnd, colcnd, amax, equed )
call zlaqge( m, n, a, lda, r, c, rowcnd, colcnd, amax, equed )
```

Description

The routine equilibrates a general m -by- n matrix A using the row and column scaling factors in the vectors r and c .

Input Parameters

m	INTEGER. The number of rows of the matrix A . $m \geq 0$.
n	INTEGER. The number of columns of the matrix A . $n \geq 0$.
a	REAL for slaqge DOUBLE PRECISION for dlaqge COMPLEX for claqge COMPLEX*16 for zlaqge Array, DIMENSION (lda, n). On entry, the m -by- n matrix A .
lda	INTEGER. The leading dimension of the array a . $lda \geq \max(m, 1)$.
r	REAL for slangge/claqge DOUBLE PRECISION for dlaqge/zlaqge Array, DIMENSION (m). The row scale factors for A .
c	REAL for slangge/claqge DOUBLE PRECISION for dlaqge/zlaqge Array, DIMENSION (n). The column scale factors for A .
$rowcnd$	REAL for slangge/claqge

colcnd DOUBLE PRECISION for dlaqge/zlaqge
Ratio of the smallest $r(i)$ to the largest $r(i)$.

amax REAL for slangge/claqge
DOUBLE PRECISION for dlaqge/zlaqge
Ratio of the smallest $c(i)$ to the largest $c(i)$.

REAL for slangge/claqge
DOUBLE PRECISION for dlaqge/zlaqge
Absolute value of largest matrix entry.

Output Parameters

a On exit, the equilibrated matrix.
See *equed* for the form of the equilibrated matrix.

equed CHARACTER*1.
Specifies the form of equilibration that was done.
If *equed* = 'N': No equilibration
If *equed* = 'R': Row equilibration, that is, A has been premultiplied by $\text{diag}(r)$.
If *equed* = 'C': Column equilibration, that is, A has been postmultiplied by $\text{diag}(c)$.
If *equed* = 'B': Both row and column equilibration, that is, A has been replaced by $\text{diag}(r) * A * \text{diag}(c)$.

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if row or column scaling should be done based on the ratio of the row or column scaling factors. If *rowcnd* < *thresh*, row scaling is done, and if *colcnd* < *thresh*, column scaling is done. *large* and *small* are threshold values used to decide if row scaling should be done based on the absolute size of the largest matrix element. If *amax* > *large* or *amax* < *small*, row scaling is done.

?laqhb

Scales a Hermetian band matrix, using scaling factors computed by ?pbequ.

Syntax

```
call claqhb( uplo, n, kd, ab, ldab, s, scond, amax, equed )
```

```
call zlaqhb( uplo, n, kd, ab, ldab, s, scond, amax, equed )
```

Description

The routine equilibrates a Hermetian band matrix A using the scaling factors in the vector s .

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the band matrix A is stored. If <i>uplo</i> = 'U': upper triangular. If <i>uplo</i> = 'L': lower triangular.
<i>n</i>	INTEGER. The order of the matrix A . $n \geq 0$.
<i>kd</i>	INTEGER. The number of super-diagonals of the matrix A if <i>uplo</i> = 'U', or the number of sub-diagonals if <i>uplo</i> = 'L'. $kd \geq 0$.
<i>ab</i>	COMPLEX for claqhb COMPLEX*16 for zlaqhb Array, DIMENSION (<i>ldab</i> , <i>n</i>). On entry, the upper or lower triangle of the band matrix A , stored in the first $kd+1$ rows of the array. The j -th column of A is stored in the j -th column of the array <i>ab</i> as follows: if <i>uplo</i> = 'U', $ab(kd+1+i-j, j) = A(i, j)$ for $\max(1, j-kd) \leq i \leq j$; if <i>uplo</i> = 'L', $ab(1+i-j, j) = A(i, j)$ for $j \leq i \leq \min(n, j+kd)$.
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i> .

ldab $\geq$ *kd*+1.

scond REAL for `claqsb`
 DOUBLE PRECISION for `zlaqsb`
 Ratio of the smallest $s(i)$ to the largest $s(i)$.

amax REAL for `claqsb`
 DOUBLE PRECISION for `zlaqsb`
 Absolute value of largest matrix entry.

Output Parameters

ab On exit, if *info* = 0, the triangular factor *U* or *L* from the Cholesky factorization $A = U' * U$ or $A = L * L'$ of the band matrix *A*, in the same storage format as *A*.

s REAL for `claqsb`
 DOUBLE PRECISION for `zlaqsb`
 Array, DIMENSION (*n*). The scale factors for *A*.

equed CHARACTER*1.
 Specifies whether or not equilibration was done.
 If *equed* = 'N': No equilibration.
 If *equed* = 'Y': Equilibration was done, that is, *A* has been replaced by $\text{diag}(s) * A * \text{diag}(s)$.

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if scaling should be based on the ratio of the scaling factors. If *scond* < *thresh*, scaling is done.

The values *large* and *small* are threshold values used to decide if scaling should be done based on the absolute size of the largest matrix element. If *amax* > *large* or *amax* < *small*, scaling is done.

?laqp2

Computes a QR factorization with column pivoting of the matrix block.

Syntax

```
call slaqp2( m, n, offset, a, lda, jpvt, tau, vn1, vn2, work )
call dlaqp2( m, n, offset, a, lda, jpvt, tau, vn1, vn2, work )
call claqp2( m, n, offset, a, lda, jpvt, tau, vn1, vn2, work )
call zlaqp2( m, n, offset, a, lda, jpvt, tau, vn1, vn2, work )
```

Description

The routine computes a QR factorization with column pivoting of the block $A(offset+1:m, 1:n)$. The block $A(1:offset, 1:n)$ is accordingly pivoted, but not factorized.

Input Parameters

<i>m</i>	INTEGER. The number of rows of the matrix <i>A</i> . $m \geq 0$.
<i>n</i>	INTEGER. The number of columns of the matrix <i>A</i> . $n \geq 0$.
<i>offset</i>	INTEGER. The number of rows of the matrix <i>A</i> that must be pivoted but no factorized. $offset \geq 0$.
<i>a</i>	REAL for slaqp2 DOUBLE PRECISION for dlaqp2 COMPLEX for claqp2 COMPLEX*16 for zlaqp2 Array, DIMENSION (<i>lda</i> , <i>n</i>). On entry, the <i>m</i> -by- <i>n</i> matrix <i>A</i> .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $lda \geq \max(1, m)$.
<i>jpvt</i>	INTEGER. Array, DIMENSION (<i>n</i>). On entry, if $jpvt(i) \neq 0$, the <i>i</i> -th column of <i>A</i> is permuted to the front of A^*P (a leading column); if $jpvt(i) = 0$, the <i>i</i> -th column of <i>A</i> is a free column.
<i>vn1, vn2</i>	REAL for slaqp2/claqp2

work DOUBLE PRECISION for dlaqp2/zlaqp2
 Arrays, DIMENSION (n) each. Contain the vectors with the partial and exact column norms, respectively.

REAL for slaqp2
 DOUBLE PRECISION for dlaqp2
 COMPLEX for claqp2
 COMPLEX*16 for zlaqp2 Workspace array, DIMENSION (n).

Output Parameters

a On exit, the upper triangle of block $A(offset+1:m, 1:n)$ is the triangular factor obtained; the elements in block $A(offset+1:m, 1:n)$ below the diagonal, together with the array *tau*, represent the orthogonal matrix Q as a product of elementary reflectors. Block $A(1:offset, 1:n)$ has been accordingly pivoted, but not factorized.

jpvt On exit, if $jpvt(i) = k$, then the i -th column of A^*P was the k -th column of A .

tau REAL for slaqp2
 DOUBLE PRECISION for dlaqp2
 COMPLEX for claqp2
 COMPLEX*16 for zlaqp2
 Array, DIMENSION ($\min(m, n)$).
 The scalar factors of the elementary reflectors.

vn1, vn2 Contain the vectors with the partial and exact column norms, respectively.

?laqps

Computes a step of QR factorization with column pivoting of a real m -by- n matrix A by using BLAS level 3.

Syntax

```
call slaqps( m, n, offset, nb, kb, a, lda, jpvt, tau, vn1, vn2, auxv, f, ldf
)
call dlaqps( m, n, offset, nb, kb, a, lda, jpvt, tau, vn1, vn2, auxv, f, ldf
)
call claqps( m, n, offset, nb, kb, a, lda, jpvt, tau, vn1, vn2, auxv, f, ldf
)
call zlaqps( m, n, offset, nb, kb, a, lda, jpvt, tau, vn1, vn2, auxv, f, ldf
)
```

Description

This routine computes a step of QR factorization with column pivoting of a real m -by- n matrix A by using BLAS level 3. The routine tries to factorize NB columns from A starting from the row $offset+1$, and updates all of the matrix with BLAS level 3 routine ?gemm.

In some cases, due to catastrophic cancellations, ?laqps cannot factorize NB columns. Hence, the actual number of factorized columns is returned in kb .

Block $A(1:offset, 1:n)$ is accordingly pivoted, but not factorized.

Input Parameters

m	INTEGER. The number of rows of the matrix A . $m \geq 0$.
n	INTEGER. The number of columns of the matrix A . $n \geq 0$.
$offset$	INTEGER. The number of rows of A that have been factorized in previous steps.
nb	INTEGER. The number of columns to factorize.
a	REAL for slaqps DOUBLE PRECISION for dlaqps COMPLEX for claqps COMPLEX*16 for zlaqps

	Array, DIMENSION (lda, n). On entry, the m -by- n matrix A .
lda	INTEGER. The leading dimension of the array a . $lda \geq \max(1, m)$.
$jpvt$	INTEGER. Array, DIMENSION (n). If $jpvt(i) = k$ then column k of the full matrix A has been permuted into position i in AP.
$vn1, vn2$	REAL for slaqps/claqps DOUBLE PRECISION for dlaqps/zlaqps Arrays, DIMENSION (n) each. Contain the vectors with the partial and exact column norms, respectively.
$auxv$	REAL for slaqps DOUBLE PRECISION for dlaqps COMPLEX for claqps COMPLEX*16 for zlaqps Array, DIMENSION (nb). Auxiliary vector.
f	REAL for slaqps DOUBLE PRECISION for dlaqps COMPLEX for claqps COMPLEX*16 for zlaqps Array, DIMENSION (ldf, nb). Matrix $F' = L^* Y'^* A$.
ldf	INTEGER. The leading dimension of the array f . $ldf \geq \max(1, n)$.

Output Parameters

kb	INTEGER. The number of columns actually factorized.
a	On exit, block $A(offset+1:m, 1:kb)$ is the triangular factor obtained and block $A(1:offset, 1:n)$ has been accordingly pivoted, but no factorized. The rest of the matrix, block $A(offset+1:m, kb+1:n)$ has been updated.
$jpvt$	INTEGER array, DIMENSION (n). If $jpvt(i) = k$ then column k of the full matrix A has been permuted into position i in AP.
τ	REAL for slaqps DOUBLE PRECISION for dlaqps

	COMPLEX for claqps
	COMPLEX*16 for zlaqps
	Array, DIMENSION (kb). The scalar factors of the elementary reflectors.
$vn1, vn2$	The vectors with the partial and exact column norms, respectively.
$auxv$	Auxiliary vector.
f	Matrix $F' = L*Y'*A$.

?laqr0

Computes the eigenvalues of a Hessenberg matrix, and optionally the matrices from the Schur decomposition.

Syntax

```
call slaqr0( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi, iloz, ihiz, z, ldz,
work, lwork, info )

call dlaqr0( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi, iloz, ihiz, z, ldz,
work, lwork, info )

call claqr0( wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, work,
lwork, info )

call zlaqr0( wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, work,
lwork, info )
```

Description

This routine computes the eigenvalues of a Hessenberg matrix H , and, optionally, the matrices T and Z from the Schur decomposition $H=Z*T*Z^H$, where T is an upper quasi-triangular/triangular matrix (the Schur form), and Z is the orthogonal/unitary matrix of Schur vectors.

Optionally Z may be postmultiplied into an input orthogonal/unitary matrix Q so that this routine can give the Schur factorization of a matrix A which has been reduced to the Hessenberg form H by the orthogonal/unitary matrix Q : $A = Q*H*Q^H = (QZ)*H*(QZ)^H$.

Input Parameters

$wantt$	LOGICAL.

If *wantt* = .TRUE., the full Schur form *T* is required;
If *wantt* = .FALSE., only eigenvalues are required.

wantz LOGICAL.
If *wantz* = .TRUE., the matrix of Schur vectors *Z* is required;
If *wantz* = .FALSE., Schur vectors are not required.

n INTEGER. The order of the Hessenberg matrix *H*. ($n \geq 0$).

ilo, ihi INTEGER.
 It is assumed that *H* is already upper triangular in rows and columns 1:*ilo*-1 and *ihi*+1:*n*, and if *ilo* > 1 then $H(i\text{lo}, i\text{lo}-1) = 0$.
ilo and *ihi* are normally set by a previous call to *cgebal*, and then passed to *cgehrd* when the matrix output by *cgebal* is reduced to Hessenberg form. Otherwise, *ilo* and *ihi* should be set to 1 and *n*, respectively.
If $n > 0$, then $1 \leq ilo \leq ihi \leq n$.
If $n=0$, then $ilo=1$ and $ihi=0$

h REAL for slaqr0
 DOUBLE PRECISION for dlaqr0
 COMPLEX for claqr0
 COMPLEX*16 for zlaqr0.
Array, DIMENSION (*ldh*, *n*), contains the upper Hessenberg matrix *H*.

ldh INTEGER. The leading dimension of the array *h*. $ldh \geq \max(1, n)$.

iloz, ihiz INTEGER. Specify the rows of *Z* to which transformations must be applied if *wantz* is .TRUE., $1 \leq iloz \leq ilo$; $ihi \leq ihiz \leq n$.

z REAL for slaqr0
 DOUBLE PRECISION for dlaqr0
 COMPLEX for claqr0
 COMPLEX*16 for zlaqr0.
Array, DIMENSION (*ldz*, *ihi*), contains the matrix *Z* if *wantz* is .TRUE.. If *wantz* is .FALSE., *z* is not referenced.

ldz INTEGER. The leading dimension of the array *z*.

If *wantz* is `.TRUE.`, then $ldz \geq \max(1, ihi_z)$. Otherwise, $ldz \geq 1$.

work REAL for `slaqr0`
 DOUBLE PRECISION for `dlaqr0`
 COMPLEX for `claqr0`
 COMPLEX*16 for `zlaqr0`.
 Workspace array with dimension *lwork*.

lwork INTEGER. The dimension of the array *work*.
 $lwork \geq \max(1, n)$ is sufficient, but for the optimal performance a greater workspace may be required, typically as large as $6*n$.
 It is recommended to use the workspace query to determine the optimal workspace size. If *lwork*=-1, then the routine performs a workspace query: it estimates the optimal workspace size for the given values of the input parameters *n*, *ilo*, and *ihi*. The estimate is returned in *work*(1). No error messages related to the *lwork* is issued by `xerbla`. Neither *H* nor *Z* are accessed.

Output Parameters

h If *info*=0, and *wantt* is `.TRUE.`, then *h* contains the upper quasi-triangular/triangular matrix *T* from the Schur decomposition (the Schur form).
 If *info*=0, and *wantt* is `.FALSE.`, then the contents of *h* are unspecified on exit.
 (The output values of *h* when *info* > 0 are given under the description of the *info* parameter below.)
 The routine may explicitly set *h*(*i*,*j*) for *i*>*j* and *j*=1,2,...*ilo*-1 or *j*=*ihi*+1, *ihi*+2,...*n*.

work(1) On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

w COMPLEX for `claqr0`
 COMPLEX*16 for `zlaqr0`.

Arrays, `DIMENSION(n)`. The computed eigenvalues of $h(i_{lo}:i_{hi}, i_{lo}:i_{hi})$ are stored in $w(i_{lo}:i_{hi})$. If `wantt` is `.TRUE.`, then the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in h , with $w(i) = h(i, i)$.

`wr, wi` REAL for `slaqr0`
DOUBLE PRECISION for `dlaqr0`
Arrays, `DIMENSION(ihi)` each. The real and imaginary parts, respectively, of the computed eigenvalues of $h(i_{lo}:i_{hi}, i_{lo}:i_{hi})$ are stored in the $wr(i_{lo}:i_{hi})$ and $wi(i_{lo}:i_{hi})$. If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of wr and wi , say the i -th and $(i+1)$ -th, with $wi(i) > 0$ and $wi(i+1) < 0$. If `wantt` is `.TRUE.`, then the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in h , with $wr(i) = h(i, i)$, and if $h(i:i+1, i:i+1)$ is a 2-by-2 diagonal block, then $wi(i) = \text{sqrt}(-h(i+1, i) * h(i, i+1))$.

`z` If `wantz` is `.TRUE.`, then $z(i_{lo}:i_{hi}, i_{loz}:i_{hiz})$ is replaced by $z(i_{lo}:i_{hi}, i_{loz}:i_{hiz}) * U$, where U is the orthogonal/unitary Schur factor of $h(i_{lo}:i_{hi}, i_{lo}:i_{hi})$. If `wantz` is `.FALSE.`, z is not referenced.
(The output values of z when `info` > 0 are given under the description of the `info` parameter below.)

`info` INTEGER.
= 0: the execution is successful.
> 0: if `info` = i , then the routine failed to compute all the eigenvalues. Elements $1:i_{lo}-1$ and $i+1:n$ of wr and wi contain those eigenvalues which have been successfully computed.
> 0: if `wantt` is `.FALSE.`, then the remaining unconverged eigenvalues are the eigenvalues of the upper Hessenberg matrix rows and columns i_{lo} through `info` of the final output value of h .

```

> 0: if wantt is .TRUE., then (initial value of  $h$ )* $U$  =
 $U$ *(final value of  $h$ , where  $U$  is an orthogonal/unitary
matrix. The final value of  $h$  is upper Hessenberg and
quasi-triangular/triangular in rows and columns info+1
through ihi.
> 0: if wantz is .TRUE., then (final value of  $z$ (ilo:ihi,
ilo:ihiz))=(initial value of  $z$ (ilo:ihi, ilo:ihiz)* $U$ ,
where  $U$  is the orthogonal/unitary matrix in the previous
expression (regardless of the value of wantt).
> 0: if wantz is .FALSE., then  $z$  is not accessed.

```

1qr?

Sets a scalar multiple of the first column of the product of 2-by-2 or 3-by-3 matrix $\mathbb{H}$ and specified shifts.

Syntax

```
call slaqr1( n, h, ldh, srl, si1, sr2, si2, v )
call dlaqr1( n, h, ldh, srl, si1, sr2, si2, v )
call claqr1( n, h, ldh, s1, s2, v )
call zlaqr1( n, h, ldh, s1, s2, v )
```

Description

Given a 2-by-2 or 3-by-3 matrix H , this routine sets v to a scalar multiple of the first column of the product

$$K = (H - s1*I) * (H - s2*I), \text{ or } K = (H - (sr1 + i*si1)*I) * (H - (sr2 + i*si2)*I)$$

scaling to avoid overflows and most underflows.

It is assumed that either 1) $sr1 = sr2$ and $si1 = -si2$, or 2) $si1 = si2 = 0$.

This is useful for starting double implicit shift bulges in the QR algorithm.

Input Parameters

n INTEGER.

The order of the matrix H . n must be equal to 2 or 3.

<i>sr1, si2, sr2, si2</i>	REAL for slaqr1 DOUBLE PRECISION for dlaqr1 Shift values that define K in the formula above.
<i>s1, s2</i>	COMPLEX for claqr1 COMPLEX*16 for zlaqr1. Shift values that define K in the formula above.
<i>h</i>	REAL for slaqr1 DOUBLE PRECISION for dlaqr1 COMPLEX for claqr1 COMPLEX*16 for zlaqr1. Array, DIMENSION (<i>ldh</i> , <i>n</i>), contains 2-by-2 or 3-by-3 matrix H in the formula above.
<i>ldh</i>	INTEGER. The leading dimension of the array <i>h</i> just as declared in the calling routine. $ldh \geq n$.

Output Parameters

<i>v</i>	REAL for slaqr1 DOUBLE PRECISION for dlaqr1 COMPLEX for claqr1 COMPLEX*16 for zlaqr1. Array with dimension (<i>n</i>). A scalar multiple of the first column of the matrix K in the formula above.

?laqr2

Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).

Syntax

```
call slaqr2( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns,
nd, sr, si, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )

call dlaqr2( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns,
nd, sr, si, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )

call claqr2( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns,
nd, sh, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )

call zlaqr2( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns,
nd, sh, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )
```

Description

This routine accepts as input an upper Hessenberg matrix H and performs an orthogonal/unitary similarity transformation designed to detect and deflate fully converged eigenvalues from a trailing principal submatrix. On output H has been overwritten by a new Hessenberg matrix that is a perturbation of an orthogonal/unitary similarity transformation of H . It is to be hoped that the final version of H has many zero subdiagonal entries.

This subroutine is identical to ?laqr3 except that it avoids recursion by calling ?lahqr instead of ?laqr4.

Input Parameters

<code>wantt</code>	LOGICAL. If <code>wantt = .TRUE.</code> , then the Hessenberg matrix H is fully updated so that the quasi-triangular/triangular Schur factor may be computed (in cooperation with the calling subroutine). If <code>wantt = .FALSE.</code> , then only enough of H is updated to preserve the eigenvalues.
<code>wantz</code>	LOGICAL.

If *wantz* = *.TRUE.*, then the orthogonal/unitary matrix *z* is updated so that the orthogonal/unitary Schur factor may be computed (in cooperation with the calling subroutine). If *wantz* = *.FALSE.*, then *z* is not referenced.

n INTEGER. The order of the Hessenberg matrix *H* and (if *wantz* = *.TRUE.*) the order of the orthogonal/unitary matrix *Z*.

ktop INTEGER.
It is assumed that either *ktop*=1 or *h*(*ktop*,*ktop*-1)=0. *ktop* and *kbot* together determine an isolated block along the diagonal of the Hessenberg matrix.

kbot INTEGER.
It is assumed without a check that either *kbot*=*n* or *h*(*kbot*+1,*kbot*)=0. *ktop* and *kbot* together determine an isolated block along the diagonal of the Hessenberg matrix.

nw INTEGER.
Size of the deflation window. $1 \leq nw \leq (kbot-ktop+1)$.

h REAL for slaqr2
DOUBLE PRECISION for dlaqr2
COMPLEX for claqr2
COMPLEX*16 for zlaqr2.
Array, DIMENSION (*ldh*, *n*), on input the initial *n*-by-*n* section of *h* stores the Hessenberg matrix *H* undergoing aggressive early deflation.

ldh INTEGER. The leading dimension of the array *h* just as declared in the calling subroutine. *ldh* ≥ *n*.

iloz, *ihiz* INTEGER. Specify the rows of *z* to which transformations must be applied if *wantz* is *.TRUE.*. $1 \leq iloz \leq ihiz \leq n$.

z REAL for slaqr2
DOUBLE PRECISION for dlaqr2
COMPLEX for claqr2
COMPLEX*16 for zlaqr2.
Array, DIMENSION (*ldz*, *ihi*), contains the matrix *z* if *wantz* is *.TRUE.*. If *wantz* is *.FALSE.*, then *z* is not referenced.

<i>ldz</i>	INTEGER. The leading dimension of the array <i>z</i> just as declared in the calling subroutine. $ldz \geq 1$.
<i>v</i>	REAL for slaqr2 DOUBLE PRECISION for dlaqr2 COMPLEX for claqr2 COMPLEX*16 for zlaqr2. Workspace array with dimension (ldv, nw) . An <i>nw</i> -by- <i>nw</i> work array.
<i>ldv</i>	INTEGER. The leading dimension of the array <i>v</i> just as declared in the calling subroutine. $ldv \geq nw$.
<i>nh</i>	INTEGER. The number of column of <i>t</i> . $nh \geq nw$.
<i>t</i>	REAL for slaqr2 DOUBLE PRECISION for dlaqr2 COMPLEX for claqr2 COMPLEX*16 for zlaqr2. Workspace array with dimension (ldt, nw) .
<i>ldt</i>	INTEGER. The leading dimension of the array <i>t</i> just as declared in the calling subroutine. $ldt \geq nw$.
<i>nv</i>	INTEGER. The number of rows of work array <i>wv</i> available for workspace. $nv \geq nw$.
<i>wv</i>	REAL for slaqr2 DOUBLE PRECISION for dlaqr2 COMPLEX for claqr2 COMPLEX*16 for zlaqr2. Workspace array with dimension $(ldwv, nw)$.
<i>ldwv</i>	INTEGER. The leading dimension of the array <i>wv</i> just as declared in the calling subroutine. $ldwv \geq nw$.
<i>work</i>	REAL for slaqr2 DOUBLE PRECISION for dlaqr2 COMPLEX for claqr2 COMPLEX*16 for zlaqr2. Workspace array with dimension <i>lwork</i> .
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> .

$lwork=2*nw$) is sufficient, but for the optimal performance a greater workspace may be required.

If $lwork=-1$, then the routine performs a workspace query: it estimates the optimal workspace size for the given values of the input parameters n , nw , $ktop$, and $kbot$. The estimate is returned in $work(1)$. No error messages related to the $lwork$ is issued by `xerbla`. Neither H nor Z are accessed.

Output Parameters

h	On output h has been transformed by an orthogonal/unitary similarity transformation, perturbed, and the returned to Hessenberg form that (it is to be hoped) has some zero subdiagonal entries.
$work(1)$	On exit $work(1)$ is set to an estimate of the optimal value of $lwork$ for the given values of the input parameters n , nw , $ktop$, and $kbot$.
z	If $wantz$ is <code>.TRUE.</code> , then the orthogonal/unitary similarity transformation is accumulated into $z(ihoz:ihiz, ilo:ihi)$ from the right. If $wantz$ is <code>.FALSE.</code> , then z is unreferenced.
nd	INTEGER. The number of converged eigenvalues uncovered by the routine.
ns	INTEGER. The number of unconverged, that is approximate eigenvalues returned in sr , si or in sh that may be used as shifts by the calling subroutine.
sh	COMPLEX for <code>claqr2</code> COMPLEX*16 for <code>zlaqr2</code> . Arrays, DIMENSION ($kbot$). The approximate eigenvalues that may be used for shifts are stored in the $sh(kbot-nd-ns+1)$ through the $sh(kbot-nd)$. The converged eigenvalues are stored in the $sh(kbot-nd+1)$ through the $sh(kbot)$.
sr, si	REAL for <code>slaqr2</code> DOUBLE PRECISION for <code>dlaqr2</code> Arrays, DIMENSION ($kbot$) each.

The real and imaginary parts of the approximate eigenvalues that may be used for shifts are stored in the $sr(kbot-nd-ns+1)$ through the $sr(kbot-nd)$, and $si(kbot-nd-ns+1)$ through the $si(kbot-nd)$, respectively. The real and imaginary parts of converged eigenvalues are stored in the $sr(kbot-nd+1)$ through the $sr(kbot)$, and $si(kbot-nd+1)$ through the $si(kbot)$, respectively.

?laqr3

Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).

Syntax

```
call slaqr3( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns,
nd, sr, si, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )

call dlaqr3( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns,
nd, sr, si, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )

call claqr3( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns,
nd, sh, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )

call zlaqr3( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns,
nd, sh, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )
```

Description

This routine accepts as input an upper Hessenberg matrix H and performs an orthogonal/unitary similarity transformation designed to detect and deflate fully converged eigenvalues from a trailing principal submatrix. On output H has been overwritten by a new Hessenberg matrix that is a perturbation of an orthogonal/unitary similarity transformation of H . It is to be hoped that the final version of H has many zero subdiagonal entries.

Input Parameters

`wantt` LOGICAL.

If *wantt* = `.TRUE.`, then the Hessenberg matrix *H* is fully updated so that the quasi-triangular/triangular Schur factor may be computed (in cooperation with the calling subroutine).

If *wantt* = `.FALSE.`, then only enough of *H* is updated to preserve the eigenvalues.

wantz LOGICAL.
If *wantz* = `.TRUE.`, then the orthogonal/unitary matrix *Z* is updated so that the orthogonal/unitary Schur factor may be computed (in cooperation with the calling subroutine).
If *wantz* = `.FALSE.`, then *Z* is not referenced.

n INTEGER. The order of the Hessenberg matrix *H* and (if *wantz* = `.TRUE.`) the order of the orthogonal/unitary matrix *Z*.

ktop INTEGER.
It is assumed that either *ktop*=1 or *h*(*ktop*,*ktop*-1)=0. *ktop* and *kbot* together determine an isolated block along the diagonal of the Hessenberg matrix.

kbot INTEGER.
It is assumed without a check that either *kbot*=*n* or *h*(*kbot*+1,*kbot*)=0. *ktop* and *kbot* together determine an isolated block along the diagonal of the Hessenberg matrix.

nw INTEGER.
Size of the deflation window. $1 \leq nw \leq (kbot - ktop + 1)$.

h REAL for slaqr3
DOUBLE PRECISION for dlaqr3
COMPLEX for claqr3
COMPLEX*16 for zlaqr3.
Array, DIMENSION (*ldh*, *n*), on input the initial *n*-by-*n* section of *h* stores the Hessenberg matrix *H* undergoing aggressive early deflation.

ldh INTEGER. The leading dimension of the array *h* just as declared in the calling subroutine. *ldh* ≥ *n*.

iloz, *ihiz* INTEGER. Specify the rows of *Z* to which transformations must be applied if *wantz* is `.TRUE.`. $1 \leq iloz \leq ihiz \leq n$.

<i>z</i>	REAL for slaqr3 DOUBLE PRECISION for dlaqr3 COMPLEX for claqr3 COMPLEX*16 for zlaqr3. Array, DIMENSION (<i>ldz</i>, <i>ihi</i>), contains the matrix <i>z</i> if <i>wantz</i> is .TRUE.. If <i>wantz</i> is .FALSE., then <i>z</i> is not referenced.
<i>ldz</i>	INTEGER. The leading dimension of the array <i>z</i> just as declared in the calling subroutine. <i>ldz</i>≥1.
<i>v</i>	REAL for slaqr3 DOUBLE PRECISION for dlaqr3 COMPLEX for claqr3 COMPLEX*16 for zlaqr3. Workspace array with dimension (<i>ldv</i>, <i>nw</i>). An <i>nw</i>-by-<i>nw</i> work array.
<i>ldv</i>	INTEGER. The leading dimension of the array <i>v</i> just as declared in the calling subroutine. <i>ldv</i>≥<i>nw</i>.
<i>nh</i>	INTEGER. The number of column of <i>t</i>. <i>nh</i>≥<i>nw</i>.
<i>t</i>	REAL for slaqr3 DOUBLE PRECISION for dlaqr3 COMPLEX for claqr3 COMPLEX*16 for zlaqr3. Workspace array with dimension (<i>ldt</i>, <i>nw</i>).
<i>ldt</i>	INTEGER. The leading dimension of the array <i>t</i> just as declared in the calling subroutine. <i>ldt</i>≥<i>nw</i>.
<i>nv</i>	INTEGER. The number of rows of work array <i>wv</i> available for workspace. <i>nv</i>≥<i>nw</i>.
<i>wv</i>	REAL for slaqr3 DOUBLE PRECISION for dlaqr3 COMPLEX for claqr3 COMPLEX*16 for zlaqr3. Workspace array with dimension (<i>ldwv</i>, <i>nw</i>).
<i>ldwv</i>	INTEGER. The leading dimension of the array <i>wv</i> just as declared in the calling subroutine. <i>ldwv</i>≥<i>nw</i>.

work REAL for slaqr3
 DOUBLE PRECISION for dlaqr3
 COMPLEX for claqr3
 COMPLEX*16 for zlaqr3.
 Workspace array with dimension *lwork*.

lwork INTEGER. The dimension of the array *work*.
lwork=2**nw*) is sufficient, but for the optimal performance
 a greater workspace may be required.
 If *lwork*=-1, then the routine performs a workspace query:
 it estimates the optimal workspace size for the given values
 of the input parameters *n*, *nw*, *ktop*, and *kbot*. The estimate
 is returned in *work*(1). No error messages related to the
lwork is issued by xerbla. Neither *H* nor *Z* are accessed.

Output Parameters

h On output *h* has been transformed by an orthogonal/unitary
 similarity transformation, perturbed, and the returned to
 Hessenberg form that (it is to be hoped) has some zero
 subdiagonal entries.

work(1) On exit *work*(1) is set to an estimate of the optimal value
 of *lwork* for the given values of the input parameters *n*,
nw, *ktop*, and *kbot*.

z If *wantz* is .TRUE., then the orthogonal/unitary similarity
 transformation is accumulated into *z*(*iloz:ihiz*, *ilo:ihi*)
 from the right.
 If *wantz* is .FALSE., then *z* is unreferenced.

nd INTEGER. The number of converged eigenvalues uncovered
 by the routine.

ns INTEGER. The number of unconverged, that is approximate
 eigenvalues returned in *sr*, *si* or in *sh* that may be used
 as shifts by the calling subroutine.

sh COMPLEX for claqr3
 COMPLEX*16 for zlaqr3.
 Arrays, DIMENSION (*kbot*).

The approximate eigenvalues that may be used for shifts are stored in the `sh(kbot-nd-ns+1)` through the `sh(kbot-nd)`.

The converged eigenvalues are stored in the `sh(kbot-nd+1)` through the `sh(kbot)`.

`sr, si` REAL for `slaqr3`
 DOUBLE PRECISION for `dlaqr3`
 Arrays, DIMENSION (`kbot`) each.

The real and imaginary parts of the approximate eigenvalues that may be used for shifts are stored in the `sr(kbot-nd-ns+1)` through the `sr(kbot-nd)`, and `si(kbot-nd-ns+1)` through the `si(kbot-nd)`, respectively.

The real and imaginary parts of converged eigenvalues are stored in the `sr(kbot-nd+1)` through the `sr(kbot)`, and `si(kbot-nd+1)` through the `si(kbot)`, respectively.

?laqr4

Computes the eigenvalues of a Hessenberg matrix, and optionally the matrices from the Schur decomposition.

Syntax

```
call slaqr4( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi, iloz, ihiz, z, ldz,
work, lwork, info )

call dlaqr4( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi, iloz, ihiz, z, ldz,
work, lwork, info )

call claqr4( wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, work,
lwork, info )

call zlaqr4( wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, work,
lwork, info )
```

Description

This routine computes the eigenvalues of a Hessenberg matrix H , and, optionally, the matrices T and Z from the Schur decomposition $H = Z^* T^* Z^H$, where T is an upper quasi-triangular/triangular matrix (the Schur form), and Z is the orthogonal/unitary matrix of Schur vectors.

Optionally Z may be postmultiplied into an input orthogonal/unitary matrix Q so that this routine can give the Schur factorization of a matrix A which has been reduced to the Hessenberg form H by the orthogonal/unitary matrix Q : $A = Q^* H^* Q^H = (QZ)^* H^* (QZ)^H$.

This routine implements one level of recursion for `?laqr0`. It is a complete implementation of the small bulge multi-shift QR algorithm. It may be called by `?laqr0` and, for large enough deflation window size, it may be called by `?laqr3`. This routine is identical to `?laqr0` except that it calls `?laqr2` instead of `?laqr3`.

Input Parameters

wantt LOGICAL.
If *wantt* = .TRUE., the full Schur form T is required;
If *wantt* = .FALSE., only eigenvalues are required.

wantz LOGICAL.
If *wantz* = .TRUE., the matrix of Schur vectors Z is required;
If *wantz* = .FALSE., Schur vectors are not required.

n INTEGER. The order of the Hessenberg matrix H . ($n \geq 0$).

ilo, ihi INTEGER.
It is assumed that H is already upper triangular in rows and columns $1:ilo-1$ and $ihi+1:n$, and if $ilo > 1$ then $h(ilo, ilo-1) = 0$.
ilo and *ihi* are normally set by a previous call to `cgebal`, and then passed to `cgehrd` when the matrix output by `cgebal` is reduced to Hessenberg form. Otherwise, *ilo* and *ihi* should be set to 1 and n , respectively.
If $n > 0$, then $1 \leq ilo \leq ihi \leq n$.
If $n=0$, then $ilo=1$ and $ihi=0$

h REAL for `slaqr4`
DOUBLE PRECISION for `dlaqr4`
COMPLEX for `claqr4`
COMPLEX*16 for `zlaqr4`.
Array, DIMENSION (*ldh*, n), contains the upper Hessenberg matrix H .

ldh INTEGER. The leading dimension of the array h . $ldh \geq \max(1, n)$.

<i>iloz, ihiz</i>	INTEGER. Specify the rows of <i>z</i> to which transformations must be applied if <i>wantz</i> is <code>.TRUE.</code>, $1 \leq iloz \leq ilo$; $ihiz \leq ihiz \leq n$.
<i>z</i>	REAL for slaqr4 DOUBLE PRECISION for dlaqr4 COMPLEX for claqr4 COMPLEX*16 for zlaqr4. Array, DIMENSION (<i>ldz</i>, <i>ihi</i>), contains the matrix <i>z</i> if <i>wantz</i> is <code>.TRUE.</code>. If <i>wantz</i> is <code>.FALSE.</code>, <i>z</i> is not referenced.
<i>ldz</i>	INTEGER. The leading dimension of the array <i>z</i>. If <i>wantz</i> is <code>.TRUE.</code>, then $ldz \geq \max(1, ihiz)$. Otherwise, $ldz \geq 1$.
<i>work</i>	REAL for slaqr4 DOUBLE PRECISION for dlaqr4 COMPLEX for claqr4 COMPLEX*16 for zlaqr4. Workspace array with dimension <i>lwork</i>.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i>. $lwork \geq \max(1, n)$ is sufficient, but for the optimal performance a greater workspace may be required, typically as large as $6*n$. It is recommended to use the workspace query to determine the optimal workspace size. If <i>lwork</i>=-1, then the routine performs a workspace query: it estimates the optimal workspace size for the given values of the input parameters <i>n</i>, <i>ilo</i>, and <i>ihi</i>. The estimate is returned in <i>work</i>(1). No error messages related to the <i>lwork</i> is issued by xerbla. Neither <i>H</i> nor <i>z</i> are accessed.

Output Parameters

<i>h</i>	If <i>info</i>=0, and <i>wantt</i> is <code>.TRUE.</code>, then <i>h</i> contains the upper quasi-triangular/triangular matrix <i>T</i> from the Schur decomposition (the Schur form). If <i>info</i>=0, and <i>wantt</i> is <code>.FALSE.</code>, then the contents of <i>h</i> are unspecified on exit.

(The output values of h when $info > 0$ are given under the description of the *info* parameter below.)
The routines may explicitly set $h(i,j)$ for $i > j$ and $j=1,2,\dots,ilo-1$ or $j=ihi+1, ihl+2,\dots,n$.

work(1) On exit *work(1)* contains the minimum value of *lwork* required for optimum performance.

w COMPLEX for *claqr4*
COMPLEX*16 for *zlaqr4*.
Arrays, DIMENSION(*n*). The computed eigenvalues of $h(ilo:ihi, ilo:ihi)$ are stored in $w(ilo:ihi)$. If *wantt* is .TRUE., then the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in h , with $w(i) = h(i,i)$.

wr, wi REAL for *slaqr4*
DOUBLE PRECISION for *dlaqr4*
Arrays, DIMENSION(*ihi*) each. The real and imaginary parts, respectively, of the computed eigenvalues of $h(ilo:ihi, ilo:ihi)$ are stored in the $wr(ilo:ihi)$ and $wi(ilo:ihi)$. If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of *wr* and *wi*, say the *i*-th and (*i*+1)-th, with $wi(i) > 0$ and $wi(i+1) < 0$. If *wantt* is .TRUE., then the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in h , with $wr(i) = h(i,i)$, and if $h(i:i+1, i:i+1)$ is a 2-by-2 diagonal block, then $wi(i) = \sqrt{-h(i+1,i) * h(i,i+1)}$.

z If *wantz* is .TRUE., then $z(ilo:ihi, iloz:ihiz)$ is replaced by $z(ilo:ihi, iloz:ihiz) * U$, where U is the orthogonal/unitary Schur factor of $h(ilo:ihi, ilo:ihi)$. If *wantz* is .FALSE., *z* is not referenced.
(The output values of z when $info > 0$ are given under the description of the *info* parameter below.)

info INTEGER.
= 0: the execution is successful.

> 0: if *info* = *i*, then the routine failed to compute all the eigenvalues. Elements 1:*ilo*-1 and *i*+1:*n* of *wr* and *wi* contain those eigenvalues which have been successfully computed.

> 0: if *wantt* is *.FALSE.*, then the remaining unconverged eigenvalues are the eigenvalues of the upper Hessenberg matrix rows and columns *ilo* through *info* of the final output value of *h*.

> 0: if *wantt* is *.TRUE.*, then (initial value of *h*)**U* = *U**(final value of *h*, where *U* is an orthogonal/unitary matrix. The final value of *h* is upper Hessenberg and quasi-triangular/triangular in rows and columns *info*+1 through *ihi*.

> 0: if *wantz* is *.TRUE.*, then (final value of *z*(*ilo:ihi*, *iloz:ihiz*))=(initial value of *z*(*ilo:ihi*, *iloz:ihiz*))**U*, where *U* is the orthogonal/unitary matrix in the previous expression (regardless of the value of *wantt*).

> 0: if *wantz* is *.FALSE.*, then *z* is not accessed.

?laqr5

Performs a single small-bulge multi-shift QR sweep.

Syntax

```
call slaqr5( wantt, wantz, kacc22, n, ktop, kbot, nshfts, sr, si, h, ldh,
            iloz, ihiz, z, ldz, v, ldv, u, ldu, nv, wv, ldwv, nh, wh, ldwh )

call dlaqr5( wantt, wantz, kacc22, n, ktop, kbot, nshfts, sr, si, h, ldh,
            iloz, ihiz, z, ldz, v, ldv, u, ldu, nv, wv, ldwv, nh, wh, ldwh )

call claqr5( wantt, wantz, kacc22, n, ktop, kbot, nshfts, s, h, ldh, iloz,
            ihiz, z, ldz, v, ldv, u, ldu, nv, wv, ldwv, nh, wh, ldwh )

call zlaqr5( wantt, wantz, kacc22, n, ktop, kbot, nshfts, s, h, ldh, iloz,
            ihiz, z, ldz, v, ldv, u, ldu, nv, wv, ldwv, nh, wh, ldwh )
```

Description

This auxiliary routine called by ?laqr0 performs a single small-bulge multi-shift QR sweep.

Input Parameters

<i>wantt</i>	LOGICAL. <i>wantt</i> = .TRUE. if the quasi-triangular/triangular Schur factor is computed. <i>wantt</i> is set to .FALSE. otherwise.
<i>wantz</i>	LOGICAL. <i>wantz</i> = .TRUE. if the orthogonal/unitary Schur factor is computed. <i>wantz</i> is set to .FALSE. otherwise.
<i>kacc22</i>	INTEGER. Possible values are 0, 1, or 2. Specifies the computation mode of far-from-diagonal orthogonal updates. = 0: the routine does not accumulate reflections and does not use matrix-matrix multiply to update far-from-diagonal matrix entries. = 1: the routine accumulates reflections and uses matrix-matrix multiply to update the far-from-diagonal matrix entries. = 2: the routine accumulates reflections, uses matrix-matrix multiply to update the far-from-diagonal matrix entries, and takes advantage of 2-by-2 block structure during matrix multiplies.
<i>n</i>	INTEGER. The order of the Hessenberg matrix <i>H</i> upon which the routine operates.
<i>ktop, kbot</i>	INTEGER. It is assumed without a check that either <i>ktop</i> =1 or $h(ktop, ktop-1)=0$, and either <i>kbot</i> = <i>n</i> or $h(kbot+1, kbot)=0$.
<i>nshfts</i>	INTEGER. Number of simultaneous shifts, must be positive and even.
<i>sr, si</i>	REAL for slaqr5 DOUBLE PRECISION for dlaqr5 Arrays, DIMENSION (<i>nshfts</i>) each. <i>sr</i> contains the real parts and <i>si</i> contains the imaginary parts of the <i>nshfts</i> shifts of origin that define the multi-shift QR sweep.

<i>s</i>	COMPLEX for <code>claqr5</code> COMPLEX*16 for <code>zlaqr5</code>. Arrays, DIMENSION (<i>nshfts</i>). <i>s</i> contains the shifts of origin that define the multi-shift QR sweep.
<i>h</i>	REAL for <code>slaqr5</code> DOUBLE PRECISION for <code>dlaqr5</code> COMPLEX for <code>claqr5</code> COMPLEX*16 for <code>zlaqr5</code>. Array, DIMENSION (<i>ldh</i>, <i>n</i>), on input contains the Hessenberg matrix.
<i>ldh</i>	INTEGER. The leading dimension of the array <i>h</i> just as declared in the calling routine. $ldh \geq \max(1, n)$.
<i>iloz, ihiz</i>	INTEGER. Specify the rows of <i>z</i> to which transformations must be applied if <i>wantz</i> is <code>.TRUE.</code>. $1 \leq iloz \leq ihiz \leq n$.
<i>z</i>	REAL for <code>slaqr5</code> DOUBLE PRECISION for <code>dlaqr5</code> COMPLEX for <code>claqr5</code> COMPLEX*16 for <code>zlaqr5</code>. Array, DIMENSION (<i>ldz</i>, <i>ihi</i>), contains the matrix <i>z</i> if <i>wantz</i> is <code>.TRUE.</code>. If <i>wantz</i> is <code>.FALSE.</code>, then <i>z</i> is not referenced.
<i>ldz</i>	INTEGER. The leading dimension of the array <i>z</i> just as declared in the calling routine. $ldz \geq n$.
<i>v</i>	REAL for <code>slaqr5</code> DOUBLE PRECISION for <code>dlaqr5</code> COMPLEX for <code>claqr5</code> COMPLEX*16 for <code>zlaqr5</code>. Workspace array with dimension (<i>ldv</i>, <i>nshfts</i>/2).
<i>ldv</i>	INTEGER. The leading dimension of the array <i>v</i> just as declared in the calling routine. $ldv \geq 3$.
<i>u</i>	REAL for <code>slaqr5</code> DOUBLE PRECISION for <code>dlaqr5</code> COMPLEX for <code>claqr5</code>

	COMPLEX*16 for <code>zlaqr5</code> . Workspace array with dimension $(ldu, 3*nshfts-3)$.
<code>ldu</code>	INTEGER. The leading dimension of the array <code>u</code> just as declared in the calling routine. $ldu \geq 3*nshfts-3$.
<code>nh</code>	INTEGER. The number of column in the array <code>wh</code> available for workspace. $nh \geq 1$.
<code>wh</code>	REAL for <code>slaqr5</code> DOUBLE PRECISION for <code>dlaqr5</code> COMPLEX for <code>claqr5</code> COMPLEX*16 for <code>zlaqr5</code> . Workspace array with dimension $(ldwh, nh)$
<code>ldwh</code>	INTEGER. The leading dimension of the array <code>wh</code> just as declared in the calling routine. $ldwh \geq 3*nshfts-3$
<code>nv</code>	INTEGER. The number of rows of the array <code>wv</code> available for workspace. $nv \geq 1$.
<code>wv</code>	REAL for <code>slaqr5</code> DOUBLE PRECISION for <code>dlaqr5</code> COMPLEX for <code>claqr5</code> COMPLEX*16 for <code>zlaqr5</code> . Workspace array with dimension $(ldwv, 3*nshfts-3)$.
<code>ldwv</code>	INTEGER. The leading dimension of the array <code>wv</code> just as declared in the calling routine. $ldwv \geq nv$.

Output Parameters

<code>h</code>	On output a multi-shift QR Sweep with shifts $sr(j)+i*si(j)$ or $s(j)$ is applied to the isolated diagonal block in rows and columns <code>k_{top}</code> through <code>k_{bot}</code> .
<code>z</code>	If <code>wantz</code> is <code>.TRUE.</code> , then the QR Sweep orthogonal/unitary similarity transformation is accumulated into <code>z(ilo:ihiz, ilo:ihi)</code> from the right. If <code>wantz</code> is <code>.FALSE.</code> , then <code>z</code> is unreferenced.

?laqsb

Scales a symmetric band matrix, using scaling factors computed by ?pbequ.

Syntax

```
call slaqsb( uplo, n, kd, ab, ldab, s, scond, amax, equed )
call dlaqsb( uplo, n, kd, ab, ldab, s, scond, amax, equed )
call claqsb( uplo, n, kd, ab, ldab, s, scond, amax, equed )
call zlaqsb( uplo, n, kd, ab, ldab, s, scond, amax, equed )
```

Description

The routine equilibrates a symmetric band matrix *A* using the scaling factors in the vector *s*.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix <i>A</i> is stored. If <i>uplo</i> = 'U': upper triangular. If <i>uplo</i> = 'L': lower triangular.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> . $n \geq 0$.
<i>kd</i>	INTEGER. The number of super-diagonals of the matrix <i>A</i> if <i>uplo</i> = 'U', or the number of sub-diagonals if <i>uplo</i> = 'L'. $kd \geq 0$.
<i>ab</i>	REAL for slaqsb DOUBLE PRECISION for dlaqsb COMPLEX for claqsb COMPLEX*16 for zlaqsb Array, DIMENSION (<i>ldab</i> , <i>n</i>). On entry, the upper or lower triangle of the symmetric band matrix <i>A</i> , stored in the first <i>kd</i> +1 rows of the array. The <i>j</i> -th column of <i>A</i> is stored in the <i>j</i> -th column of the array <i>ab</i> as follows:

	<pre> if uplo = 'U', ab(kd+1+i-j,j) = A(i,j) for max(1,j-kd) ≤ i ≤ j; if uplo = 'L', ab(1+i-j,j) = A(i,j) for j ≤ i ≤ min(n,j+kd). </pre>
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i>. $ldab \geq kd+1$.
<i>s</i>	REAL for slaqsb/claqsb DOUBLE PRECISION for dlaqsb/zlaqsb Array, DIMENSION (<i>n</i>). The scale factors for <i>A</i>.
<i>scond</i>	REAL for slaqsb/claqsb DOUBLE PRECISION for dlaqsb/zlaqsb Ratio of the smallest $s(i)$ to the largest $s(i)$.
<i>amax</i>	REAL for slaqsb/claqsb DOUBLE PRECISION for dlaqsb/zlaqsb Absolute value of largest matrix entry.

Output Parameters

<i>ab</i>	On exit, if <i>info</i> = 0, the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $A = U'^*U$ or $A = L*L'$ of the band matrix <i>A</i>, in the same storage format as <i>A</i>.
<i>equed</i>	CHARACTER*1. Specifies whether or not equilibration was done. If <i>equed</i> = 'N': No equilibration. If <i>equed</i> = 'Y': Equilibration was done, that is, <i>A</i> has been replaced by $\text{diag}(s) * A * \text{diag}(s)$.

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if scaling should be based on the ratio of the scaling factors. If $scond < thresh$, scaling is done. *large* and *small* are threshold values used to decide if scaling should be done based on the absolute size of the largest matrix element. If $amax > large$ or $amax < small$, scaling is done.

?laqsp

Scales a symmetric/Hermitian matrix in packed storage, using scaling factors computed by ?ppequ.

Syntax

```
call slaqsp( uplo, n, ap, s, scond, amax, equed )
call dlaqsp( uplo, n, ap, s, scond, amax, equed )
call claqsp( uplo, n, ap, s, scond, amax, equed )
call zlaqsp( uplo, n, ap, s, scond, amax, equed )
```

Description

The routine ?laqsp equilibrates a symmetric matrix *A* using the scaling factors in the vector *s*.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix <i>A</i> is stored. If <i>uplo</i> = 'U': upper triangular. If <i>uplo</i> = 'L': lower triangular.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> . $n \geq 0$.
<i>ap</i>	REAL for slaqsp DOUBLE PRECISION for dlaqsp COMPLEX for claqsp COMPLEX*16 for zlaqsp Array, DIMENSION $(n(n+1)/2)$. On entry, the upper or lower triangle of the symmetric matrix <i>A</i> , packed columnwise in a linear array. The <i>j</i> -th column of <i>A</i> is stored in the array <i>ap</i> as follows: if <i>uplo</i> = 'U', $ap(i + (j-1)j/2) = A(i, j)$ for $1 \leq i \leq j$; if <i>uplo</i> = 'L', $ap(i + (j-1)(2n-j)/2) = A(i, j)$ for $j \leq i \leq n$.

s REAL for slaqsp/claqsp
 DOUBLE PRECISION for dlaqsp/zlaqsp
 Array, DIMENSION (*n*). The scale factors for *A*.

scond REAL for slaqsp/claqsp
 DOUBLE PRECISION for dlaqsp/zlaqsp
 Ratio of the smallest $s(i)$ to the largest $s(i)$.

amax REAL for slaqsp/claqsp
 DOUBLE PRECISION for dlaqsp/zlaqsp
 Absolute value of largest matrix entry.

Output Parameters

ap On exit, the equilibrated matrix: $\text{diag}(s) * A * \text{diag}(s)$, in the same storage format as *A*.

equed CHARACTER*1.
 Specifies whether or not equilibration was done.
 If *equed* = 'N': No equilibration.
 If *equed* = 'Y': Equilibration was done, that is, *A* has been replaced by $\text{diag}(s) * A * \text{diag}(s)$.

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if scaling should be based on the ratio of the scaling factors. If $scond < thresh$, scaling is done. *large* and *small* are threshold values used to decide if scaling should be done based on the absolute size of the largest matrix element. If $amax > large$ or $amax < small$, scaling is done.

?laqsy

Scales a symmetric/Hermitian matrix, using scaling factors computed by ?poequ.

Syntax

```
call slaqsy( uplo, n, a, lda, s, scond, amax, equed )
call dlaqsy( uplo, n, a, lda, s, scond, amax, equed )
call claqsy( uplo, n, a, lda, s, scond, amax, equed )
call zlaqsy( uplo, n, a, lda, s, scond, amax, equed )
```

Description

The routine equilibrates a symmetric matrix *A* using the scaling factors in the vector *s*.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix <i>A</i> is stored. If <i>uplo</i> = 'U': upper triangular. If <i>uplo</i> = 'L': lower triangular.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> . $n \geq 0$.
<i>a</i>	REAL for slaqsy DOUBLE PRECISION for dlaqsy COMPLEX for claqsy COMPLEX*16 for zlaqsy Array, DIMENSION (<i>lda</i> , <i>n</i>). On entry, the symmetric matrix <i>A</i> . If <i>uplo</i> = 'U', the leading <i>n</i> -by- <i>n</i> upper triangular part of <i>a</i> contains the upper triangular part of the matrix <i>A</i> , and the strictly lower triangular part of <i>a</i> is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i> -by- <i>n</i> lower triangular part of <i>a</i> contains the lower triangular part of the matrix <i>A</i> , and the strictly upper triangular part of <i>a</i> is not referenced.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> .

	$lda \geq \max(n, 1)$.
<i>s</i>	REAL for slaqsy/claqsy DOUBLE PRECISION for dlaqsy/zlaqsy Array, DIMENSION (<i>n</i>). The scale factors for A.
<i>scond</i>	REAL for slaqsy/claqsy DOUBLE PRECISION for dlaqsy/zlaqsy Ratio of the smallest $s(i)$ to the largest $s(i)$.
<i>amax</i>	REAL for slaqsy/claqsy DOUBLE PRECISION for dlaqsy/zlaqsy Absolute value of largest matrix entry.

Output Parameters

<i>a</i>	On exit, if <i>equed</i> = 'Y', the equilibrated matrix: $\text{diag}(s) * A * \text{diag}(s)$.
<i>equed</i>	CHARACTER*1. Specifies whether or not equilibration was done. If <i>equed</i> = 'N': No equilibration. If <i>equed</i> = 'Y': Equilibration was done, i.e., A has been replaced by $\text{diag}(s) * A * \text{diag}(s)$.

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if scaling should be based on the ratio of the scaling factors. If $scond < thresh$, scaling is done. *large* and *small* are threshold values used to decide if scaling should be done based on the absolute size of the largest matrix element. If $amax > large$ or $amax < small$, scaling is done.

?laqtr

*Solves a real quasi-triangular system of equations,
or a complex quasi-triangular system of special
form, in real arithmetic.*

Syntax

```
call slaqtr( ltran, lreal, n, t, ldt, b, w, scale, x, work, info )
call dlaqtr( ltran, lreal, n, t, ldt, b, w, scale, x, work, info )
```

Description

The routine `?laqtr` solves the real quasi-triangular system

$\text{op}(T) * p = \text{scale} * c$, if $\text{lreal} = \text{.TRUE.}$

or the complex quasi-triangular systems

$\text{op}(T + iB) * (p+iq) = \text{scale} * (c+id)$, if $\text{lreal} = \text{.FALSE.}$

in real arithmetic, where T is upper quasi-triangular.

If $\text{lreal} = \text{.FALSE.}$, then the first diagonal block of T must be 1-by-1, B is the specially structured matrix

$$B = \begin{bmatrix} b_1 & b_2 & \dots & \dots & b_n \\ & W & & & \\ & & W & & \\ & & & \dots & \\ & & & & W \end{bmatrix}$$

$\text{op}(A) = A$ or A' , A' denotes the conjugate transpose of matrix A .

On input,

$$x = \begin{bmatrix} c \\ d \end{bmatrix}, \text{ on output } x = \begin{bmatrix} p \\ q \end{bmatrix}$$

This routine is designed for the condition number estimation in routine `?trsna`.

Input Parameters

ltran

LOGICAL.

On entry, *ltran* specifies the option of conjugate transpose:

= `.FALSE.`, $\text{op}(T + iB) = T + iB$,

$= .TRUE., \text{op}(T + iB) = (T + iB)'$.
lreal LOGICAL.
 On entry, *lreal* specifies the input matrix structure:
 $= .FALSE.,$ the input is complex
 $= .TRUE.,$ the input is real.

n INTEGER.
 On entry, *n* specifies the order of $T + iB$. $n \geq 0$.

t REAL for slaqtr
 DOUBLE PRECISION for dlaqtr
 Array, dimension (ldt, n) . On entry, *t* contains a matrix in Schur canonical form. If *lreal* = $.FALSE.,$ then the first diagonal block of *t* must be 1-by-1.

ldt INTEGER. The leading dimension of the matrix *T*.
 $ldt \geq \max(1, n)$.

b REAL for slaqtr
 DOUBLE PRECISION for dlaqtr
 Array, dimension (n) . On entry, *b* contains the elements to form the matrix *B* as described above. If *lreal* = $.TRUE.,$ *b* is not referenced.

w REAL for slaqtr
 DOUBLE PRECISION for dlaqtr
 On entry, *w* is the diagonal element of the matrix *B*.
 If *lreal* = $.TRUE.,$ *w* is not referenced.

x REAL for slaqtr
 DOUBLE PRECISION for dlaqtr
 Array, dimension $(2n)$. On entry, *x* contains the right hand side of the system.

work REAL for slaqtr
 DOUBLE PRECISION for dlaqtr
 Workspace array, dimension (n) .

Output Parameters

scale REAL for slaqtr
 DOUBLE PRECISION for dlaqtr
 On exit, *scale* is the scale factor.

<i>x</i>	On exit, <i>x</i> is overwritten by the solution.
<i>info</i>	INTEGER. If <i>info</i> = 0: successful exit. If <i>info</i> = 1: the some diagonal 1-by-1 block has been perturbed by a small number <i>smin</i> to keep nonsingularity. If <i>info</i> = 2: the some diagonal 2-by-2 block has been perturbed by a small number in ?1aln2 to keep nonsingularity.



NOTE. For higher speed, this routine does not check the inputs for errors.

?lar1v

Computes the (scaled) r -th column of the inverse of the submatrix in rows $b1$ through bn of tridiagonal matrix.

Syntax

```
call slar1v( n, b1, bn, lambda, d, l, ld, lld, pivmin, gaptol, z, wantnc,
negcnt, ztz, mingma, r, isuppz, nrminv, resid, rqcorr, work )
```

```
call dlar1v( n, b1, bn, lambda, d, l, ld, lld, pivmin, gaptol, z, wantnc,
negcnt, ztz, mingma, r, isuppz, nrminv, resid, rqcorr, work )
```

```
call clar1v( n, b1, bn, lambda, d, l, ld, lld, pivmin, gaptol, z, wantnc,
negcnt, ztz, mingma, r, isuppz, nrminv, resid, rqcorr, work )
```

```
call zlar1v( n, b1, bn, lambda, d, l, ld, lld, pivmin, gaptol, z, wantnc,
negcnt, ztz, mingma, r, isuppz, nrminv, resid, rqcorr, work )
```

Description

The routine `?lar1v` computes the (scaled) r -th column of the inverse of the submatrix in rows $b1$ through bn of the tridiagonal matrix $L * D * L^T - \lambda * I$. When λ is close to an eigenvalue, the computed vector is an accurate eigenvector. Usually, r corresponds to the index where the eigenvector is largest in magnitude.

The following steps accomplish this computation :

- Stationary qd transform, $L^*D^*L^T - \lambda^*I = L(+)*D(+)*L(+)^T$
- Progressive qd transform, $L^*D^*L^T - \lambda^*I = U(-)*D(-)*U(-)^T$,
- Computation of the diagonal elements of the inverse of $L^*D^*L^T - \lambda^*I$ by combining the above transforms, and choosing r as the index where the diagonal of the inverse is (one of the) largest in magnitude.
- Computation of the (scaled) r -th column of the inverse using the twisted factorization obtained by combining the top part of the stationary and the bottom part of the progressive transform.

Input Parameters

n	INTEGER. The order of the matrix $L^*D^*L^T$.
$b1$	INTEGER. First index of the submatrix of $L^*D^*L^T$.
bn	INTEGER. Last index of the submatrix of $L^*D^*L^T$.
$lambda$	REAL for slarlvc/clarlvc DOUBLE PRECISION for dlarlv/zlarlv The shift. To compute an accurate eigenvector, $lambda$ should be a good approximation to an eigenvalue of $L^*D^*L^T$.
l	REAL for slarlvc/clarlvc DOUBLE PRECISION for dlarlv/zlarlv Array, DIMENSION $(n-1)$. The $(n-1)$ subdiagonal elements of the unit bidiagonal matrix L , in elements 1 to $n-1$.
d	REAL for slarlvc/clarlvc DOUBLE PRECISION for dlarlv/zlarlv Array, DIMENSION (n) . The n diagonal elements of the diagonal matrix D .
ld	REAL for slarlvc/clarlvc DOUBLE PRECISION for dlarlv/zlarlv Array, DIMENSION $(n-1)$. The $n-1$ elements $L_i^*D_i$.
lld	REAL for slarlvc/clarlvc DOUBLE PRECISION for dlarlv/zlarlv Array, DIMENSION $(n-1)$. The $n-1$ elements $L_i^*L_i^*D_i$.

<i>pivmin</i>	REAL for slarlv/clarlv DOUBLE PRECISION for dlarlv/zlarlv The minimum pivot in the Sturm sequence.
<i>gaptol</i>	REAL for slarlv/clarlv DOUBLE PRECISION for dlarlv/zlarlv Tolerance that indicates when eigenvector entries are negligible with respect to their contribution to the residual.
<i>z</i>	REAL for slarlv DOUBLE PRECISION for dlarlv COMPLEX for clarlv COMPLEX*16 for zlarlv Array, DIMENSION (<i>n</i>). All entries of <i>z</i> must be set to 0.
<i>wantnc</i>	LOGICAL. Specifies whether <i>negcnt</i> has to be computed.
<i>r</i>	INTEGER. The twist index for the twisted factorization used to compute <i>z</i> . On input, $0 \leq r \leq n$. If <i>r</i> is input as 0, <i>r</i> is set to the index where $(L^*D^*L^T - \lambda I)^{-1}$ is largest in magnitude. If $1 \leq r \leq n$, <i>r</i> is unchanged.
<i>work</i>	REAL for slarlv/clarlv DOUBLE PRECISION for dlarlv/zlarlv Workspace array, DIMENSION (4* <i>n</i>).

Output Parameters

<i>z</i>	REAL for slarlv DOUBLE PRECISION for dlarlv COMPLEX for clarlv COMPLEX*16 for zlarlv Array, DIMENSION (<i>n</i>). The (scaled) <i>r</i> -th column of the inverse. <i>z</i> (<i>r</i>) is returned to be 1.
<i>negcnt</i>	INTEGER. If <i>wantnc</i> is .TRUE. then <i>negcnt</i> = the number of pivots < <i>pivmin</i> in the matrix factorization $L^*D^*L^T$, and <i>negcnt</i> = -1 otherwise.
<i>ztz</i>	REAL for slarlv/clarlv DOUBLE PRECISION for dlarlv/zlarlv The square of the 2-norm of <i>z</i> .

<i>mingma</i>	REAL for slarlv/clarlv DOUBLE PRECISION for dlarlv/zlarlv The reciprocal of the largest (in magnitude) diagonal element of the inverse of $L^*D^*L^T - \lambda I$.
<i>r</i>	On output, <i>r</i> is the twist index used to compute <i>z</i> . Ideally, <i>r</i> designates the position of the maximum entry in the eigenvector.
<i>isuppz</i>	INTEGER. Array, DIMENSION (2). The support of the vector in <i>z</i> , that is, the vector <i>z</i> is nonzero only in elements <i>isuppz</i> (1) through <i>isuppz</i> (2).
<i>nrminv</i>	REAL for slarlv/clarlv DOUBLE PRECISION for dlarlv/zlarlv Equals $1/\sqrt{ztz}$.
<i>resid</i>	REAL for slarlv/clarlv DOUBLE PRECISION for dlarlv/zlarlv The residual of the FP vector. $resid = \text{ABS}(mingma)/\sqrt{ztz}$.
<i>rqcorr</i>	REAL for slarlv/clarlv DOUBLE PRECISION for dlarlv/zlarlv The Rayleigh Quotient correction to λ . $rqcorr = mingma/ztz$.

?lar2v

Applies a vector of plane rotations with real cosines and real/complex sines from both sides to a sequence of 2-by-2 symmetric/Hermitian matrices.

Syntax

```
call slar2v( n, x, y, z, incx, c, s, incc )
call dlar2v( n, x, y, z, incx, c, s, incc )
call clar2v( n, x, y, z, incx, c, s, incc )
call zlar2v( n, x, y, z, incx, c, s, incc )
```

Description

The routine `?lar2v` applies a vector of real/complex plane rotations with real cosines from both sides to a sequence of 2-by-2 real symmetric or complex Hermitian matrices, defined by the elements of the vectors `x`, `y` and `z`. For $i = 1, 2, \dots, n$

$$\begin{bmatrix} x_i & z_i \\ \text{conjg}(z_i) & y_i \end{bmatrix} := \begin{bmatrix} c(i) & \text{conjg}(s(i)) \\ -s(i) & c(i) \end{bmatrix} \begin{bmatrix} x_i & z_i \\ \text{conjg}(z_i) & y_i \end{bmatrix} \begin{bmatrix} c(i) & -\text{conjg}(s(i)) \\ s(i) & c(i) \end{bmatrix}$$

Input Parameters

<code>n</code>	INTEGER. The number of plane rotations to be applied.
<code>x, y, z</code>	REAL for <code>slar2v</code> DOUBLE PRECISION for <code>dlar2v</code> COMPLEX for <code>clar2v</code> COMPLEX*16 for <code>zlar2v</code> Arrays, DIMENSION $(1+(n-1)*incx)$ each. Contain the vectors <code>x</code> , <code>y</code> and <code>z</code> , respectively. For all flavors of <code>?lar2v</code> , elements of <code>x</code> and <code>y</code> are assumed to be real.
<code>incx</code>	INTEGER. The increment between elements of <code>x</code> , <code>y</code> , and <code>z</code> . $incx > 0$.
<code>c</code>	REAL for <code>slar2v/clar2v</code> DOUBLE PRECISION for <code>dlar2v/zlar2v</code> Array, DIMENSION $(1+(n-1)*incc)$. The cosines of the plane rotations.
<code>s</code>	REAL for <code>slar2v</code> DOUBLE PRECISION for <code>dlar2v</code> COMPLEX for <code>clar2v</code> COMPLEX*16 for <code>zlar2v</code> Array, DIMENSION $(1+(n-1)*incc)$. The sines of the plane rotations.
<code>incc</code>	INTEGER. The increment between elements of <code>c</code> and <code>s</code> . $incc > 0$.

Output Parameters

x, y, z Vectors x, y and z , containing the results of transform.

?larf

Applies an elementary reflector to a general rectangular matrix.

Syntax

```
call slarf( side, m, n, v, incv, tau, c, ldc, work )
call dlarf( side, m, n, v, incv, tau, c, ldc, work )
call clarf( side, m, n, v, incv, tau, c, ldc, work )
call zlarf( side, m, n, v, incv, tau, c, ldc, work )
```

Description

The routine applies a real/complex elementary reflector H to a real/complex m -by- n matrix C , from either the left or the right. H is represented in the form

$$H = I - \tau v v',$$

where τ is a real/complex scalar and v is a real/complex vector.

If $\tau = 0$, then H is taken to be the unit matrix. For `clarf/zlarf`, to apply H' (the conjugate transpose of H), supply `conjg(tau)` instead of τ .

Input Parameters

$side$	CHARACTER*1. If $side = 'L'$: form $H*C$ If $side = 'R'$: form $C*H$.
m	INTEGER. The number of rows of the matrix C .
n	INTEGER. The number of columns of the matrix C .
v	REAL for <code>slarf</code> DOUBLE PRECISION for <code>dlarf</code> COMPLEX for <code>clarf</code> COMPLEX*16 for <code>zlarf</code> Array, DIMENSION

	$(1 + (m-1) * \text{abs}(\text{incv}))$ if <i>side</i> = 'L' or $(1 + (n-1) * \text{abs}(\text{incv}))$ if <i>side</i> = 'R'. The vector <i>v</i> in the representation of <i>H</i> . <i>v</i> is not used if <i>tau</i> = 0.
<i>incv</i>	INTEGER. The increment between elements of <i>v</i> . <i>incv</i> ≠ 0.
<i>tau</i>	REAL for slarf DOUBLE PRECISION for dlarf COMPLEX for clarf COMPLEX*16 for zlarf The value <i>tau</i> in the representation of <i>H</i> .
<i>c</i>	REAL for slarf DOUBLE PRECISION for dlarf COMPLEX for clarf COMPLEX*16 for zlarf Array, DIMENSION (<i>ldc</i> , <i>n</i>). On entry, the <i>m</i> -by- <i>n</i> matrix <i>C</i> .
<i>ldc</i>	INTEGER. The leading dimension of the array <i>c</i> . $ldc \geq \max(1, m)$.
<i>work</i>	REAL for slarf DOUBLE PRECISION for dlarf COMPLEX for clarf COMPLEX*16 for zlarf Workspace array, DIMENSION (<i>n</i>) if <i>side</i> = 'L' or (<i>m</i>) if <i>side</i> = 'R'.

Output Parameters

<i>c</i>	On exit, <i>c</i> is overwritten by the matrix $H * C$ if <i>side</i> = 'L', or $C * H$ if <i>side</i> = 'R'.

?larfb

Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.

Syntax

```
call slarfb( side, trans, direct, storev, m, n, k, v, ldv, t, ldt, c, ldc,
work, ldwork )

call dlarfb( side, trans, direct, storev, m, n, k, v, ldv, t, ldt, c, ldc,
work, ldwork )

call clarfb( side, trans, direct, storev, m, n, k, v, ldv, t, ldt, c, ldc,
work, ldwork )

call zlarfb( side, trans, direct, storev, m, n, k, v, ldv, t, ldt, c, ldc,
work, ldwork )
```

Description

The routine ?larfb applies a complex block reflector H or its transpose H' to a complex m -by- n matrix C from either left or right.

Input Parameters

<i>side</i>	CHARACTER*1. If <i>side</i> = 'L': apply H or H' from the left If <i>side</i> = 'R': apply H or H' from the right
<i>trans</i>	CHARACTER*1. If <i>trans</i> = 'N': apply H (No transpose) If <i>trans</i> = 'C': apply H' (Conjugate transpose)
<i>direct</i>	CHARACTER*1. Indicates how H is formed from a product of elementary reflectors If <i>direct</i> = 'F': $H = H(1) * H(2) * \dots * H(k)$ (forward) If <i>direct</i> = 'B': $H = H(k) * \dots * H(2) * H(1)$ (backward)
<i>storev</i>	CHARACTER*1. Indicates how the vectors which define the elementary reflectors are stored:

	If <i>storev</i> = 'C': Column-wise If <i>storev</i> = 'R': Row-wise
<i>m</i>	INTEGER. The number of rows of the matrix <i>C</i> .
<i>n</i>	INTEGER. The number of columns of the matrix <i>C</i> .
<i>k</i>	INTEGER. The order of the matrix <i>T</i> (equal to the number of elementary reflectors whose product defines the block reflector).
<i>v</i>	REAL for slarfb DOUBLE PRECISION for dlarfb COMPLEX for clarfb COMPLEX*16 for zlarfb Array, DIMENSION (<i>ldv</i>, <i>k</i>) if <i>storev</i> = 'C' (<i>ldv</i>, <i>m</i>) if <i>storev</i> = 'R' and <i>side</i> = 'L' (<i>ldv</i>, <i>n</i>) if <i>storev</i> = 'R' and <i>side</i> = 'R' The matrix <i>v</i>.
<i>ldv</i>	INTEGER. The leading dimension of the array <i>v</i>. If <i>storev</i> = 'C' and <i>side</i> = 'L', <i>ldv</i> ≥ max(1, <i>m</i>); if <i>storev</i> = 'C' and <i>side</i> = 'R', <i>ldv</i> ≥ max(1, <i>n</i>); if <i>storev</i> = 'R', <i>ldv</i> ≥ <i>k</i>.
<i>t</i>	REAL for slarfb DOUBLE PRECISION for dlarfb COMPLEX for clarfb COMPLEX*16 for zlarfb Array, DIMENSION (<i>ldt</i>, <i>k</i>). Contains the triangular <i>k</i>-by-<i>k</i> matrix <i>T</i> in the representation of the block reflector.
<i>LDT</i>	INTEGER. The leading dimension of the array <i>t</i>. <i>ldt</i> ≥ <i>k</i>.
<i>c</i>	REAL for slarfb DOUBLE PRECISION for dlarfb COMPLEX for clarfb COMPLEX*16 for zlarfb Array, DIMENSION (<i>ldc</i>, <i>n</i>). On entry, the <i>m</i>-by-<i>n</i> matrix <i>C</i>.

ldc INTEGER. The leading dimension of the array *c*.
 $ldc \geq \max(1, m)$.

work REAL for *slarfb*
 DOUBLE PRECISION for *dlarfb*
 COMPLEX for *clarfb*
 COMPLEX*16 for *zlarfb*
 Workspace array, DIMENSION (*ldwork*, *k*).

ldwork INTEGER. The leading dimension of the array *work*.
 If *side* = 'L', $ldwork \geq \max(1, n)$;
 if *side* = 'R', $ldwork \geq \max(1, m)$.

Output Parameters

c On exit, *c* is overwritten by H^*C , or H'^*C , or C^*H , or C^*H' .

?larfg

Generates an elementary reflector (Householder matrix).

Syntax

```
call slarfg( n, alpha, x, incx, tau )
call dlarfg( n, alpha, x, incx, tau )
call clarfg( n, alpha, x, incx, tau )
call zlarfg( n, alpha, x, incx, tau )
```

Description

The routine ?larfg generates a real/complex elementary reflector H of order n , such that

$$H'^* \begin{bmatrix} \alpha \\ x \end{bmatrix} = \begin{bmatrix} \beta \\ 0 \end{bmatrix}, \quad H'^*H = I,$$

where α and β are scalars (with β real for all flavors), and x is an $(n-1)$ -element real/complex vector. H is represented in the form

$$H = I - \tau \begin{bmatrix} 1 \\ v \end{bmatrix} \begin{bmatrix} 1 & v' \end{bmatrix}$$

where τ is a real/complex scalar and v is a real/complex $(n-1)$ -element vector. Note that for `clarfg/zlarfg`, H is not Hermitian.

If the elements of x are all zero (and, for complex flavors, α is real), then $\tau = 0$ and H is taken to be the unit matrix.

Otherwise, $1 \leq \tau \leq 2$ (for real flavors), or

$1 \leq \text{Re}(\tau) \leq 2$ and $\text{abs}(\tau-1) \leq 1$ (for complex flavors).

Input Parameters

n	INTEGER. The order of the elementary reflector.
α	REAL for <code>slarfg</code> DOUBLE PRECISION for <code>dlarfg</code> COMPLEX for <code>clarfg</code> COMPLEX*16 for <code>zlarfg</code> On entry, the value α .
x	REAL for <code>slarfg</code> DOUBLE PRECISION for <code>dlarfg</code> COMPLEX for <code>clarfg</code> COMPLEX*16 for <code>zlarfg</code> Array, DIMENSION $(1+(n-2)*\text{abs}(\text{incx}))$. On entry, the vector x .
incx	INTEGER. The increment between elements of x . $\text{incx} > 0$.

Output Parameters

α	On exit, it is overwritten with the value β .
x	On exit, it is overwritten with the vector v .
τ	REAL for <code>slarfg</code> DOUBLE PRECISION for <code>dlarfg</code> COMPLEX for <code>clarfg</code> COMPLEX*16 for <code>zlarfg</code> The value τ .

?larft

Forms the triangular factor T of a block reflector H
 $= I - V^* T^* V^H$.

Syntax

```
call slarft( direct, storev, n, k, v, ldv, tau, t, ldt )
call dlarft( direct, storev, n, k, v, ldv, tau, t, ldt )
call clarft( direct, storev, n, k, v, ldv, tau, t, ldt )
call zlarft( direct, storev, n, k, v, ldv, tau, t, ldt )
```

Description

The routine ?larft forms the triangular factor T of a real/complex block reflector H of order n , which is defined as a product of k elementary reflectors.

If $direct = 'F'$, $H = H(1) * H(2) * \dots * H(k)$ and T is upper triangular;

If $direct = 'B'$, $H = H(k) * \dots * H(2) * H(1)$ and T is lower triangular.

If $storev = 'C'$, the vector which defines the elementary reflector $H(i)$ is stored in the i -th column of the array v , and $H = I - V^* T^* V^H$.

If $storev = 'R'$, the vector which defines the elementary reflector $H(i)$ is stored in the i -th row of the array v , and $H = I - V^H T^* V$.

Input Parameters

<i>direct</i>	CHARACTER*1. Specifies the order in which the elementary reflectors are multiplied to form the block reflector: = 'F': $H = H(1) * H(2) * \dots * H(k)$ (forward) = 'B': $H = H(k) * \dots * H(2) * H(1)$ (backward)
<i>storev</i>	CHARACTER*1. Specifies how the vectors which define the elementary reflectors are stored (see also <i>Application Notes</i> below): = 'C': column-wise = 'R': row-wise.
<i>n</i>	INTEGER. The order of the block reflector H . $n \geq 0$.

<i>k</i>	INTEGER. The order of the triangular factor <i>T</i> (equal to the number of elementary reflectors). $k \geq 1$.
<i>v</i>	REAL for slarft DOUBLE PRECISION for dlarft COMPLEX for clarft COMPLEX*16 for zlarft Array, DIMENSION (<i>ldv</i> , <i>k</i>) if <i>storev</i> = 'C' or (<i>ldv</i> , <i>n</i>) if <i>storev</i> = 'R'. The matrix <i>v</i> .
<i>ldv</i>	INTEGER. The leading dimension of the array <i>v</i> . If <i>storev</i> = 'C', $ldv \geq \max(1, n)$; if <i>storev</i> = 'R', $ldv \geq k$.
<i>tau</i>	REAL for slarft DOUBLE PRECISION for dlarft COMPLEX for clarft COMPLEX*16 for zlarft Array, DIMENSION (<i>k</i>). <i>tau</i> (<i>i</i>) must contain the scalar factor of the elementary reflector $H(i)$.
<i>ldt</i>	INTEGER. The leading dimension of the output array <i>t</i> . $ldt \geq k$.

Output Parameters

<i>t</i>	REAL for slarft DOUBLE PRECISION for dlarft COMPLEX for clarft COMPLEX*16 for zlarft Array, DIMENSION (<i>ldt</i> , <i>k</i>). The <i>k</i> -by- <i>k</i> triangular factor <i>T</i> of the block reflector. If <i>direct</i> = 'F', <i>T</i> is upper triangular; if <i>direct</i> = 'B', <i>T</i> is lower triangular. The rest of the array is not used.
<i>v</i>	The matrix <i>v</i> .

Application Notes

The shape of the matrix v and the storage of the vectors which define the $H(i)$ is best illustrated by the following example with $n = 5$ and $k = 3$. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

direct = 'F' and *storev* = 'C': *direct* = 'F' and *storev* = 'R':

$$\begin{bmatrix} 1 & & & & \\ v_1 & 1 & & & \\ v_1 & v_2 & 1 & & \\ v_1 & v_2 & v_3 & & \\ v_1 & v_2 & v_3 & & \end{bmatrix}$$

$$\begin{bmatrix} 1 & v_1 & v_1 & v_1 & v_1 \\ & 1 & v_2 & v_2 & v_2 \\ & & 1 & v_3 & v_3 \end{bmatrix}$$

direct = 'B' and *storev* = 'C': *direct* = 'B' and *storev* = 'R':

$$\begin{bmatrix} v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \\ 1 & v_2 & v_3 \\ & 1 & v_3 \\ & & 1 \end{bmatrix}$$

$$\begin{bmatrix} v_1 & v_1 & 1 & & \\ v_2 & v_2 & v_2 & 1 & \\ v_3 & v_3 & v_3 & v_3 & 1 \end{bmatrix}$$

?larfx

Applies an elementary reflector to a general rectangular matrix, with loop unrolling when the reflector has order less than or equal to 10.

Syntax

```
call slarfx( side, m, n, v, tau, c, ldc, work )
call dlarfx( side, m, n, v, tau, c, ldc, work )
call clarfx( side, m, n, v, tau, c, ldc, work )
call zlarfx( side, m, n, v, tau, c, ldc, work )
```

Description

The routine ?larfx applies a real/complex elementary reflector H to a real/complex m -by- n matrix C , from either the left or the right.

H is represented in the form

$H = I - \tau v v'$, where τ is a real/complex scalar and v is a real/complex vector.

If $\tau = 0$, then H is taken to be the unit matrix

Input Parameters

<i>side</i>	CHARACTER*1. If <i>side</i> = 'L': form H^*C If <i>side</i> = 'R': form C^*H .
<i>m</i>	INTEGER. The number of rows of the matrix C .
<i>n</i>	INTEGER. The number of columns of the matrix C .
<i>v</i>	REAL for slarfx DOUBLE PRECISION for dlarfx COMPLEX for clarfx COMPLEX*16 for zlarfx Array, DIMENSION (<i>m</i>) if <i>side</i> = 'L' or (<i>n</i>) if <i>side</i> = 'R'. The vector v in the representation of H .
<i>tau</i>	REAL for slarfx

	DOUBLE PRECISION for dlarfx
	COMPLEX for clarfx
	COMPLEX*16 for zlarfx
	The value <i>tau</i> in the representation of <i>H</i> .
<i>c</i>	REAL for slarfx
	DOUBLE PRECISION for dlarfx
	COMPLEX for clarfx
	COMPLEX*16 for zlarfx
	Array, DIMENSION (<i>ldc</i> , <i>n</i>). On entry, the <i>m</i> -by- <i>n</i> matrix <i>C</i> .
<i>ldc</i>	INTEGER. The leading dimension of the array <i>c</i> . <i>lda</i> ≥ (1, <i>m</i>).
<i>work</i>	REAL for slarfx
	DOUBLE PRECISION for dlarfx
	COMPLEX for clarfx
	COMPLEX*16 for zlarfx
	Workspace array, DIMENSION
	(<i>n</i>) if <i>side</i> = 'L' or
	(<i>m</i>) if <i>side</i> = 'R'.
	<i>work</i> is not referenced if <i>H</i> has order < 11.

Output Parameters

<i>c</i>	On exit, <i>C</i> is overwritten by the matrix H^*C if <i>side</i> = 'L', or C^*H if <i>side</i> = 'R'.

?largv

Generates a vector of plane rotations with real cosines and real/complex sines.

Syntax

```
call slargv( n, x, incx, y, incy, c, incc )
call dlargv( n, x, incx, y, incy, c, incc )
call clargv( n, x, incx, y, incy, c, incc )
call zlargv( n, x, incx, y, incy, c, incc )
```

Description

The routine generates a vector of real/complex plane rotations with real cosines, determined by elements of the real/complex vectors x and y .

For slargv/dlargv:

$$\begin{bmatrix} c(i) & s(i) \\ -s(i) & c(i) \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} a_i \\ 0 \end{bmatrix}, \text{ for } i = 1, 2, \dots, n$$

For clargv/zlargv:

$$\begin{bmatrix} c(i) & s(i) \\ -\text{conjg}(s(i)) & c(i) \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} r_i \\ 0 \end{bmatrix}, \text{ for } i = 1, 2, \dots, n$$

where $c(i)^2 + \text{abs}(s(i))^2 = 1$ and the following conventions are used (these are the same as in clartg/zlartg but differ from the BLAS Level 1 routine crotg/zrotg):

If $y_i = 0$, then $c(i) = 1$ and $s(i) = 0$;

If $x_i = 0$, then $c(i) = 0$ and $s(i)$ is chosen so that r_i is real.

Input Parameters

n	INTEGER. The number of plane rotations to be generated.
x, y	REAL for slargv DOUBLE PRECISION for dlargv COMPLEX for clargv COMPLEX*16 for zlargv Arrays, DIMENSION $(1+(n-1)*incx)$ and $(1+(n-1)*incy)$, respectively. On entry, the vectors x and y .
$incx$	INTEGER. The increment between elements of x . $incx > 0$.
$incy$	INTEGER. The increment between elements of y . $incy > 0$.

incc INTEGER. The increment between elements of the output array *c*. *incc* > 0.

Output Parameters

x On exit, $x(i)$ is overwritten by a_i (for real flavors), or by r_i (for complex flavors), for $i = 1, \dots, n$.

y On exit, the sines $s(i)$ of the plane rotations.

c REAL for slargv/clargv
DOUBLE PRECISION for dlargv/zlargv
Array, DIMENSION $(1+(n-1)*incc)$. The cosines of the plane rotations.

?larnv

Returns a vector of random numbers from a uniform or normal distribution.

Syntax

```
call slarnv( idist, iseed, n, x )
call dlarnv( idist, iseed, n, x )
call clarnv( idist, iseed, n, x )
call zlarnv( idist, iseed, n, x )
```

Description

The routine ?larnv returns a vector of n random real/complex numbers from a uniform or normal distribution.

This routine calls the auxiliary routine ?laruv to generate random real numbers from a uniform (0,1) distribution, in batches of up to 128 using vectorisable code. The Box-Muller method is used to transform numbers from a uniform to a normal distribution.

Input Parameters

idist INTEGER. Specifies the distribution of the random numbers:
for slarnv and dlarnv:
= 1: uniform (0,1)
= 2: uniform (-1,1)

= 3: normal (0,1).
for `clarndv` and `zlarndv`:
= 1: real and imaginary parts each uniform (0,1)
= 2: real and imaginary parts each uniform (-1,1)
= 3: real and imaginary parts each normal (0,1)
= 4: uniformly distributed on the disc $\text{abs}(z) < 1$
= 5: uniformly distributed on the circle $\text{abs}(z) = 1$

iseed INTEGER. Array, DIMENSION (4).
On entry, the seed of the random number generator; the array elements must be between 0 and 4095, and *iseed*(4) must be odd.

n INTEGER. The number of random numbers to be generated.

Output Parameters

x REAL for `slarndv`
DOUBLE PRECISION for `dlarndv`
COMPLEX for `clarndv`
COMPLEX*16 for `zlarndv`
Array, DIMENSION (*n*). The generated random numbers.

iseed On exit, the seed is updated.

?larra

Computes the splitting points with the specified threshold.

Syntax

```
call slarra( n, d, e, e2, spltol, tnrm, nsplit, isplit, info )
call dlarra( n, d, e, e2, spltol, tnrm, nsplit, isplit, info )
```

Description

This routine computes the splitting points with the specified threshold and sets any "small" off-diagonal elements to zero.

Input Parameters

n INTEGER. The order of the matrix ($n > 1$).

<i>d</i>	REAL for slarra DOUBLE PRECISION for dlarra Array, DIMENSION (<i>n</i>). Contains <i>n</i> diagonal elements of the tridiagonal matrix <i>T</i> .
<i>e</i>	REAL for slarra DOUBLE PRECISION for dlarra Array, DIMENSION (<i>n</i>). First (<i>n</i> -1) entries contain the subdiagonal elements of the tridiagonal matrix <i>T</i> ; <i>e</i> (<i>n</i>) need not be set.
<i>e2</i>	REAL for slarra DOUBLE PRECISION for dlarra Array, DIMENSION (<i>n</i>). First (<i>n</i> -1) entries contain the squares of the subdiagonal elements of the tridiagonal matrix <i>T</i> ; <i>e2</i> (<i>n</i>) need not be set.
<i>spltol</i>	REAL for slarra DOUBLE PRECISION for dlarra The threshold for splitting. Two criteria can be used: <i>spltol</i> <0 : criterion based on absolute off-diagonal value; <i>spltol</i> >0 : criterion that preserves relative accuracy.
<i>tnrm</i>	REAL for slarra DOUBLE PRECISION for dlarra The norm of the matrix.

Output Parameters

<i>e</i>	On exit, the entries <i>e</i> (<i>isplit</i> (<i>i</i>)), $1 \leq i \leq nsplit$, are set to zero, the other entries of <i>e</i> are untouched.
<i>e2</i>	On exit, the entries <i>e2</i> (<i>isplit</i> (<i>i</i>)), $1 \leq i \leq nsplit$, are set to zero.
<i>nsplit</i>	INTEGER. The number of blocks the matrix <i>T</i> splits into. $1 \leq nsplit \leq n$
<i>isplit</i>	INTEGER. Array, DIMENSION (<i>n</i>).

The splitting points, at which T breaks up into blocks. The first block consists of rows/columns 1 to $isplit(1)$, the second of rows/columns $isplit(1)+1$ through $isplit(2)$, and so on, and the $nsplit$ -th consists of rows/columns $isplit(nsplit-1)+1$ through $isplit(nsplit)=n$.

info

INTEGER.

= 0: successful exit.

?larrb

Provides limited bisection to locate eigenvalues for more accuracy.

Syntax

call slarrb(*n*, *d*, *lld*, *ifirst*, *ilast*, *rtol1*, *rtol2*, *offset*, *w*, *wgap*, *werr*,
work, *iwork*, *pivmin*, *spdiam*, *twist*, *info*)

call dlarrb(*n*, *d*, *lld*, *ifirst*, *ilast*, *rtol1*, *rtol2*, *offset*, *w*, *wgap*, *werr*,
work, *iwork*, *pivmin*, *spdiam*, *twist*, *info*)

Description

Given the relatively robust representation (RRR) $L^*D^*L^T$, the routine does "limited" bisection to refine the eigenvalues of $L^*D^*L^T$, $w(ifirst-offset)$ through $w(ilast-offset)$, to more accuracy. Initial guesses for these eigenvalues are input in w . The corresponding estimate of the error in these guesses and their gaps are input in $werr$ and $wgap$, respectively. During bisection, intervals $[left, right]$ are maintained by storing their mid-points and semi-widths in the arrays w and $werr$ respectively.

Input Parameters

n INTEGER. The order of the matrix.

d REAL for slarrb
DOUBLE PRECISION for dlarrb
Array, DIMENSION (*n*). The *n* diagonal elements of the diagonal matrix D .

lld REAL for slarrb
DOUBLE PRECISION for dlarrb
Array, DIMENSION (*n-1*).

	The $n-1$ elements $L_i * L_i * D_i$.
<i>ifirst</i>	INTEGER. The index of the first eigenvalue to be computed.
<i>ilast</i>	INTEGER. The index of the last eigenvalue to be computed.
<i>rtol1, rtol2</i>	REAL for slarrb DOUBLE PRECISION for dlarrb Tolerance for the convergence of the bisection intervals. An interval [<i>left</i> , <i>right</i>] has converged if $\text{RIGHT-LEFT.LT.MAX}(\text{rtol1} * \text{gap}, \text{rtol2} * \max(\text{left} , \text{right}))$, where <i>gap</i> is the (estimated) distance to the nearest eigenvalue.
<i>offset</i>	INTEGER. Offset for the arrays <i>w</i> , <i>wgap</i> and <i>werr</i> , that is, the <i>ifirst</i> - <i>offset</i> through <i>ilast</i> - <i>offset</i> elements of these arrays are to be used.
<i>w</i>	REAL for slarrb DOUBLE PRECISION for dlarrb Array, DIMENSION (<i>n</i>). On input, <i>w</i> (<i>ifirst</i> - <i>offset</i>) through <i>w</i> (<i>ilast</i> - <i>offset</i>) are estimates of the eigenvalues of $L * D * L^T$ indexed <i>ifirst</i> through <i>ilast</i> .
<i>wgap</i>	REAL for slarrb DOUBLE PRECISION for dlarrb Array, DIMENSION (<i>n</i> -1). The estimated gaps between consecutive eigenvalues of $L * D * L^T$, that is, <i>wgap</i> (<i>i</i> - <i>offset</i>) is the gap between eigenvalues <i>i</i> and <i>i</i> +1. Note that if <i>IFIRST</i> .EQ. <i>ILAST</i> then <i>wgap</i> (<i>ifirst</i> - <i>offset</i>) must be set to 0.
<i>werr</i>	REAL for slarrb DOUBLE PRECISION for dlarrb Array, DIMENSION (<i>n</i>). On input, <i>werr</i> (<i>ifirst</i> - <i>offset</i>) through <i>werr</i> (<i>ilast</i> - <i>offset</i>) are the errors in the estimates of the corresponding elements in <i>w</i> .
<i>work</i>	REAL for slarrb DOUBLE PRECISION for dlarrb Workspace array, DIMENSION (2* <i>n</i>).
<i>pivmin</i>	REAL for slarrb DOUBLE PRECISION for dlarrb The minimum pivot in the Sturm sequence.

<i>spdiam</i>	REAL for slarrb DOUBLE PRECISION for dlarrb The spectral diameter of the matrix.
<i>twist</i>	INTEGER. The twist index for the twisted factorization that is used for the negcount. $twist = n: \text{ Compute negcount from } L^*D^*L^T - \lambda i$ $= L_+^* D_+^* L_+^T$ $twist = n: \text{ Compute negcount from } L^*D^*L^T - \lambda i$ $= U_-^* D_-^* U_-^T$ $twist = n: \text{ Compute negcount from } L^*D^*L^T - \lambda i$ $= N_r^* D_r^* N_r$
<i>iwork</i>	INTEGER. Workspace array, DIMENSION (2*n).

Output Parameters

<i>w</i>	On output, the estimates of the eigenvalues are “refined”.
<i>wgap</i>	On output, the gaps are refined.
<i>werr</i>	On output, “refined” errors in the estimates of <i>w</i> .
<i>info</i>	INTEGER. Error flag.

slarrc

Computes the number of eigenvalues of the symmetric tridiagonal matrix.

Syntax

```
call slarrc( jobt, n, vl, vu, d, e, pivmin, eigcnt, lcnt, rcnt, info )
call dlarrc( jobt, n, vl, vu, d, e, pivmin, eigcnt, lcnt, rcnt, info )
```

Description

This routine finds the number of eigenvalues of the symmetric tridiagonal matrix *T* or of its factorization $L^*D^*L^T$ in the specified interval.

Input Parameters

<i>jobt</i>	CHARACTER*1. = 'T': computes Sturm count for matrix T . = 'L': computes Sturm count for matrix $L^*D^*L^T$.
<i>n</i>	INTEGER. The order of the matrix. ($n > 1$).
<i>vl,vu</i>	REAL for slarrc DOUBLE PRECISION for dlarrc The lower and upper bounds for the eigenvalues.
<i>d</i>	REAL for slarrc DOUBLE PRECISION for dlarrc Array, DIMENSION (n). If <i>jobt</i> = 'T': contains the n diagonal elements of the tridiagonal matrix T . If <i>jobt</i> = 'L': contains the n diagonal elements of the diagonal matrix D .
<i>e</i>	REAL for slarrc DOUBLE PRECISION for dlarrc Array, DIMENSION (n). If <i>jobt</i> = 'T': contains the $(n-1)$ offdiagonal elements of the matrix T . If <i>jobt</i> = 'L': contains the $(n-1)$ offdiagonal elements of the matrix L .
<i>pivmin</i>	REAL for slarrc DOUBLE PRECISION for dlarrc The minimum pivot in the Sturm sequence for the matrix T .

Output Parameters

<i>eigcnt</i>	INTEGER. The number of eigenvalues of the symmetric tridiagonal matrix T that are in the half-open interval $(vl, vu]$.
<i>lcnt,rcnt</i>	INTEGER. The left and right negcounts of the interval.
<i>info</i>	INTEGER.

Now it is not used and always is set to 0.

?larrd

Computes the eigenvalues of a symmetric tridiagonal matrix to suitable accuracy.

Syntax

```
call slarrd( range, order, n, vl, vu, il, iu, gers, reltol, d, e, e2, pivmin,
            nsplit, isplit, m, w, werr, wl, wu, iblock, indexw, work, iwork, info )

call dlarrd( range, order, n, vl, vu, il, iu, gers, reltol, d, e, e2, pivmin,
            nsplit, isplit, m, w, werr, wl, wu, iblock, indexw, work, iwork, info )
```

Description

The routine computes the eigenvalues of a symmetric tridiagonal matrix T to suitable accuracy. This is an auxiliary code to be called from `?stemr`. The user may ask for all eigenvalues, all eigenvalues in the half-open interval $(vl, vu]$, or the il -th through iu -th eigenvalues.

To avoid overflow, the matrix must be scaled so that its largest element is no greater than $(\text{overflow}^{1/2} * \text{underflow}^{1/4})$ in absolute value, and for greatest accuracy, it should not be much smaller than that. (For more details see [Kahan66]).

Input Parameters

<i>range</i>	CHARACTER. = 'A': ("All") all eigenvalues will be found. = 'V': ("Value") all eigenvalues in the half-open interval $(vl, vu]$ will be found. = 'I': ("Index") the il-th through iu-th eigenvalues will be found.
<i>order</i>	CHARACTER. = 'B': ("By block") the eigenvalues will be grouped by split-off block (see <i>iblock</i>, <i>isplit</i> below) and ordered from smallest to largest within the block. = 'E': ("Entire matrix") the eigenvalues for the entire matrix will be ordered from smallest to largest.
<i>n</i>	INTEGER. The order of the tridiagonal matrix T ($n \geq 1$).
<i>vl,vu</i>	REAL for <code>slarrd</code>

DOUBLE PRECISION for dlarrrd
 If *range* = 'V': the lower and upper bounds of the interval to be searched for eigenvalues. Eigenvalues less than or equal to *vl*, or greater than *vu*, will not be returned.
vl < *vu*.
 If *range* = 'A' or 'I': not referenced.

il,iu INTEGER.
 If *range* = 'I': the indices (in ascending order) of the smallest and largest eigenvalues to be returned. $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n=0$.
 If *range* = 'A' or 'V': not referenced.

gers REAL for slarrrd
 DOUBLE PRECISION for dlarrrd
 Array, DIMENSION (2*n).
 The n Gerschgorin intervals (the i -th Gerschgorin interval is (*gers*(2*i-1), *gers*(2*i))).

reltol REAL for slarrrd
 DOUBLE PRECISION for dlarrrd
 The minimum relative width of an interval. When an interval is narrower than *reltol* times the larger (in magnitude) endpoint, then it is considered to be sufficiently small, that is converged. Note: this should always be at least *radix*machine epsilon*.

d REAL for slarrrd
 DOUBLE PRECISION for dlarrrd
 Array, DIMENSION (n).
 Contains n diagonal elements of the tridiagonal matrix T .

e REAL for slarrrd
 DOUBLE PRECISION for dlarrrd
 Array, DIMENSION ($n-1$).
 Contains ($n-1$) off-diagonal elements of the tridiagonal matrix T .

e2 REAL for slarrrd
 DOUBLE PRECISION for dlarrrd
 Array, DIMENSION ($n-1$).

	Contains $(n-1)$ squared off-diagonal elements of the tridiagonal matrix T .
<i>pivmin</i>	REAL for slarrd DOUBLE PRECISION for dlarrrd The minimum pivot in the Sturm sequence for the matrix T .
<i>nsplit</i>	INTEGER. The number of diagonal blocks the matrix T . $1 \leq nsplit \leq n$
<i>isplit</i>	INTEGER. Arrays, DIMENSION (n) . The splitting points, at which T breaks up into submatrices. The first submatrix consists of rows/columns 1 to <i>isplit</i> (1), the second of rows/columns <i>isplit</i> (1)+1 through <i>isplit</i> (2), and so on, and the <i>nsplit</i> -th consists of rows/columns <i>isplit</i> (<i>nsplit</i> -1)+1 through <i>isplit</i> (<i>nsplit</i>)= n . (Only the first <i>nsplit</i> elements actually is used, but since the user cannot know a priori value of <i>nsplit</i> , n words must be reserved for <i>isplit</i> .)
<i>work</i>	REAL for slarrd DOUBLE PRECISION for dlarrrd Workspace array, DIMENSION $(4*n)$.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION $(4*n)$.

Output Parameters

<i>m</i>	INTEGER. The actual number of eigenvalues found. $0 \leq m \leq n$. (See also the description of <i>info</i> =2, 3.)
<i>w</i>	REAL for slarrd DOUBLE PRECISION for dlarrrd Array, DIMENSION (n) .

The first m elements of w contain the eigenvalue approximations. ?laprd computes an interval $I_j = (a_j, b_j]$ that includes eigenvalue j . The eigenvalue approximation is given as the interval midpoint $w(j) = (a_j + b_j) / 2$. The corresponding error is bounded by $werr(j) = \text{abs}(a_j - b_j) / 2$.

werr

REAL for slarrd
DOUBLE PRECISION for dlarrrd
Array, DIMENSION (n).

The error bound on the corresponding eigenvalue approximation in w .

wl, wu

REAL for slarrd
DOUBLE PRECISION for dlarrrd
The interval $(wl, wu]$ contains all the wanted eigenvalues.
If *range* = 'V': then $wl = vl$ and $wu = vu$.
If *range* = 'A': then wl and wu are the global Gerschgorin bounds on the spectrum.
If *range* = 'I': then wl and wu are computed by ?laebz from the index range specified.

iblock

INTEGER.
Array, DIMENSION (n).
At each row/column j where $e(j)$ is zero or small, the matrix T is considered to split into a block diagonal matrix.
If $info = 0$, then $iblock(i)$ specifies to which block (from 1 to the number of blocks) the eigenvalue $w(i)$ belongs.
(The routine may use the remaining $n-m$ elements as workspace.)

indexw

INTEGER.
Array, DIMENSION (n).
The indices of the eigenvalues within each block (submatrix); for example, $indexw(i) = j$ and $iblock(i) = k$ imply that the i -th eigenvalue $w(i)$ is the j -th eigenvalue in block k .

info

INTEGER.
= 0: successful exit.
< 0: if $info = -i$, the i -th argument has an illegal value
> 0: some or all of the eigenvalues fail to converge or are not computed:

=1 or 3: bisection fail to converge for some eigenvalues; these eigenvalues are flagged by a negative block number. The effect is that the eigenvalues may not be as accurate as the absolute and relative tolerances.

=2 or 3: `range='I'` only: not all of the eigenvalues `il:iu` are found.

=4: `range='I'`, and the Gershgorin interval initially used is too small. No eigenvalues are computed.

?larre

Given the tridiagonal matrix T , sets small off-diagonal elements to zero and for each unreduced block T_i , finds base representations and eigenvalues.

Syntax

```
call slarre( range, n, vl, vu, il, iu, d, e, e2, rtol1, rtol2, spltol, nsplit,
            isplit, m, w, werr, wgap, iblock, indexw, gers, pivmin, work, iwork, info )

call dlarre( range, n, vl, vu, il, iu, d, e, e2, rtol1, rtol2, spltol, nsplit,
            isplit, m, w, werr, wgap, iblock, indexw, gers, pivmin, work, iwork, info )
```

Description

To find the desired eigenvalues of a given real symmetric tridiagonal matrix T , the routine sets any “small” off-diagonal elements to zero, and for each unreduced block T_i , it finds

- a suitable shift at one end of the block spectrum
- the base representation, $T_i - \sigma_i * I = L_i * D_i * L_i^T$, and
- eigenvalues of each $L_i * D_i * L_i^T$.

The representations and eigenvalues found are then used by `?stemr` to compute the eigenvectors of a symmetric tridiagonal matrix. The accuracy varies depending on whether bisection is used to find a few eigenvalues or the `dqds` algorithm (subroutine `?lasq2`) to compute all and discard any unwanted one. As an added benefit, `?larre` also outputs the n Gerschgorin intervals for the matrices $L_i * D_i * L_i^T$.

Input Parameters

<i>range</i>	CHARACTER. = 'A': ("All") all eigenvalues will be found. = 'V': ("Value") all eigenvalues in the half-open interval $(v_l, v_u]$ will be found. = 'I': ("Index") the i_l-th through i_u-th eigenvalues of the entire matrix will be found.
<i>n</i>	INTEGER. The order of the matrix. $n > 0$.
<i>v_l, v_u</i>	REAL for slarre DOUBLE PRECISION for dlarre If <i>range</i>='V', the lower and upper bounds for the eigenvalues. Eigenvalues less than or equal to v_l, or greater than v_u, are not returned. $v_l < v_u$.
<i>i_l, i_u</i>	INTEGER. If <i>range</i>='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. $1 \leq i_l \leq i_u \leq n$.
<i>d</i>	REAL for slarre DOUBLE PRECISION for dlarre Array, DIMENSION (n). The n diagonal elements of the diagonal matrices T.
<i>e</i>	REAL for slarre DOUBLE PRECISION for dlarre Array, DIMENSION (n). The first $(n-1)$ entries contain the subdiagonal elements of the tridiagonal matrix T; $e(n)$ need not be set.
<i>e2</i>	REAL for slarre DOUBLE PRECISION for dlarre Array, DIMENSION (n). The first $(n-1)$ entries contain the squares of the subdiagonal elements of the tridiagonal matrix T; $e2(n)$ need not be set.
<i>rtol1, rtol2</i>	REAL for slarre DOUBLE PRECISION for dlarre

	Parameters for bisection. An interval <code>[LEFT,RIGHT]</code> has converged if <code>RIGHT-LEFT.LT.MAX(rtol1*gap, rtol2*max(LEFT , RIGHT))</code> .
<code>spltol</code>	REAL for <code>slarre</code> DOUBLE PRECISION for <code>dlarre</code> The threshold for splitting.
<code>work</code>	REAL for <code>slarre</code> DOUBLE PRECISION for <code>dlarre</code> Workspace array, DIMENSION <code>(6*n)</code> .
<code>iwork</code>	INTEGER. Workspace array, DIMENSION <code>(5*n)</code> .

Output Parameters

<code>vl, vu</code>	On exit, if <code>range='I'</code> or <code>'A'</code> , contain the bounds on the desired part of the spectrum.
<code>d</code>	On exit, the n diagonal elements of the diagonal matrices D_i .
<code>e</code>	On exit, the subdiagonal elements of the unit bidiagonal matrices L_i . The entries <code>e(isplit(i))</code> , $1 \leq i \leq nsplit$, contain the base points σ_{σ_i} on output.
<code>e2</code>	On exit, the entries <code>e2(isplit(i))</code> , $1 \leq i \leq nsplit$, have been set to zero.
<code>nsplit</code>	INTEGER. The number of blocks T splits into. $1 \leq nsplit \leq n$.
<code>isplit</code>	INTEGER. Array, DIMENSION <code>(n)</code> . The splitting points, at which T breaks up into blocks. The first block consists of rows/columns 1 to <code>isplit(1)</code> , the second of rows/columns <code>isplit(1)+1</code> through <code>isplit(2)</code> , etc., and the <code>nsplit</code> -th consists of rows/columns <code>isplit(nsplit-1)+1</code> through <code>isplit(nsplit)=n</code> .
<code>m</code>	INTEGER. The total number of eigenvalues (of all the $L_i * D_i * L_i^T$) found.
<code>w</code>	REAL for <code>slarre</code> DOUBLE PRECISION for <code>dlarre</code>

Array, DIMENSION (n). The first m elements contain the eigenvalues. The eigenvalues of each of the blocks, $L_i * D_i * L_i^T$, are sorted in ascending order. The routine may use the remaining $n-m$ elements as workspace.

werr REAL for slarre
DOUBLE PRECISION for dlarre
Array, DIMENSION (n). The error bound on the corresponding eigenvalue in w .

wgap REAL for slarre
DOUBLE PRECISION for dlarre
Array, DIMENSION (n). The separation from the right neighbor eigenvalue in w . The gap is only with respect to the eigenvalues of the same block as each block has its own representation tree. Exception: at the right end of a block the left gap is stored.

iblock INTEGER. Array, DIMENSION (n).
The indices of the blocks (submatrices) associated with the corresponding eigenvalues in w ; $iblock(i)=1$ if eigenvalue $w(i)$ belongs to the first block from the top, $=2$ if $w(i)$ belongs to the second block, etc.

indexw INTEGER. Array, DIMENSION (n).
The indices of the eigenvalues within each block (submatrix); for example, $indexw(i)=10$ and $iblock(i)=2$ imply that the i -th eigenvalue $w(i)$ is the 10-th eigenvalue in the second block.

gers REAL for slarre
DOUBLE PRECISION for dlarre
Array, DIMENSION ($2*n$). The n Gerschgorin intervals (the i -th Gerschgorin interval is $(gers(2*i-1), gers(2*i))$).

pivmin REAL for slarre
DOUBLE PRECISION for dlarre
The minimum pivot in the Sturm sequence for T .

info INTEGER.
If $info = 0$: successful exit
If $info > 0$: A problem occurred in ?larre. If $info = 5$, the Rayleigh Quotient Iteration failed to converge to full accuracy.

If $info < 0$: One of the called subroutines signaled an internal problem. Inspection of the corresponding parameter $info$ for further information is required.

- If $info = -1$, there is a problem in `?larrrd`
- If $info = -2$, no base representation could be found in `maxtry` iterations. Increasing `maxtry` and recompilation might be a remedy.
- If $info = -3$, there is a problem in `?larrrb` when computing the refined root representation for `?lasq2`.
- If $info = -4$, there is a problem in `?larrrb` when performing bisection on the desired part of the spectrum.
- If $info = -5$, there is a problem in `?lasq2`.
- If $info = -6$, there is a problem in `?lasq2`.

?larrrf

Finds a new relatively robust representation such that at least one of the eigenvalues is relatively isolated.

Syntax

```
call slarrrf( n, d, l, ld, clstrt, clend, w, wgap, werr, spdiam, clgapl, clgapr,
             pivmin, sigma, dplus, lplus, work, info )
```

```
call dlarrrf( n, d, l, ld, clstrt, clend, w, wgap, werr, spdiam, clgapl, clgapr,
             pivmin, sigma, dplus, lplus, work, info )
```

Description

Given the initial representation $L * D * L^T$ and its cluster of close eigenvalues (in a relative measure), $w(clstrt), w(clstrt+1), \dots w(clend)$, the routine `?larrrf` finds a new relatively robust representation

$$L * D * L^T - \sigma_i * I = L(+) * D(+) * L(+)^T$$

such that at least one of the eigenvalues of $L(+) * D(+) * L(+)^T$ is relatively isolated.

Input Parameters

<i>n</i>	INTEGER. The order of the matrix (subblock, if the matrix is splitted).
<i>d</i>	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION (<i>n</i>). The <i>n</i> diagonal elements of the diagonal matrix <i>D</i> .
<i>l</i>	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION (<i>n</i> -1). The (<i>n</i> -1) subdiagonal elements of the unit bidiagonal matrix <i>L</i> .
<i>ld</i>	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION (<i>n</i> -1). The <i>n</i> -1 elements $L_i * D_i$.
<i>clstrt</i>	INTEGER. The index of the first eigenvalue in the cluster.
<i>clend</i>	INTEGER. The index of the last eigenvalue in the cluster.
<i>w</i>	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION $\geq (clend - clstrt + 1)$. The eigenvalue approximations of $L * D * L^T$ in ascending order. $w(clstrt)$ through $w(clend)$ form the cluster of relatively close eigenvalues.
<i>wgap</i>	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION $\geq (clend - clstrt + 1)$. The separation from the right neighbor eigenvalue in <i>w</i> .
<i>werr</i>	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION $\geq (clend - clstrt + 1)$. On input, <i>werr</i> contains the semiwidth of the uncertainty interval of the corresponding eigenvalue approximation in <i>w</i> .
<i>spdiam</i>	REAL for slarrf DOUBLE PRECISION for dlarrf

	Estimate of the spectral diameter obtained from the Gerschgorin intervals.
<i>clgapl, clgapr</i>	REAL for slarrf DOUBLE PRECISION for dlarrf Absolute gap on each end of the cluster. Set by the calling routine to protect against shifts too close to eigenvalues outside the cluster.
<i>pivmin</i>	REAL for slarrf DOUBLE PRECISION for dlarrf The minimum pivot allowed in the Sturm sequence.
<i>work</i>	REAL for slarrf DOUBLE PRECISION for dlarrf Workspace array, DIMENSION (2*n).

Output Parameters

<i>wgap</i>	On output, the gaps are refined.
<i>sigma</i>	REAL for slarrf DOUBLE PRECISION for dlarrf The shift used to form $L(+) * D^*(+) * L(+)^T$.
<i>dplus</i>	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION (n). The n diagonal elements of the diagonal matrix $D(+)$.
<i>lplus</i>	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION (n). The first (n-1) elements of <i>lplus</i> contain the subdiagonal elements of the unit bidiagonal matrix $L(+)$.

zlarrj

Performs refinement of the initial estimates of the eigenvalues of the matrix T .

Syntax

```
call slarrj( n, d, e2, ifirst, ilast, rtol, offset, w, werr, work, iwork,
            pivmin, spdiam, info )
```

```
call dlarrj( n, d, e2, ifirst, ilast, rtol, offset, w, werr, work, iwork,
            pivmin, spdiam, info )
```

Description

Given the initial eigenvalue approximations of T , this routine does bisection to refine the eigenvalues of T , $w(\text{ifirst}-\text{offset})$ through $w(\text{ilast}-\text{offset})$, to more accuracy. Initial guesses for these eigenvalues are input in w , the corresponding estimate of the error in these guesses in $werr$. During bisection, intervals $[a,b]$ are maintained by storing their mid-points and semi-widths in the arrays w and $werr$ respectively.

Input Parameters

<i>n</i>	INTEGER. The order of the matrix T .
<i>d</i>	REAL for slarrj DOUBLE PRECISION for dlarrj Array, DIMENSION (n). Contains n diagonal elements of the matrix T .
<i>e2</i>	REAL for slarrj DOUBLE PRECISION for dlarrj Array, DIMENSION ($n-1$). Contains ($n-1$) squared sub-diagonal elements of the T .
<i>ifirst</i>	INTEGER. The index of the first eigenvalue to be computed.
<i>ilast</i>	INTEGER. The index of the last eigenvalue to be computed.
<i>rtol</i>	REAL for slarrj DOUBLE PRECISION for dlarrj

	Tolerance for the convergence of the bisection intervals. An interval $[a, b]$ is considered to be converged if $(b-a) \leq rtol * \max(a , b)$.
<i>offset</i>	INTEGER. Offset for the arrays <i>w</i> and <i>werr</i> , that is the <i>ifirst-offset</i> through <i>ilast-offset</i> elements of these arrays are to be used.
<i>w</i>	REAL for slarrj DOUBLE PRECISION for dlarrj Array, DIMENSION (<i>n</i>). On input, <i>w(ifirst-offset)</i> through <i>w(ilast-offset)</i> are estimates of the eigenvalues of L^*D*L^T indexed <i>ifirst</i> through <i>ilast</i> .
<i>werr</i>	REAL for slarrj DOUBLE PRECISION for dlarrj Array, DIMENSION (<i>n</i>). On input, <i>werr(ifirst-offset)</i> through <i>werr(ilast-offset)</i> are the errors in the estimates of the corresponding elements in <i>w</i> .
<i>work</i>	REAL for slarrj DOUBLE PRECISION for dlarrj Workspace array, DIMENSION ($2*n$).
<i>iwork</i>	INTEGER. Workspace array, DIMENSION ($2*n$).
<i>pivmin</i>	REAL for slarrj DOUBLE PRECISION for dlarrj The minimum pivot in the Sturm sequence for the matrix <i>T</i> .
<i>spdiam</i>	REAL for slarrj DOUBLE PRECISION for dlarrj The spectral diameter of the matrix <i>T</i> .

Output Parameters

<i>w</i>	On exit, contains the refined estimates of the eigenvalues.

<code>werr</code>	On exit, contains the refined errors in the estimates of the corresponding elements in w .
<code>info</code>	INTEGER. Now it is not used and always is set to 0.

?larrk

Computes one eigenvalue of a symmetric tridiagonal matrix T to suitable accuracy.

Syntax

```
call slarrk( n, iw, gl, gu, d, e2, pivmin, reltol, w, werr, info )
call dlarrk( n, iw, gl, gu, d, e2, pivmin, reltol, w, werr, info )
```

Description

The routine computes one eigenvalue of a symmetric tridiagonal matrix T to suitable accuracy. This is an auxiliary code to be called from `?stemr`.

To avoid overflow, the matrix must be scaled so that its largest element is no greater than $(\text{overflow}^{1/2} * \text{underflow}^{1/4})$ in absolute value, and for greatest accuracy, it should not be much smaller than that. (For more details see [Kahan66].)

Input Parameters

n	INTEGER. The order of the matrix T . ($n \geq 1$).
iw	INTEGER. The index of the eigenvalue to be returned.
gl, gu	REAL for <code>slarrk</code> DOUBLE PRECISION for <code>dlarrk</code> An upper and a lower bound on the eigenvalue.
d	REAL for <code>slarrk</code> DOUBLE PRECISION for <code>dlarrk</code> Array, DIMENSION (n). Contains n diagonal elements of the matrix T .
$e2$	REAL for <code>slarrk</code> DOUBLE PRECISION for <code>dlarrk</code> Array, DIMENSION ($n-1$).

<i>pivmin</i>	Contains $(n-1)$ squared off-diagonal elements of the T. REAL for slarrk DOUBLE PRECISION for dlarrk The minimum pivot in the Sturm sequence for the matrix T.
<i>reltol</i>	REAL for slarrk DOUBLE PRECISION for dlarrk The minimum relative width of an interval. When an interval is narrower than <i>reltol</i> times the larger (in magnitude) endpoint, then it is considered to be sufficiently small, that is converged. Note: this should always be at least <i>radix*machine epsilon</i>.

Output Parameters

<i>w</i>	REAL for slarrk DOUBLE PRECISION for dlarrk Contains the eigenvalue approximation.
<i>werr</i>	REAL for slarrk DOUBLE PRECISION for dlarrk Contains the error bound on the corresponding eigenvalue approximation in w.
<i>info</i>	INTEGER. = 0: Eigenvalue converges = -1: Eigenvalue does not converge

?larr

Performs tests to decide whether the symmetric tridiagonal matrix T warrants expensive computations which guarantee high relative accuracy in the eigenvalues.

Syntax

```
call slarr( n, d, e, info )
call dlarr( n, d, e, info )
```

Description

The routine performs tests to decide whether the symmetric tridiagonal matrix T warrants expensive computations which guarantee high relative accuracy in the eigenvalues.

Input Parameters

n INTEGER. The order of the matrix T . ($n > 0$).

d REAL for slarrv
DOUBLE PRECISION for dlarrv
Array, DIMENSION (n).
Contains n diagonal elements of the matrix T .

e REAL for slarrv
DOUBLE PRECISION for dlarrv
Array, DIMENSION (n).
The first $(n-1)$ entries contain sub-diagonal elements of the tridiagonal matrix T ; $e(n)$ is set to 0.

Output Parameters

$info$ INTEGER.
= 0: the matrix warrants computations preserving relative accuracy (default value).
= -1: the matrix warrants computations guaranteeing only absolute accuracy.

?larrv

Computes the eigenvectors of the tridiagonal matrix $T = L^*D^*L^T$ given L , D and the eigenvalues of $L^*D^*L^T$.

Syntax

```
call slarrv( n, vl, vu, d, l, pivmin, isplit, m, dol, dou, minrgp, rtol1,
            rtol2, w, werr, wgap, iblock, indexw, gers, z, ldz, isuppz, work, iwork, info
            )

call dlarrv( n, vl, vu, d, l, pivmin, isplit, m, dol, dou, minrgp, rtol1,
            rtol2, w, werr, wgap, iblock, indexw, gers, z, ldz, isuppz, work, iwork, info
            )

call clarrv( n, vl, vu, d, l, pivmin, isplit, m, dol, dou, minrgp, rtol1,
            rtol2, w, werr, wgap, iblock, indexw, gers, z, ldz, isuppz, work, iwork, info
            )

call zlarrv( n, vl, vu, d, l, pivmin, isplit, m, dol, dou, minrgp, rtol1,
            rtol2, w, werr, wgap, iblock, indexw, gers, z, ldz, isuppz, work, iwork, info
            )
```

Description

The routine ?larrv computes the eigenvectors of the tridiagonal matrix $T = L^*D^*L^T$ given L , D and approximations to the eigenvalues of $L^*D^*L^T$.

The input eigenvalues should have been computed by slarre for real flavors (slarrv/clarrv) and by dlarre for double precision flavors (dlarrv/zlarrv).

Input Parameters

n	INTEGER. The order of the matrix. $n \geq 0$.
vl, vu	REAL for slarrv/clarrv DOUBLE PRECISION for dlarrv/zlarrv Lower and upper bounds respectively of the interval that contains the desired eigenvalues. $vl < vu$. Needed to compute gaps on the left or right end of the extremal eigenvalues in the desired range.
d	REAL for slarrv/clarrv

DOUBLE PRECISION for dlarrv/zlarrv
 Array, DIMENSION (n). On entry, the n diagonal elements of the diagonal matrix D .

l REAL for slarrv/clarrv
 DOUBLE PRECISION for dlarrv/zlarrv
 Array, DIMENSION (n).
 On entry, the $(n-1)$ subdiagonal elements of the unit bidiagonal matrix L are contained in elements 1 to $n-1$ of L if the matrix is not splitted. At the end of each block the corresponding shift is stored as given by slarre for real flavors and by dlarre for double precision flavors.

pivmin REAL for slarrv/clarrv
 DOUBLE PRECISION for dlarrv/zlarrv
 The minimum pivot allowed in the Sturm sequence.

isplit INTEGER. Array, DIMENSION (n).
 The splitting points, at which T breaks up into blocks. The first block consists of rows/columns 1 to isplit(1), the second of rows/columns isplit(1)+1 through isplit(2), etc.

m INTEGER. The total number of eigenvalues found.
 $0 \leq m \leq n$. If range = 'A', $m = n$, and if range = 'I', $m = iu - il + 1$.

dol, dou INTEGER.
 If you want to compute only selected eigenvectors from all the eigenvalues supplied, specify an index range dol:dou. Or else apply the setting dol=1, dou=m. Note that dol and dou refer to the order in which the eigenvalues are stored in w .
 If you want to compute only selected eigenpairs, then the columns dol-1 to dou+1 of the eigenvector space Z contain the computed eigenvectors. All other columns of Z are set to zero.

minrgp, rtol1, rtol2 REAL for slarrv/clarrv
 DOUBLE PRECISION for dlarrv/zlarrv
 Parameters for bisection. An interval [LEFT,RIGHT] has converged if RIGHT-LEFT.LT.MAX(rtol1*gap, rtol2*max(|LEFT|,|RIGHT|)).

<i>w</i>	REAL for slarrv/clarrv DOUBLE PRECISION for dlarrv/zlarrv Array, DIMENSION (<i>n</i>). The first <i>m</i> elements of <i>w</i> contain the approximate eigenvalues for which eigenvectors are to be computed. The eigenvalues should be grouped by split-off block and ordered from smallest to largest within the block (the output array <i>w</i> from ?larre is expected here). These eigenvalues are set with respect to the shift of the corresponding root representation for their block.
<i>werr</i>	REAL for slarrv/clarrv DOUBLE PRECISION for dlarrv/zlarrv Array, DIMENSION (<i>n</i>). The first <i>m</i> elements contain the semiwidth of the uncertainty interval of the corresponding eigenvalue in <i>w</i>.
<i>wgap</i>	REAL for slarrv/clarrv DOUBLE PRECISION for dlarrv/zlarrv Array, DIMENSION (<i>n</i>). The separation from the right neighbor eigenvalue in <i>w</i>.
<i>iblock</i>	INTEGER. Array, DIMENSION (<i>n</i>). The indices of the blocks (submatrices) associated with the corresponding eigenvalues in <i>w</i>; <i>iblock</i>(<i>i</i>)=1 if eigenvalue <i>w</i>(<i>i</i>) belongs to the first block from the top, =2 if <i>w</i>(<i>i</i>) belongs to the second block, etc.
<i>indexw</i>	INTEGER. Array, DIMENSION (<i>n</i>). The indices of the eigenvalues within each block (submatrix); for example, <i>indexw</i>(<i>i</i>)= 10 and <i>iblock</i>(<i>i</i>)=2 imply that the <i>i</i>-th eigenvalue <i>w</i>(<i>i</i>) is the 10-th eigenvalue in the second block.
<i>gers</i>	REAL for slarrv/clarrv DOUBLE PRECISION for dlarrv/zlarrv Array, DIMENSION (2*<i>n</i>). The <i>n</i> Gerschgorin intervals (the <i>i</i>-th Gerschgorin interval is (<i>gers</i>(2*<i>i</i>-1), <i>gers</i>(2*<i>i</i>)). The Gerschgorin intervals should be computed from the original unshifted matrix.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i>. <i>ldz</i> ≥ 1, and if <i>jobz</i> = 'V', <i>ldz</i> ≥ max(1, <i>n</i>).

work REAL for slarrv/clarrv
DOUBLE PRECISION for dlarrv/zlarrv
Workspace array, DIMENSION (12*n).

iwork INTEGER.
Workspace array, DIMENSION (7*n).

Output Parameters

d On exit, *d* may be overwritten.

l On exit, *l* is overwritten.

w On exit, *w* holds the eigenvalues of the unshifted matrix.

werr On exit, *werr* contains refined values of its input approximations.

wgap On exit, *wgap* contains refined values of its input approximations. Very small gaps are changed.

z REAL for slarrv
DOUBLE PRECISION for dlarrv
COMPLEX for clarrv
COMPLEX*16 for zlarrv
Array, DIMENSION (ldz, max(1,m)).
If *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *T* corresponding to the input eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).



NOTE. The user must ensure that at least max(1,m) columns are supplied in the array *z*.

isuppz INTEGER .
Array, DIMENSION (2*max(1,m)) . The support of the eigenvectors in *z*, that is, the indices indicating the nonzero elements in *z*. The *i*-th eigenvector is nonzero only in elements *isuppz*(2*i*-1) through *isuppz*(2*i*).

info INTEGER.
If *info* = 0: successful exit

If *info* > 0: A problem occurred in ?larrv. If *info* = 5, the Rayleigh Quotient Iteration failed to converge to full accuracy.

If *info* < 0: One of the called subroutines signaled an internal problem. Inspection of the corresponding parameter *info* for further information is required.

- If *info* = -1, there is a problem in ?larrrb when refining a child eigenvalue;
- If *info* = -2, there is a problem in ?larrrf when computing the relatively robust representation (RRR) of a child. When a child is inside a tight cluster, it can be difficult to find an RRR. A partial remedy from the user's point of view is to make the parameter *minrgp* smaller and recompile. However, as the orthogonality of the computed vectors is proportional to $1/\text{minrgp}$, you should be aware that you might be trading in precision when you decrease *minrgp*.
- If *info* = -3, there is a problem in ?larrrb when refining a single eigenvalue after the Rayleigh correction was rejected.

?lartg

Generates a plane rotation with real cosine and real/complex sine.

Syntax

```
call slartg( f, g, cs, sn, r )
```

```
call dlartg( f, g, cs, sn, r )
```

```
call clartg( f, g, cs, sn, r )
```

```
call zlartg( f, g, cs, sn, r )
```

Description

The routine generates a plane rotation so that

$$\begin{bmatrix} cs & sn \\ -\text{conjg}(sn) & cs \end{bmatrix} \cdot \begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

where $cs^2 + |sn|^2 = 1$

This is a slower, more accurate version of the BLAS Level 1 routine `?rotg`, except for the following differences.

For `slartg/dlartg`:

f and g are unchanged on return;

If $g=0$, then $cs=1$ and $sn=0$;

If $f=0$ and $g \neq 0$, then $cs=0$ and $sn=1$ without doing any floating point operations (saves work in `?bdsqr` when there are zeros on the diagonal);

If f exceeds g in magnitude, cs will be positive.

For `clartg/zlartg`:

f and g are unchanged on return;

If $g=0$, then $cs=1$ and $sn=0$;

If $f=0$, then $cs=0$ and sn is chosen so that r is real.

Input Parameters

f, g REAL for `slartg`
 DOUBLE PRECISION for `dlartg`
 COMPLEX for `clartg`
 COMPLEX*16 for `zlartg`
 The first and second component of vector to be rotated.

Output Parameters

cs REAL for `slartg/clartg`
 DOUBLE PRECISION for `dlartg/zlartg`
 The cosine of the rotation.

sn REAL for `slartg`
 DOUBLE PRECISION for `dlartg`
 COMPLEX for `clartg`

	COMPLEX*16 for zlarg
	The sine of the rotation.
r	REAL for slarg
	DOUBLE PRECISION for dlarg
	COMPLEX for clarg
	COMPLEX*16 for zlarg
	The nonzero component of the rotated vector.

?larg

Applies a vector of plane rotations with real cosines and real/complex sines to the elements of a pair of vectors.

Syntax

```
call slarg( n, x, incx, y, incy, c, s, incc )
call dlarg( n, x, incx, y, incy, c, s, incc )
call clarg( n, x, incx, y, incy, c, s, incc )
call zlarg( n, x, incx, y, incy, c, s, incc )
```

Description

The routine applies a vector of real/complex plane rotations with real cosines to elements of the real/complex vectors x and y . For $i = 1, 2, \dots, n$

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} := \begin{bmatrix} c(i) & s(i) \\ -\text{conjg}(s(i)) & c(i) \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

Input Parameters

n	INTEGER. The number of plane rotations to be applied.
x, y	REAL for slarg DOUBLE PRECISION for dlarg COMPLEX for clarg COMPLEX*16 for zlarg

Arrays, DIMENSION $(1+(n-1)*incx)$ and $(1+(n-1)*incy)$, respectively. The input vectors x and y .

incx INTEGER. The increment between elements of x . $incx > 0$.

incy INTEGER. The increment between elements of y . $incy > 0$.

c REAL for slartv/clartv
DOUBLE PRECISION for dlartv/zlartv
Array, DIMENSION $(1+(n-1)*incc)$.
The cosines of the plane rotations.

s REAL for slartv
DOUBLE PRECISION for dlartv
COMPLEX for clartv
COMPLEX*16 for zlartv
Array, DIMENSION $(1+(n-1)*incc)$.
The sines of the plane rotations.

incc INTEGER. The increment between elements of c and s . $incc > 0$.

Output Parameters

x, y The rotated vectors x and y .

?laruv

Returns a vector of n random real numbers from a uniform distribution.

Syntax

```
call slaruv( iseed, n, x )
call dlaruv( iseed, n, x )
```

Description

The routine ?laruv returns a vector of n random real numbers from a uniform (0,1) distribution ($n \leq 128$).

This is an auxiliary routine called by ?larnv.

Input Parameters

<i>iseed</i>	INTEGER. Array, DIMENSION (4). On entry, the seed of the random number generator; the array elements must be between 0 and 4095, and <i>iseed</i> (4) must be odd.
<i>n</i>	INTEGER. The number of random numbers to be generated. $n \leq 128$.

Output Parameters

<i>x</i>	REAL for slaruv DOUBLE PRECISION for dlaruv Array, DIMENSION (<i>n</i>). The generated random numbers.
<i>seed</i>	On exit, the seed is updated.

?larz

Applies an elementary reflector (as returned by ?tzzrzf) to a general matrix.

Syntax

```
call slarz( side, m, n, l, v, incv, tau, c, ldc, work )
call dlarz( side, m, n, l, v, incv, tau, c, ldc, work )
call clarz( side, m, n, l, v, incv, tau, c, ldc, work )
call zlarz( side, m, n, l, v, incv, tau, c, ldc, work )
```

Description

The routine ?larz applies a real/complex elementary reflector H to a real/complex m -by- n matrix C , from either the left or the right. H is represented in the form

$$H = I - \tau v v',$$

where τ is a real/complex scalar and v is a real/complex vector.

If $\tau = 0$, then H is taken to be the unit matrix.

For complex flavors, to apply H^H (the conjugate transpose of H), supply `conjg( $\tau$ )` instead of τ .

H is a product of k elementary reflectors as returned by ?tzzrzf.

Input Parameters

side CHARACTER*1.
 If *side* = 'L': form H^*C
 If *side* = 'R': form C^*H

m INTEGER. The number of rows of the matrix C .

n INTEGER. The number of columns of the matrix C .

l INTEGER. The number of entries of the vector v containing the meaningful part of the Householder vectors.
 If *side* = 'L', $m \geq L \geq 0$,
 if *side* = 'R', $n \geq L \geq 0$.

v REAL for slarz
 DOUBLE PRECISION for dlarz
 COMPLEX for clarz
 COMPLEX*16 for zlarz
 Array, DIMENSION (1+(*l*-1)*abs(*incv*)).
 The vector v in the representation of H as returned by ?tzzrzf.
v is not used if $\tau = 0$.

incv INTEGER. The increment between elements of v .
incv $\neq 0$.

tau REAL for slarz
 DOUBLE PRECISION for dlarz
 COMPLEX for clarz
 COMPLEX*16 for zlarz
 The value τ in the representation of H .

c REAL for slarz
 DOUBLE PRECISION for dlarz
 COMPLEX for clarz
 COMPLEX*16 for zlarz
 Array, DIMENSION (*ldc*,*n*).
 On entry, the m -by- n matrix C .

ldc INTEGER. The leading dimension of the array c .
ldc $\geq \max(1, m)$.

work REAL for slarz
 DOUBLE PRECISION for dlarz
 COMPLEX for clarz
 COMPLEX*16 for zlarz
 Workspace array, DIMENSION
 (*n*) if *side* = 'L' or
 (*m*) if *side* = 'R'.

Output Parameters

c On exit, *c* is overwritten by the matrix H^*C if *side* = 'L',
 or C^*H if *side* = 'R'.

?larzb

*Applies a block reflector or its
 transpose/conjugate-transpose to a general matrix.*

Syntax

```
call slarzb( side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, c, ldc,  
work, ldwork )
```

```
call dlarzb( side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, c, ldc,  
work, ldwork )
```

```
call clarzb( side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, c, ldc,  
work, ldwork )
```

```
call zlarzb( side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, c, ldc,  
work, ldwork )
```

Description

The routine applies a real/complex block reflector H or its transpose H^T (or H^H for complex flavors) to a real/complex distributed m -by- n matrix C from the left or the right. Currently, only *storev* = 'R' and *direct* = 'B' are supported.

Input Parameters

side CHARACTER*1.
 If *side* = 'L': apply H or H^T/H^H from the left

If $side = 'R'$: apply H or H^T/H^H from the right
trans CHARACTER*1.
If $trans = 'N'$: apply H (No transpose)
If $trans='C'$: apply H^T/H^H (transpose/conjugate transpose)
direct CHARACTER*1.
Indicates how H is formed from a product of elementary reflectors
= 'F': $H = H(1) * H(2) * \dots * H(k)$ (forward, not supported)
= 'B': $H = H(k) * \dots * H(2) * H(1)$ (backward)
storev CHARACTER*1.
Indicates how the vectors which define the elementary reflectors are stored:
= 'C': Column-wise (not supported)
= 'R': Row-wise.
m INTEGER. The number of rows of the matrix C .
n INTEGER. The number of columns of the matrix C .
k INTEGER. The order of the matrix T (equal to the number of elementary reflectors whose product defines the block reflector).
l INTEGER. The number of columns of the matrix v containing the meaningful part of the Householder reflectors.
If $side = 'L'$, $m \geq 1 \geq 0$, if $side = 'R'$, $n \geq 1 \geq 0$.
v REAL for slarzb
DOUBLE PRECISION for dlarzb
COMPLEX for clarzb
COMPLEX*16 for zlarzb
Array, DIMENSION (ldv , nv).
If $storev = 'C'$, $nv = k$;
if $storev = 'R'$, $nv = l$.
ldv INTEGER. The leading dimension of the array v .
If $storev = 'C'$, $ldv \geq l$; if $storev = 'R'$, $ldv \geq k$.
t REAL for slarzb
DOUBLE PRECISION for dlarzb
COMPLEX for clarzb
COMPLEX*16 for zlarzb

	Array, DIMENSION (ldt, k). The triangular k -by- k matrix T in the representation of the block reflector.
ldt	INTEGER. The leading dimension of the array t . $ldt \geq k$.
c	REAL for slarzb DOUBLE PRECISION for dlarzb COMPLEX for clarzb COMPLEX*16 for zlarzb Array, DIMENSION (ldc, n). On entry, the m -by- n matrix C .
ldc	INTEGER. The leading dimension of the array c . $ldc \geq \max(1, m)$.
$work$	REAL for slarzb DOUBLE PRECISION for dlarzb COMPLEX for clarzb COMPLEX*16 for zlarzb Workspace array, DIMENSION ($ldwork, k$).
$ldwork$	INTEGER. The leading dimension of the array $work$. If $side = 'L'$, $ldwork \geq \max(1, n)$; if $side = 'R'$, $ldwork \geq \max(1, m)$.

Output Parameters

c	On exit, c is overwritten by H^*C , or H^T/H^H*C , or C^*H , or C^*H^T/H^H .

slarzt

Forms the triangular factor T of a block reflector H
 $= I - V^* T^* V^H$.

Syntax

```
call slarzt( direct, storev, n, k, v, ldv, tau, t, ldt )
call dlarzt( direct, storev, n, k, v, ldv, tau, t, ldt )
call clarzt( direct, storev, n, k, v, ldv, tau, t, ldt )
call zlarzt( direct, storev, n, k, v, ldv, tau, t, ldt )
```

Description

The routine forms the triangular factor T of a real/complex block reflector H of order $> n$, which is defined as a product of k elementary reflectors.

If $direct = 'F'$, $H = H(1) * H(2) * \dots * H(k)$, and T is upper triangular.

If $direct = 'B'$, $H = H(k) * \dots * H(2) * H(1)$, and T is lower triangular.

If $storev = 'C'$, the vector which defines the elementary reflector $H(i)$ is stored in the i -th column of the array v , and $H = I - V^* T^* V^H$

If $storev = 'R'$, the vector which defines the elementary reflector $H(i)$ is stored in the i -th row of the array v , and $H = I - V^* T^* V^H$

Currently, only $storev = 'R'$ and $direct = 'B'$ are supported.

Input Parameters

<i>direct</i>	CHARACTER*1. Specifies the order in which the elementary reflectors are multiplied to form the block reflector: If $direct = 'F'$: $H = H(1) * H(2) * \dots * H(k)$ (forward, not supported) If $direct = 'B'$: $H = H(k) * \dots * H(2) * H(1)$ (backward)
<i>storev</i>	CHARACTER*1. Specifies how the vectors which define the elementary reflectors are stored (see also <i>Application Notes</i> below): If $storev = 'C'$: column-wise (not supported) If $storev = 'R'$: row-wise

<i>n</i>	INTEGER. The order of the block reflector H . $n \geq 0$.
<i>k</i>	INTEGER. The order of the triangular factor T (equal to the number of elementary reflectors). $k \geq 1$.
<i>v</i>	REAL for slarzt DOUBLE PRECISION for dlarzt COMPLEX for clarzt COMPLEX*16 for zlarzt Array, DIMENSION (<i>ldv</i> , <i>k</i>) if <i>storev</i> = 'C' (<i>ldv</i> , <i>n</i>) if <i>storev</i> = 'R' The matrix V .
<i>ldv</i>	INTEGER. The leading dimension of the array V . If <i>storev</i> = 'C', $ldv \geq \max(1, n)$; if <i>storev</i> = 'R', $ldv \geq k$.
<i>tau</i>	REAL for slarzt DOUBLE PRECISION for dlarzt COMPLEX for clarzt COMPLEX*16 for zlarzt Array, DIMENSION (<i>k</i>). <i>tau</i> (<i>i</i>) must contain the scalar factor of the elementary reflector $H(i)$.
<i>ldt</i>	INTEGER. The leading dimension of the output array t . $ldt \geq k$.

Output Parameters

<i>t</i>	REAL for slarzt DOUBLE PRECISION for dlarzt COMPLEX for clarzt COMPLEX*16 for zlarzt Array, DIMENSION (<i>ldt</i> , <i>k</i>). The k -by- k triangular factor T of the block reflector. If <i>direct</i> = 'F', T is upper triangular; if <i>direct</i> = 'B', T is lower triangular. The rest of the array is not used.
<i>v</i>	The matrix V . See <i>Application Notes</i> below.

Application Notes

The shape of the matrix v and the storage of the vectors which define the $H(i)$ is best illustrated by the following example with $n = 5$ and $k = 3$. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

`direct = 'F' and storev = 'C':` `direct = 'F' and storev = 'R':`

$$v = \begin{bmatrix} v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ 1 & \cdot & \cdot \\ & 1 & \cdot \\ & & 1 \end{bmatrix}$$

$$\begin{array}{c} \text{---}V\text{---} \\ / \qquad \backslash \end{array} \begin{bmatrix} v_1 & v_2 & v_3 & v_1 & v_2 & \cdot & \cdot & \cdot & 1 \\ v_2 & v_2 & v_2 & v_2 & v_2 & \cdot & \cdot & \cdot & 1 \\ v_3 & v_3 & v_3 & v_3 & v_3 & \cdot & \cdot & 1 & \cdot \end{bmatrix}$$

`direct = 'B' and storev = 'C':` `direct = 'B' and storev = 'R':`

$$V = \begin{bmatrix} 1 & & \\ & 1 & \\ & & 1 \end{bmatrix} \begin{bmatrix} v_1 & v_1 & v_1 \\ v_1 & v_1 & v_1 \\ v_1 & v_1 & v_1 \\ v_1 & v_1 & v_1 \\ v_1 & v_1 & v_1 \end{bmatrix}$$

$$\begin{matrix} & \text{---} V \text{---} \\ / & & \backslash \end{matrix} \begin{bmatrix} 1 & & & & v_1 & v_1 & v_1 & v_1 & v_1 \\ & 1 & & & v_1 & v_1 & v_1 & v_1 & v_1 \\ & & 1 & & v_1 & v_1 & v_1 & v_1 & v_1 \end{bmatrix}$$

?las2

Computes singular values of a 2-by-2 triangular matrix.

Syntax

`call slas2( f, g, h, ssmin, ssmax )`

`call dlas2( f, g, h, ssmin, ssmax )`

Description

The routine ?las2 computes the singular values of the 2-by-2 matrix

$$\begin{bmatrix} f & g \\ 0 & h \end{bmatrix}$$

On return, *ssmin* is the smaller singular value and *SSMAX* is the larger singular value.

Input Parameters

f, g, h REAL for `slas2`
 DOUBLE PRECISION for `dlas2`
 The (1,1), (1,2) and (2,2) elements of the 2-by-2 matrix,
 respectively.

Output Parameters

$ssmin, ssmax$ REAL for `slas2`
 DOUBLE PRECISION for `dlas2`
 The smaller and the larger singular values, respectively.

Application Notes

Barring over/underflow, all output quantities are correct to within a few units in the last place (*ulps*), even in the absence of a guard digit in addition/subtraction. In ieee arithmetic, the code works correctly if one matrix element is infinite. Overflow will not occur unless the largest singular value itself overflows, or is within a few ulps of overflow. (On machines with partial overflow, like the Cray, overflow may occur if the largest singular value is within a factor of 2 of overflow.) Underflow is harmless if underflow is gradual. Otherwise, results may correspond to a matrix modified by perturbations of size near the underflow threshold.

?lascl

Multiplies a general rectangular matrix by a real scalar defined as c_{to}/c_{from} .

Syntax

```
call slascl( type, kl, ku, cfrom, cto, m, n, a, lda, info )
call dlascl( type, kl, ku, cfrom, cto, m, n, a, lda, info )
call clascl( type, kl, ku, cfrom, cto, m, n, a, lda, info )
call zlascl( type, kl, ku, cfrom, cto, m, n, a, lda, info )
```

Description

The routine `?lascl` multiplies the m -by- n real/complex matrix A by the real scalar c_{to}/c_{from} . The operation is performed without over/underflow as long as the final result $c_{to} * A(i, j) / c_{from}$ does not over/underflow.

type specifies that *A* may be full, upper triangular, lower triangular, upper Hessenberg, or banded.

Input Parameters

<i>type</i>	CHARACTER*1. This parameter specifies the storage type of the input matrix. = 'G': <i>A</i> is a full matrix. = 'L': <i>A</i> is a lower triangular matrix. = 'U': <i>A</i> is an upper triangular matrix. = 'H': <i>A</i> is an upper Hessenberg matrix. = 'B': <i>A</i> is a symmetric band matrix with lower bandwidth <i>kl</i> and upper bandwidth <i>ku</i> and with the only the lower half stored = 'Q': <i>A</i> is a symmetric band matrix with lower bandwidth <i>kl</i> and upper bandwidth <i>ku</i> and with the only the upper half stored. = 'Z': <i>A</i> is a band matrix with lower bandwidth <i>kl</i> and upper bandwidth <i>ku</i>.
<i>kl</i>	INTEGER. The lower bandwidth of <i>A</i>. Referenced only if <i>type</i> = 'B', 'Q' or 'Z'.
<i>ku</i>	INTEGER. The upper bandwidth of <i>A</i>. Referenced only if <i>type</i> = 'B', 'Q' or 'Z'.
<i>cfrom, cto</i>	REAL for <code>slascl/clascl</code> DOUBLE PRECISION for <code>dlascl/zlascl</code> The matrix <i>A</i> is multiplied by <i>cto/cfrom</i>. <i>A</i>(<i>i</i>, <i>j</i>) is computed without over/underflow if the final result <i>cto</i>*<i>A</i>(<i>i</i>, <i>j</i>) / <i>cfrom</i> can be represented without over/underflow. <i>cfrom</i> must be nonzero.
<i>m</i>	INTEGER. The number of rows of the matrix <i>A</i>. $m \geq 0$.
<i>n</i>	INTEGER. The number of columns of the matrix <i>A</i>. $n \geq 0$.
<i>a</i>	REAL for <code>slascl</code> DOUBLE PRECISION for <code>dlascl</code> COMPLEX for <code>clascl</code> COMPLEX*16 for <code>zlascl</code> Array, DIMENSION (<i>lda</i>, <i>n</i>). The matrix to be multiplied by <i>cto/cfrom</i>. See <i>type</i> for the storage type.

lda INTEGER. The leading dimension of the array *a*.
 $lda \geq \max(1, m)$.

Output Parameters

a The multiplied matrix *A*.
info INTEGER.
 If *info* = 0 - successful exit
 If *info* = -*i* < 0, the *i*-th argument had an illegal value.

?lasd0

Computes the singular values of a real upper bidiagonal n -by- m matrix B with diagonal d and off-diagonal e . Used by ?bdsdc.

Syntax

```
call slasd0( n, sqre, d, e, u, ldu, vt, ldvt, smlsiz, iwork, work, info )
call dlasd0( n, sqre, d, e, u, ldu, vt, ldvt, smlsiz, iwork, work, info )
```

Description

Using a divide and conquer approach, the routine ?lasd0 computes the singular value decomposition (SVD) of a real upper bidiagonal n -by- m matrix B with diagonal d and offdiagonal e , where $m = n + sqre$.

The algorithm computes orthogonal matrices U and VT such that $B = U*S*VT$. The singular values s are overwritten on d .

The related subroutine ?lasda computes only the singular values, and optionally, the singular vectors in compact form.

Input Parameters

n INTEGER. On entry, the row dimension of the upper bidiagonal matrix. This is also the dimension of the main diagonal array d .
sqre INTEGER. Specifies the column dimension of the bidiagonal matrix.

	If $s_{qre} = 0$: the bidiagonal matrix has column dimension $m = n$. If $s_{qre} = 1$: the bidiagonal matrix has column dimension $m = n+1$.
<i>d</i>	REAL for slasd0 DOUBLE PRECISION for dlasd0 Array, DIMENSION (n). On entry, <i>d</i> contains the main diagonal of the bidiagonal matrix.
<i>e</i>	REAL for slasd0 DOUBLE PRECISION for dlasd0 Array, DIMENSION ($m-1$). Contains the subdiagonal entries of the bidiagonal matrix. On exit, <i>e</i> is destroyed.
<i>ldu</i>	INTEGER. On entry, leading dimension of the output array <i>u</i>.
<i>ldvt</i>	INTEGER. On entry, leading dimension of the output array <i>vt</i>.
<i>smlsiz</i>	INTEGER. On entry, maximum size of the subproblems at the bottom of the computation tree.
<i>iwork</i>	INTEGER. Workspace array, dimension must be at least $(8n)$.
<i>work</i>	REAL for slasd0 DOUBLE PRECISION for dlasd0 Workspace array, dimension must be at least $(3m^2+2m)$.

Output Parameters

<i>d</i>	On exit <i>d</i>, If $info = 0$, contains singular values of the bidiagonal matrix.
<i>u</i>	REAL for slasd0 DOUBLE PRECISION for dlasd0 Array, DIMENSION at least (ldq, n). On exit, <i>u</i> contains the left singular vectors.
<i>vt</i>	REAL for slasd0 DOUBLE PRECISION for dlasd0 Array, DIMENSION at least $(ldvt, m)$. On exit, <i>vt'</i> contains the right singular vectors.

info INTEGER.
 If *info* = 0: successful exit.
 If *info* = -*i* < 0, the *i*-th argument had an illegal value.
 If *info* = 1, an singular value did not converge.

?lasd1

Computes the SVD of an upper bidiagonal matrix B of the specified size. Used by ?bdsdc.

Syntax

```
call slasd1( nl, nr, sqre, d, alpha, beta, u, ldu, vt, ldvt, idxq, iwork,
work, info )
```

```
call dlasd1( nl, nr, sqre, d, alpha, beta, u, ldu, vt, ldvt, idxq, iwork,
work, info )
```

Description

This routine computes the SVD of an upper bidiagonal n -by- m matrix B , where $n = nl + nr + 1$ and $m = n + sqre$.

The routine ?lasd1 is called from ?lasd0.

A related subroutine ?lasd7 handles the case in which the singular values (and the singular vectors in factored form) are desired.

?lasd1 computes the SVD as follows:

$$VT = U(in) * \begin{bmatrix} D1(in) & 0 & 0 & 0 \\ Z1' & a & Z2' & b \\ 0 & 0 & D2(in) & 0 \end{bmatrix} * VT(in)$$

$$= U(out) * (D(out) \ 0) * VT(out)$$

where $Z' = (Z1' \ a \ Z2' \ b) = u' * VT'$, and u is a vector of dimension m with *alpha* and *beta* in the $nl+1$ and $nl+2$ -th entries and zeros elsewhere; and the entry b is empty if $sqre = 0$.

The left singular vectors of the original matrix are stored in u , and the transpose of the right singular vectors are stored in vt , and the singular values are in d . The algorithm consists of three stages:

1. The first stage consists of deflating the size of the problem when there are multiple singular values or when there are zeros in the z vector. For each such occurrence the dimension of the secular equation problem is reduced by one. This stage is performed by the routine `?lasd2`.
2. The second stage consists of calculating the updated singular values. This is done by finding the square roots of the roots of the secular equation via the routine `?lasd4` (as called by `?lasd3`). This routine also calculates the singular vectors of the current problem.
3. The final stage consists of computing the updated singular vectors directly using the updated singular values. The singular vectors for the current problem are multiplied with the singular vectors from the overall problem.

Input Parameters

<i>nl</i>	INTEGER. The row dimension of the upper block. $nl \geq 1$.
<i>nr</i>	INTEGER. The row dimension of the lower block. $nr \geq 1$.
<i>sqre</i>	INTEGER. If <i>sqre</i> = 0: the lower block is an <i>nr</i> -by- <i>nr</i> square matrix. If <i>sqre</i> = 1: the lower block is an <i>nr</i> -by-(<i>nr</i> +1) rectangular matrix. The bidiagonal matrix has row dimension $n = nl + nr + 1$, and column dimension $m = n + sqre$.
<i>d</i>	REAL for <code>slasd1</code> DOUBLE PRECISION for <code>dlsd1</code> Array, DIMENSION ($nl+nr+1$). $n = nl+nr+1$. On entry $d(1:nl, 1:nl)$ contains the singular values of the upper block; and $d(nl+2:n)$ contains the singular values of the lower block.
<i>alpha</i>	REAL for <code>slasd1</code> DOUBLE PRECISION for <code>dlsd1</code> Contains the diagonal element associated with the added row.
<i>beta</i>	REAL for <code>slasd1</code>

	DOUBLE PRECISION for dlasd1
	Contains the off-diagonal element associated with the added row.
<i>u</i>	REAL for slasd1 DOUBLE PRECISION for dlasd1 Array, DIMENSION (<i>ldu</i> , <i>n</i>). On entry <i>u</i> (1: <i>nl</i> , 1: <i>nl</i>) contains the left singular vectors of the upper block; <i>u</i> (<i>nl</i> +2: <i>n</i> , <i>nl</i> +2: <i>n</i>) contains the left singular vectors of the lower block.
<i>ldu</i>	INTEGER. The leading dimension of the array <i>u</i> . <i>ldu</i> ≥ max(1, <i>n</i>).
<i>vt</i>	REAL for slasd1 DOUBLE PRECISION for dlasd1 Array, DIMENSION (<i>ldvt</i> , <i>m</i>), where <i>m</i> = <i>n</i> + <i>sqr</i> . On entry <i>vt</i> (1: <i>nl</i> +1, 1: <i>nl</i> +1) ' contains the right singular vectors of the upper block; <i>vt</i> (<i>nl</i> +2: <i>m</i> , <i>nl</i> +2: <i>m</i>) ' contains the right singular vectors of the lower block.
<i>ldvt</i>	INTEGER. The leading dimension of the array <i>vt</i> . <i>ldvt</i> ≥ max(1, <i>M</i>).
<i>iwork</i>	INTEGER. Workspace array, DIMENSION (4 <i>n</i>).
<i>work</i>	REAL for slasd1 DOUBLE PRECISION for dlasd1 Workspace array, DIMENSION (3 <i>m</i> ₂ + 2 <i>m</i>).

Output Parameters

<i>d</i>	On exit <i>d</i> (1: <i>n</i>) contains the singular values of the modified matrix.
<i>alpha</i>	On exit, the diagonal element associated with the added row deflated by max(abs(<i>alpha</i>), abs(<i>beta</i>), abs(<i>D</i> (<i>I</i>))), <i>I</i> = 1, <i>n</i> .
<i>beta</i>	On exit, the off-diagonal element associated with the added row deflated by max(abs(<i>alpha</i>), abs(<i>beta</i>), abs(<i>D</i> (<i>I</i>))), <i>I</i> = 1, <i>n</i> .

<code>u</code>	On exit <code>u</code> contains the left singular vectors of the bidiagonal matrix.
<code>vt</code>	On exit <code>vt'</code> contains the right singular vectors of the bidiagonal matrix.
<code>idxq</code>	INTEGER Array, DIMENSION (n). Contains the permutation which will reintegrate the subproblem just solved back into sorted order, that is, $d(idxq(i = 1, n))$ will be in ascending order.
<code>info</code>	INTEGER. If <code>info = 0</code> : successful exit. If <code>info = -i < 0</code> , the i -th argument had an illegal value. If <code>info = 1</code> , an singular value did not converge.

?lasd2

Merges the two sets of singular values together into a single sorted set. Used by ?bdsdc.

Syntax

```
call slasd2( nl, nr, sqre, k, d, z, alpha, beta, u, ldu, vt, ldvt, dsigma,
u2, ldu2, vt2, ldvt2, idxp, idx, idxp, idxq, coltyp, info )

call dlasd2( nl, nr, sqre, k, d, z, alpha, beta, u, ldu, vt, ldvt, dsigma,
u2, ldu2, vt2, ldvt2, idxp, idx, idxp, idxq, coltyp, info )
```

Description

The routine `?lasd2` merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. There are two ways in which deflation can occur: when two or more singular values are close together or if there is a tiny entry in the `z` vector. For each such occurrence the order of the related secular equation problem is reduced by one.

The routine `?lasd2` is called from `?lasd1`.

Input Parameters

`nl` INTEGER. The row dimension of the upper block.
 $nl \geq 1$.

<i>nr</i>	INTEGER. The row dimension of the lower block. $nr \geq 1$.
<i>sqre</i>	INTEGER. If <i>sqre</i> = 0): the lower block is an <i>nr</i>-by-<i>nr</i> square matrix If <i>sqre</i> = 1): the lower block is an <i>nr</i>-by-(<i>nr</i>+1) rectangular matrix. The bidiagonal matrix has $n = n1 + nr + 1$ rows and $m = n + sqre \geq n$ columns.
<i>d</i>	REAL for slasd2 DOUBLE PRECISION for dlasd2 Array, DIMENSION (<i>n</i>). On entry <i>d</i> contains the singular values of the two submatrices to be combined.
<i>alpha</i>	REAL for slasd2 DOUBLE PRECISION for dlasd2 Contains the diagonal element associated with the added row.
<i>beta</i>	REAL for slasd2 DOUBLE PRECISION for dlasd2 Contains the off-diagonal element associated with the added row.
<i>u</i>	REAL for slasd2 DOUBLE PRECISION for dlasd2 Array, DIMENSION (<i>ldu</i>, <i>n</i>). On entry <i>u</i> contains the left singular vectors of two submatrices in the two square blocks with corners at (1,1), (<i>n1</i>, <i>n1</i>), and (<i>n1</i>+2, <i>n1</i>+2), (<i>n</i>,<i>n</i>).
<i>ldu</i>	INTEGER. The leading dimension of the array <i>u</i>. $ldu \geq n$.
<i>ldu2</i>	INTEGER. The leading dimension of the output array <i>u2</i>. $ldu2 \geq n$.
<i>vt</i>	REAL for slasd2 DOUBLE PRECISION for dlasd2 Array, DIMENSION (<i>ldvt</i>, <i>m</i>). On entry, <i>vt'</i> contains the right singular vectors of two submatrices in the two square blocks with corners at (1,1), (<i>n1</i>+1, <i>n1</i>+1), and (<i>n1</i>+2, <i>n1</i>+2), (<i>m</i>, <i>m</i>).

<i>ldvt</i>	INTEGER. The leading dimension of the array <i>vt</i> . $ldvt \geq m$.
<i>ldvt2</i>	INTEGER. The leading dimension of the output array <i>vt2</i> . $ldvt2 \geq m$.
<i>idxp</i>	INTEGER. Workspace array, DIMENSION (<i>n</i>). This will contain the permutation used to place deflated values of <i>D</i> at the end of the array. On output <i>idxp</i> (2: <i>k</i>) points to the nondeflated <i>d</i> -values and <i>idxp</i> (<i>k</i> +1: <i>n</i>) points to the deflated singular values.
<i>idx</i>	INTEGER. Workspace array, DIMENSION (<i>n</i>). This will contain the permutation used to sort the contents of <i>d</i> into ascending order.
<i>coltyp</i>	INTEGER. Workspace array, DIMENSION (<i>n</i>). As workspace, this array contains a label that indicates which of the following types a column in the <i>u2</i> matrix or a row in the <i>vt2</i> matrix is: 1 : non-zero in the upper half only 2 : non-zero in the lower half only 3 : dense 4 : deflated.
<i>idxq</i>	INTEGER. Array, DIMENSION (<i>n</i>). This parameter contains the permutation that separately sorts the two sub-problems in <i>D</i> into ascending order. Note that entries in the first half of this permutation must first be moved one position backward; and entries in the second half must first have <i>n</i> l+1 added to their values.

Output Parameters

<i>k</i>	INTEGER. Contains the dimension of the non-deflated matrix, This is the order of the related secular equation. $1 \leq k \leq n$.
<i>d</i>	On exit <i>D</i> contains the trailing (<i>n</i> - <i>k</i>) updated singular values (those which were deflated) sorted into increasing order.

<i>u</i>	On exit <i>u</i> contains the trailing $(n-k)$ updated left singular vectors (those which were deflated) in its last $n-k$ columns.
<i>z</i>	REAL for <code>slasd2</code> DOUBLE PRECISION for <code>dlsd2</code> Array, DIMENSION (<i>n</i>). On exit, <i>z</i> contains the updating row vector in the secular equation.
<i>dsigma</i>	REAL for <code>slasd2</code> DOUBLE PRECISION for <code>dlsd2</code> Array, DIMENSION (<i>n</i>). Contains a copy of the diagonal elements ($k-1$ singular values and one zero) in the secular equation.
<i>u2</i>	REAL for <code>slasd2</code> DOUBLE PRECISION for <code>dlsd2</code> Array, DIMENSION (<i>ldu2</i> , <i>n</i>). Contains a copy of the first $k-1$ left singular vectors which will be used by <code>?lasd3</code> in a matrix multiply (<code>?gemm</code>) to solve for the new left singular vectors. <i>u2</i> is arranged into four blocks. The first block contains a column with 1 at $n1+1$ and zero everywhere else; the second block contains non-zero entries only at and above $n1$; the third contains non-zero entries only below $n1+1$; and the fourth is dense.
<i>vt</i>	On exit, <i>vt'</i> contains the trailing $(n-k)$ updated right singular vectors (those which were deflated) in its last $n-k$ columns. In case <i>sqr</i> = 1, the last row of <i>vt</i> spans the right null space.
<i>vt2</i>	REAL for <code>slasd2</code> DOUBLE PRECISION for <code>dlsd2</code> Array, DIMENSION (<i>ldvt2</i> , <i>n</i>). <i>vt2'</i> contains a copy of the first k right singular vectors which will be used by <code>?lasd3</code> in a matrix multiply (<code>?gemm</code>) to solve for the new right singular vectors. <i>vt2</i> is arranged into three blocks. The first block contains a row that corresponds to the special 0 diagonal element in <i>sigma</i> ; the second block contains non-zeros only at and before $n1 + 1$; the third block contains non-zeros only at and after $n1 + 2$.

<i>idxc</i>	INTEGER. Array, DIMENSION (<i>n</i>). This will contain the permutation used to arrange the columns of the deflated <i>u</i> matrix into three groups: the first group contains non-zero entries only at and above <i>nl</i> , the second contains non-zero entries only below <i>nl+2</i> , and the third is dense.
<i>coltyp</i>	On exit, it is an array of dimension 4, with <i>coltyp</i> (<i>i</i>) being the dimension of the <i>i</i> -th type columns.
<i>info</i>	INTEGER. If <i>info</i> = 0): successful exit If <i>info</i> = - <i>i</i> < 0, the <i>i</i> -th argument had an illegal value.

?lasd3

Finds all square roots of the roots of the secular equation, as defined by the values in D and Z, and then updates the singular vectors by matrix multiplication. Used by ?bdsdc.

Syntax

```
call slasd3( nl, nr, sqre, k, d, q, ldq, dsigma, u, ldu, u2, ldu2, vt, ldvt,
vt2, ldvt2, idxc, ctot, z, info )

call dlasd3( nl, nr, sqre, k, d, q, ldq, dsigma, u, ldu, u2, ldu2, vt, ldvt,
vt2, ldvt2, idxc, ctot, z, info )
```

Description

The routine ?lasd3 finds all the square roots of the roots of the secular equation, as defined by the values in *D* and *Z*.

It makes the appropriate calls to ?lasd4 and then updates the singular vectors by matrix multiplication.

The routine ?lasd3 is called from ?lasd1.

Input Parameters

<i>nl</i>	INTEGER. The row dimension of the upper block. $nl \geq 1$.
<i>nr</i>	INTEGER. The row dimension of the lower block.

$nr \geq 1$.

sqre INTEGER.
 If $sqre = 0$): the lower block is an nr -by- nr square matrix.
 If $sqre = 1$): the lower block is an nr -by- $(nr+1)$ rectangular matrix. The bidiagonal matrix has $n = n1 + nr + 1$ rows and $m = n + sqre \geq n$ columns.

k INTEGER. The size of the secular equation, $1 \leq k \leq n$.

q REAL for slasd3
 DOUBLE PRECISION for dlasd3
 Workspace array, DIMENSION at least (ldq, k).

ldq INTEGER. The leading dimension of the array *q*.
 $ldq \geq k$.

dsigma REAL for slasd3
 DOUBLE PRECISION for dlasd3
 Array, DIMENSION (k). The first k elements of this array contain the old roots of the deflated updating problem. These are the poles of the secular equation.

ldu INTEGER. The leading dimension of the array *u*.
 $ldu \geq n$.

u2 REAL for slasd3
 DOUBLE PRECISION for dlasd3
 Array, DIMENSION ($ldu2, n$).
 The first k columns of this matrix contain the non-deflated left singular vectors for the split problem.

ldu2 INTEGER. The leading dimension of the array *u2*.
 $ldu2 \geq n$.

ldvt INTEGER. The leading dimension of the array *vt*.
 $ldvt \geq n$.

vt2 REAL for slasd3
 DOUBLE PRECISION for dlasd3
 Array, DIMENSION ($ldvt2, n$).
 The first k columns of *vt2'* contain the non-deflated right singular vectors for the split problem.

<i>ldvt2</i>	INTEGER. The leading dimension of the array <i>vt2</i> . $ldvt2 \geq n$.
<i>idxc</i>	INTEGER. Array, DIMENSION (<i>n</i>). The permutation used to arrange the columns of <i>u</i> (and rows of <i>vt</i>) into three groups: the first group contains non-zero entries only at and above (or before) <i>nl + 1</i> ; the second contains non-zero entries only at and below (or after) <i>nl+2</i> ; and the third is dense. The first column of <i>u</i> and the row of <i>vt</i> are treated separately, however. The rows of the singular vectors found by ?lasd4 must be likewise permuted before the matrix multiplies can take place.
<i>ctot</i>	INTEGER. Array, DIMENSION (4). A count of the total number of the various types of columns in <i>u</i> (or rows in <i>vt</i>), as described in <i>idxc</i> . The fourth column type is any column which has been deflated.
<i>z</i>	REAL for slasd3 DOUBLE PRECISION for dlasd3 Array, DIMENSION (<i>k</i>). The first <i>k</i> elements of this array contain the components of the deflation-adjusted updating row vector.

Output Parameters

<i>d</i>	REAL for slasd3 DOUBLE PRECISION for dlasd3 Array, DIMENSION (<i>k</i>). On exit the square roots of the roots of the secular equation, in ascending order.
<i>u</i>	REAL for slasd3 DOUBLE PRECISION for dlasd3 Array, DIMENSION (<i>ldu</i> , <i>n</i>). The last <i>n - k</i> columns of this matrix contain the deflated left singular vectors.
<i>vt</i>	REAL for slasd3 DOUBLE PRECISION for dlasd3 Array, DIMENSION (<i>ldvt</i> , <i>m</i>).

The last $m - k$ columns of vt' contain the deflated right singular vectors.

vt2 Destroyed on exit.

z Destroyed on exit.

info INTEGER.
 If *info* = 0): successful exit.
 If *info* = -*i* < 0, the *i*-th argument had an illegal value.
 If *info* = 1, an singular value did not converge.

Application Notes

This code makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray XMP, Cray YMP, Cray C 90, or Cray 2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

?lasd4

*Computes the square root of the *i*-th updated eigenvalue of a positive symmetric rank-one modification to a positive diagonal matrix. Used by ?bdsdc.*

Syntax

```
call slasd4( n, i, d, z, delta, rho, sigma, work, info )
call dlasd4( n, i, d, z, delta, rho, sigma, work, info )
```

Description

This routine computes the square root of the *i*-th updated eigenvalue of a positive symmetric rank-one modification to a positive diagonal matrix whose entries are given as the squares of the corresponding entries in the array *d*, and that $0 \leq d(i) < d(j)$ for $i < j$ and that $\rho > 0$. This is arranged by the calling routine, and is no loss in generality. The rank-one modified system is thus

$$\text{diag}(d) * \text{diag}(d) + \rho * z * z^T,$$

where the Euclidean norm of *z* is equal to 1. The method consists of approximating the rational functions in the secular equation by simpler interpolating rational functions.

Input Parameters

<i>n</i>	INTEGER. The length of all arrays.
<i>i</i>	INTEGER. The index of the eigenvalue to be computed. $1 \leq i \leq n$.
<i>d</i>	REAL for slasd4 DOUBLE PRECISION for dlasd4 Array, DIMENSION (<i>n</i>). The original eigenvalues. They must be in order, $0 \leq d(i) < d(j)$ for $i < j$.
<i>z</i>	REAL for slasd4 DOUBLE PRECISION for dlasd4 Array, DIMENSION (<i>n</i>). The components of the updating vector.
<i>rho</i>	REAL for slasd4 DOUBLE PRECISION for dlasd4 The scalar in the symmetric updating formula.
<i>work</i>	REAL for slasd4 DOUBLE PRECISION for dlasd4 Workspace array, DIMENSION (<i>n</i>). If $n \neq 1$, <i>work</i> contains ($d(j) + \text{sigma}_i$) in its <i>j</i> -th component. If $n = 1$, then <i>work</i> (1) = 1.

Output Parameters

<i>delta</i>	REAL for slasd4 DOUBLE PRECISION for dlasd4 Array, DIMENSION (<i>n</i>). If $n \neq 1$, <i>delta</i> contains ($d(j) - \text{sigma}_i$) in its <i>j</i> -th component. If $n = 1$, then <i>delta</i> (1) = 1. The vector <i>delta</i> contains the information necessary to construct the (singular) eigenvectors.
<i>sigma</i>	REAL for slasd4 DOUBLE PRECISION for dlasd4

info The computed σ_i , the i -th updated eigenvalue.
 INTEGER.
 = 0: successful exit
 > 0: If *info* = 1, the updating process failed.

?lasd5

Computes the square root of the i -th eigenvalue of a positive symmetric rank-one modification of a 2-by-2 diagonal matrix. Used by ?bdsdc.

Syntax

```
call slasd5( i, d, z, delta, rho, dsigma, work )
call dlasd5( i, d, z, delta, rho, dsigma, work )
```

Description

This routine computes the square root of the i -th eigenvalue of a positive symmetric rank-one modification of a 2-by-2 diagonal matrix $\text{diag}(d) * \text{diag}(d) + \rho * Z * Z^T$

The diagonal entries in the array d must satisfy $0 \leq d(i) < d(j)$ for $i < j$, ρ must be greater than 0, and that the Euclidean norm of the vector z is equal to 1.

Input Parameters

i *INTEGER*. The index of the eigenvalue to be computed. $i = 1$ or $i = 2$.

d *REAL* for *slasd5*
 DOUBLE PRECISION for *dlasd5*
 Array, dimension (2).
 The original eigenvalues, $0 \leq d(1) < d(2)$.

z *REAL* for *slasd5*
 DOUBLE PRECISION for *dlasd5*
 Array, dimension (2).
 The components of the updating vector.

rho *REAL* for *slasd5*
 DOUBLE PRECISION for *dlasd5*
 The scalar in the symmetric updating formula.

work REAL for slasd5
DOUBLE PRECISION for dlasd5.
Workspace array, dimension (2). Contains ($d(j) + \sigma_i$) in its j -th component.

Output Parameters

delta REAL for slasd5
DOUBLE PRECISION for dlasd5.
Array, dimension (2).
Contains ($d(j) - \sigma_i$) in its j -th component. The vector *delta* contains the information necessary to construct the eigenvectors.

dsigma REAL for slasd5
DOUBLE PRECISION for dlasd5.
The computed σ_i , the i -th updated eigenvalue.

?lasd6

Computes the SVD of an updated upper bidiagonal matrix obtained by merging two smaller ones by appending a row. Used by ?bdsdc.

Syntax

```
call slasd6( icompg, nl, nr, sqre, d, vf, vl, alpha, beta, idxq, perm, givptr,
            givcol, ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, work, iwork,
            info )
```

```
call dlasd6( icompg, nl, nr, sqre, d, vf, vl, alpha, beta, idxq, perm, givptr,
            givcol, ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, work, iwork,
            info )
```

Description

The routine ?lasd6 computes the *SVD* of an updated upper bidiagonal matrix B obtained by merging two smaller ones by appending a row. This routine is used only for the problem which requires all singular values and optionally singular vector matrices in factored form. B is an n -by- m matrix with $n = nl + nr + 1$ and $m = n + sqre$. A related subroutine, ?lasd1, handles the case in which all singular values and singular vectors of the bidiagonal matrix are desired. ?lasd6 computes the *SVD* as follows:

$$B = U(in) * \begin{bmatrix} D1(in) & 0 & 0 & 0 \\ Z1' & a & Z2' & b \\ 0 & 0 & D2(in) & 0 \end{bmatrix} * VT(in)$$

$$= U(out) * (D(out) * VT(out))$$

where $Z' = (Z1' \ a \ Z2' \ b) = u' * VT'$, and u is a vector of dimension m with $alpha$ and $beta$ in the $nl+1$ and $nl+2$ -th entries and zeros elsewhere; and the entry b is empty if $sqr = 0$.

The singular values of B can be computed using $D1$, $D2$, the first components of all the right singular vectors of the lower block, and the last components of all the right singular vectors of the upper block. These components are stored and updated in vf and vl , respectively, in `?lasd6`. Hence U and VT are not explicitly referenced.

The singular values are stored in D . The algorithm consists of two stages:

1. The first stage consists of deflating the size of the problem when there are multiple singular values or if there is a zero in the z vector. For each such occurrence the dimension of the secular equation problem is reduced by one. This stage is performed by the routine `?lasd7`.
2. The second stage consists of calculating the updated singular values. This is done by finding the roots of the secular equation via the routine `?lasd4` (as called by `?lasd8`). This routine also updates vf and vl and computes the distances between the updated singular values and the old singular values. `?lasd6` is called from `?lasda`.

Input Parameters

<i>icompg</i>	INTEGER. Specifies whether singular vectors are to be computed in factored form: = 0: Compute singular values only = 1: Compute singular vectors in factored form as well.
<i>nl</i>	INTEGER. The row dimension of the upper block. $nl \geq 1$.
<i>nr</i>	INTEGER. The row dimension of the lower block. $nr \geq 1$.
<i>sqr</i>	INTEGER. = 0: the lower block is an nr -by- nr square matrix.

	= 1: the lower block is an nr-by-$(nr+1)$ rectangular matrix. The bidiagonal matrix has row dimension $n=nl+nr+1$, and column dimension $m = n + sqre$.
<i>d</i>	REAL for slasd6 DOUBLE PRECISION for dlasd6 Array, dimension ($nl+nr+1$). On entry $d(1:nl,1:nl)$ contains the singular values of the upper block, and $d(nl+2:n)$ contains the singular values of the lower block.
<i>vf</i>	REAL for slasd6 DOUBLE PRECISION for dlasd6 Array, dimension (m). On entry, $vf(1:nl+1)$ contains the first components of all right singular vectors of the upper block; and $vf(nl+2:m)$ contains the first components of all right singular vectors of the lower block.
<i>vl</i>	REAL for slasd6 DOUBLE PRECISION for dlasd6 Array, dimension (m). On entry, $vl(1:nl+1)$ contains the last components of all right singular vectors of the upper block; and $vl(nl+2:m)$ contains the last components of all right singular vectors of the lower block.
<i>alpha</i>	REAL for slasd6 DOUBLE PRECISION for dlasd6 Contains the diagonal element associated with the added row.
<i>beta</i>	REAL for slasd6 DOUBLE PRECISION for dlasd6 Contains the off-diagonal element associated with the added row.
<i>ldgcol</i>	INTEGER. The leading dimension of the output array <i>givcol</i>, must be at least n.
<i>ldgnum</i>	INTEGER. The leading dimension of the output arrays <i>givnum</i> and <i>poles</i>, must be at least n.
<i>work</i>	REAL for slasd6 DOUBLE PRECISION for dlasd6

iwork Workspace array, dimension ($4m$).
 INTEGER.
 Workspace array, dimension ($3n$).

Output Parameters

d On exit $d(1:n)$ contains the singular values of the modified matrix.

vf On exit, *vf* contains the first components of all right singular vectors of the bidiagonal matrix.

vl On exit, *vl* contains the last components of all right singular vectors of the bidiagonal matrix.

alpha On exit, the diagonal element associated with the added row deflated by $\max(\text{abs}(\text{alpha}), \text{abs}(\text{beta}), \text{abs}(D(I))), I = 1, n$.

beta On exit, the off-diagonal element associated with the added row deflated by $\max(\text{abs}(\text{alpha}), \text{abs}(\text{beta}), \text{abs}(D(I))), I = 1, n$.

idxq INTEGER.
 Array, dimension (n). This contains the permutation which will reintegrate the subproblem just solved back into sorted order, that is, $d(\text{idxq}(i = 1, n))$ will be in ascending order.

perm INTEGER.
 Array, dimension (n). The permutations (from deflation and sorting) to be applied to each block. Not referenced if $\text{icompq} = 0$.

givptr INTEGER. The number of Givens rotations which took place in this subproblem. Not referenced if $\text{icompq} = 0$.

givcol INTEGER.
 Array, dimension ($\text{ldgcol}, 2$). Each pair of numbers indicates a pair of columns to take place in a Givens rotation. Not referenced if $\text{icompq} = 0$.

givnum REAL for `slasd6`
 DOUBLE PRECISION for `dlsd6`

	Array, dimension (<i>ldgnum</i>, 2). Each number indicates the <i>c</i> or <i>s</i> value to be used in the corresponding Givens rotation. Not referenced if <i>icompq</i> = 0.
<i>poles</i>	REAL for slasd6 DOUBLE PRECISION for dlasd6 Array, dimension (<i>ldgnum</i>, 2). On exit, <i>poles</i>(1,*) is an array containing the new singular values obtained from solving the secular equation, and <i>poles</i>(2,*) is an array containing the poles in the secular equation. Not referenced if <i>icompq</i> = 0.
<i>difl</i>	REAL for slasd6 DOUBLE PRECISION for dlasd6 Array, dimension (<i>n</i>). On exit, <i>difl</i>(<i>i</i>) is the distance between <i>i</i>-th updated (undeflated) singular value and the <i>i</i>-th (undeflated) old singular value.
<i>difr</i>	REAL for slasd6 DOUBLE PRECISION for dlasd6 Array, dimension (<i>ldgnum</i>, 2) if <i>icompq</i> = 1 and dimension (<i>n</i>) if <i>icompq</i> = 0. On exit, <i>difr</i>(<i>i</i>, 1) is the distance between <i>i</i>-th updated (undeflated) singular value and the <i>i</i>+1-th (undeflated) old singular value. If <i>icompq</i> = 1, <i>difr</i>(1: <i>k</i>, 2) is an array containing the normalizing factors for the right singular vector matrix. See ?lasd8 for details on <i>difl</i> and <i>difr</i>.
<i>z</i>	REAL for slasd6 DOUBLE PRECISION for dlasd6 Array, dimension (<i>m</i>). The first elements of this array contain the components of the deflation-adjusted updating row vector.
<i>k</i>	INTEGER. Contains the dimension of the non-deflated matrix. This is the order of the related secular equation. $1 \leq k \leq n$.
<i>c</i>	REAL for slasd6 DOUBLE PRECISION for dlasd6

c contains garbage if *sqre* = 0 and the *c*-value of a Givens rotation related to the right null space if *sqre* = 1.

s REAL for slasd6
DOUBLE PRECISION for dlasd6
s contains garbage if *sqre* = 0 and the *s*-value of a Givens rotation related to the right null space if *sqre* = 1.

info INTEGER.
= 0: successful exit.
< 0: if *info* = -*i*, the *i*-th argument had an illegal value.
> 0: if *info* = 1, an singular value did not converge

?lasd7

Merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. Used by ?bdsdc.

Syntax

```
call slasd7( icompg, nl, nr, sqre, k, d, z, zw, vf, vfw, vl, vlw, alpha, beta,
dsigma, idx, idxp, idxq, perm, givptr, givcol, ldgcol, givnum, ldgnum, c, s,
info )
```

```
call dlasd7( icompg, nl, nr, sqre, k, d, z, zw, vf, vfw, vl, vlw, alpha, beta,
dsigma, idx, idxp, idxq, perm, givptr, givcol, ldgcol, givnum, ldgnum, c, s,
info )
```

Description

The routine ?lasd7 merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. There are two ways in which deflation can occur: when two or more singular values are close together or if there is a tiny entry in the *z* vector. For each such occurrence the order of the related secular equation problem is reduced by one. ?lasd7 is called from ?lasd6.

Input Parameters

icompg INTEGER. Specifies whether singular vectors are to be computed in compact form, as follows:
= 0: Compute singular values only.

	= 1: Compute singular vectors of upper bidiagonal matrix in compact form.
<i>nl</i>	INTEGER. The row dimension of the upper block. $nl \geq 1$.
<i>nr</i>	INTEGER. The row dimension of the lower block. $nr \geq 1$.
<i>sqre</i>	INTEGER. = 0: the lower block is an <i>nr</i> -by- <i>nr</i> square matrix. = 1: the lower block is an <i>nr</i> -by-(<i>nr</i> +1) rectangular matrix. The bidiagonal matrix has $n = nl + nr + 1$ rows and $m = n + sqre \geq n$ columns.
<i>d</i>	REAL for slasd7 DOUBLE PRECISION for dlasd7 Array, DIMENSION (<i>n</i>). On entry <i>d</i> contains the singular values of the two submatrices to be combined.
<i>zw</i>	REAL for slasd7 DOUBLE PRECISION for dlasd7 Array, DIMENSION (<i>m</i>). Workspace for <i>z</i> .
<i>vf</i>	REAL for slasd7 DOUBLE PRECISION for dlasd7 Array, DIMENSION (<i>m</i>). On entry, <i>vf</i> (1: <i>nl</i> +1) contains the first components of all right singular vectors of the upper block; and <i>vf</i> (<i>nl</i> +2: <i>m</i>) contains the first components of all right singular vectors of the lower block.
<i>vfw</i>	REAL for slasd7 DOUBLE PRECISION for dlasd7 Array, DIMENSION (<i>m</i>). Workspace for <i>vf</i> .
<i>vl</i>	REAL for slasd7 DOUBLE PRECISION for dlasd7 Array, DIMENSION (<i>m</i>).

	On entry, $vl(1:nl+1)$ contains the last components of all right singular vectors of the upper block; and $vl(nl+2:m)$ contains the last components of all right singular vectors of the lower block.
<i>VLW</i>	REAL for <code>slasd7</code> DOUBLE PRECISION for <code>dlsd7</code> Array, DIMENSION (<i>m</i>). Workspace for VL.
<i>alpha</i>	REAL for <code>slasd7</code> DOUBLE PRECISION for <code>dlsd7</code>. Contains the diagonal element associated with the added row.
<i>beta</i>	REAL for <code>slasd7</code> DOUBLE PRECISION for <code>dlsd7</code> Contains the off-diagonal element associated with the added row.
<i>idx</i>	INTEGER. Workspace array, DIMENSION (<i>n</i>). This will contain the permutation used to sort the contents of <i>d</i> into ascending order.
<i>idxp</i>	INTEGER. Workspace array, DIMENSION (<i>n</i>). This will contain the permutation used to place deflated values of <i>d</i> at the end of the array.
<i>idxq</i>	INTEGER. Array, DIMENSION (<i>n</i>). This contains the permutation which separately sorts the two sub-problems in <i>d</i> into ascending order. Note that entries in the first half of this permutation must first be moved one position backward; and entries in the second half must first have $nl+1$ added to their values.
<i>ldgcol</i>	INTEGER. The leading dimension of the output array <i>givcol</i>, must be at least <i>n</i>.
<i>ldgnum</i>	INTEGER. The leading dimension of the output array <i>givnum</i>, must be at least <i>n</i>.

Output Parameters

<i>k</i>	INTEGER. Contains the dimension of the non-deflated matrix, this is the order of the related secular equation. $1 \leq k \leq n$.
<i>d</i>	On exit, <i>d</i> contains the trailing (<i>n-k</i>) updated singular values (those which were deflated) sorted into increasing order.
<i>z</i>	REAL for <code>slasd7</code> DOUBLE PRECISION for <code>dlsd7</code>. Array, DIMENSION (<i>m</i>). On exit, <i>z</i> contains the updating row vector in the secular equation.
<i>vf</i>	On exit, <i>vf</i> contains the first components of all right singular vectors of the bidiagonal matrix.
<i>vl</i>	On exit, <i>vl</i> contains the last components of all right singular vectors of the bidiagonal matrix.
<i>dsigma</i>	REAL for <code>slasd7</code> DOUBLE PRECISION for <code>dlsd7</code>. Array, DIMENSION (<i>n</i>). Contains a copy of the diagonal elements (<i>k</i>-1 singular values and one zero) in the secular equation.
<i>idxp</i>	On output, <i>idxp</i>(2: <i>k</i>) points to the nondeflated <i>d</i>-values and <i>idxp</i>(<i>k</i>+1:<i>n</i>) points to the deflated singular values.
<i>perm</i>	INTEGER. Array, DIMENSION (<i>n</i>). The permutations (from deflation and sorting) to be applied to each singular block. Not referenced if <i>icompq</i> = 0.
<i>givptr</i>	INTEGER. The number of Givens rotations which took place in this subproblem. Not referenced if <i>icompq</i> = 0.
<i>givcol</i>	INTEGER. Array, DIMENSION (<i>ldgcol</i>, 2). Each pair of numbers indicates a pair of columns to take place in a Givens rotation. Not referenced if <i>icompq</i> = 0.
<i>givnum</i>	REAL for <code>slasd7</code> DOUBLE PRECISION for <code>dlsd7</code>.

	Array, DIMENSION (<i>ldgnum</i> , 2). Each number indicates the <i>c</i> or <i>s</i> value to be used in the corresponding Givens rotation. Not referenced if <i>icompq</i> = 0.
<i>c</i>	REAL for <i>slasd7</i> . DOUBLE PRECISION for <i>dlasd7</i> . If <i>sqre</i> = 0, then <i>c</i> contains garbage, and if <i>sqre</i> = 1, then <i>c</i> contains <i>c</i> -value of a Givens rotation related to the right null space.
<i>s</i>	REAL for <i>slasd7</i> . DOUBLE PRECISION for <i>dlasd7</i> . If <i>sqre</i> = 0, then <i>s</i> contains garbage, and if <i>sqre</i> = 1, then <i>s</i> contains <i>s</i> -value of a Givens rotation related to the right null space.
<i>info</i>	INTEGER. = 0: successful exit. < 0: if <i>info</i> = - <i>i</i> , the <i>i</i> -th argument had an illegal value.

?lasd8

*Finds the square roots of the roots of the secular equation, and stores, for each element in *D*, the distance to its two nearest poles. Used by ?bdsdc.*

Syntax

```
call slasd8( icompq, k, d, z, vf, vl, difl, difr, lddifr, dsigma, work, info
)
call dlasd8( icompq, k, d, z, vf, vl, difl, difr, lddifr, dsigma, work, info
)
```

Description

The routine ?lasd8 finds the square roots of the roots of the secular equation, as defined by the values in *dsigma* and *z*. It makes the appropriate calls to ?lasd4, and stores, for each element in *d*, the distance to its two nearest poles (elements in *dsigma*). It also updates the arrays *vf* and *vl*, the first and last components of all the right singular vectors of the original bidiagonal matrix. ?lasd8 is called from ?lasd6.

Input Parameters

<i>icompq</i>	INTEGER. Specifies whether singular vectors are to be computed in factored form in the calling routine: = 0: Compute singular values only. = 1: Compute singular vectors in factored form as well.
<i>k</i>	INTEGER. The number of terms in the rational function to be solved by ?lasd4. $k \geq 1$.
<i>z</i>	REAL for slasd8 DOUBLE PRECISION for dlasd8. Array, DIMENSION (<i>k</i>). The first <i>k</i> elements of this array contain the components of the deflation-adjusted updating row vector.
<i>vf</i>	REAL for slasd8 DOUBLE PRECISION for dlasd8. Array, DIMENSION (<i>k</i>). On entry, <i>vf</i> contains information passed through dbede8.
<i>vl</i>	REAL for slasd8 DOUBLE PRECISION for dlasd8. Array, DIMENSION (<i>k</i>). On entry, <i>vl</i> contains information passed through dbede8.
<i>lddifr</i>	INTEGER. The leading dimension of the output array <i>difr</i>, must be at least <i>k</i>.
<i>dsigma</i>	REAL for slasd8 DOUBLE PRECISION for dlasd8. Array, DIMENSION (<i>k</i>). The first <i>k</i> elements of this array contain the old roots of the deflated updating problem. These are the poles of the secular equation.
<i>work</i>	REAL for slasd8 DOUBLE PRECISION for dlasd8. Workspace array, DIMENSION at least $(3k)$.

Output Parameters

<i>d</i>	REAL for slasd8 DOUBLE PRECISION for dlasd8.

Array, `DIMENSION ( k )`.
 On output, *D* contains the updated singular values.

vf
 On exit, *vf* contains the first *k* components of the first components of all right singular vectors of the bidiagonal matrix.

vl
 On exit, *vl* contains the first *k* components of the last components of all right singular vectors of the bidiagonal matrix.

difl
 REAL for `slasd8`
 DOUBLE PRECISION for `dlasd8`.
 Array, `DIMENSION ( k )`. On exit, $difl(i) = d(i) - dsigma(i)$.

difr
 REAL for `slasd8`
 DOUBLE PRECISION for `dlasd8`.
 Array,
`DIMENSION ( lddifr, 2 )` if *icompg* = 1 and
`DIMENSION ( k )` if *icompg* = 0.
 On exit, $difr(i,1) = d(i) - dsigma(i+1)$, $difr(k,1)$ is not defined and will not be referenced. If *icompg* = 1, $difr(1:k,2)$ is an array containing the normalizing factors for the right singular vector matrix.

info
 INTEGER.
 = 0: successful exit.
 < 0: if *info* = -*i*, the *i*-th argument had an illegal value.
 > 0: If *info* = 1, an singular value did not converge.

?lasd9

*Finds the square roots of the roots of the secular equation, and stores, for each element in *D*, the distance to its two nearest poles. Used by ?bdsdc.*

Syntax

```
call slasd9( icompg, ldu, k, d, z, vf, vl, difl, difr, dsigma, work, info )
call dlasd9( icompg, ldu, k, d, z, vf, vl, difl, difr, dsigma, work, info )
```

Description

The routine `?lasd9` finds the square roots of the roots of the secular equation, as defined by the values in *dsigma* and *z*. It makes the appropriate calls to `?lasd4`, and stores, for each element in *d*, the distance to its two nearest poles (elements in *dsigma*). It also updates the arrays *vf* and *vl*, the first and last components of all the right singular vectors of the original bidiagonal matrix. `?lasd9` is called from `?lasd7`.

Input Parameters

<i>icompg</i>	INTEGER. Specifies whether singular vectors are to be computed in factored form in the calling routine: If <i>icompg</i> = 0, compute singular values only; If <i>icompg</i> = 1, compute singular vector matrices in factored form also.
<i>k</i>	INTEGER. The number of terms in the rational function to be solved by <code>slasd4</code>. $k \geq 1$.
<i>dsigma</i>	REAL for <code>slasd9</code> DOUBLE PRECISION for <code>dlasd9</code>. Array, DIMENSION(<i>k</i>). The first <i>k</i> elements of this array contain the old roots of the deflated updating problem. These are the poles of the secular equation.
<i>z</i>	REAL for <code>slasd9</code> DOUBLE PRECISION for <code>dlasd9</code>. Array, DIMENSION (<i>k</i>). The first <i>k</i> elements of this array contain the components of the deflation-adjusted updating row vector.
<i>vf</i>	REAL for <code>slasd9</code> DOUBLE PRECISION for <code>dlasd9</code>. Array, DIMENSION(<i>k</i>). On entry, <i>vf</i> contains information passed through <code>sbede8</code>.
<i>vl</i>	REAL for <code>slasd9</code> DOUBLE PRECISION for <code>dlasd9</code>. Array, DIMENSION(<i>k</i>). On entry, <i>vl</i> contains information passed through <code>sbede8</code>.
<i>work</i>	REAL for <code>slasd9</code> DOUBLE PRECISION for <code>dlasd9</code>.

Workspace array, `DIMENSION` at least $(3k)$.

Output Parameters

<i>d</i>	REAL for <code>slasd9</code> DOUBLE PRECISION for <code>dlasd9</code>. Array, <code>DIMENSION(k)</code>. <i>d</i>(<i>i</i>) contains the updated singular values.
<i>vf</i>	On exit, <i>vf</i> contains the first <i>k</i> components of the first components of all right singular vectors of the bidiagonal matrix.
<i>vl</i>	On exit, <i>vl</i> contains the first <i>k</i> components of the last components of all right singular vectors of the bidiagonal matrix.
<i>difl</i>	REAL for <code>slasd9</code> DOUBLE PRECISION for <code>dlasd9</code>. Array, <code>DIMENSION(k)</code>. On exit, <i>difl</i>(<i>i</i>) = <i>d</i>(<i>i</i>) - <i>dsigma</i>(<i>i</i>).
<i>difr</i>	REAL for <code>slasd9</code> DOUBLE PRECISION for <code>dlasd9</code>. Array, <code>DIMENSION(ldu, 2)</code> if <i>icompg</i> = 1 and <code>DIMENSION(k)</code> if <i>icompg</i> = 0. On exit, <i>difr</i>(<i>i</i>, 1) = <i>d</i>(<i>i</i>) - <i>dsigma</i>(<i>i</i>+1), <i>difr</i>(<i>k</i>, 1) is not defined and will not be referenced. If <i>icompg</i> = 1, <i>difr</i>(1:<i>k</i>, 2) is an array containing the normalizing factors for the right singular vector matrix.
<i>info</i>	INTEGER. = 0: successful exit. < 0: if <i>info</i> = -<i>i</i>, the <i>i</i>-th argument had an illegal value. > 0: If <i>info</i> = 1, an singular value did not converge

?lasda

Computes the singular value decomposition (SVD) of a real upper bidiagonal matrix with diagonal d and off-diagonal e . Used by ?bdsdc.

Syntax

```
call slasda( icompg, smlsiz, n, sqre, d, e, u, ldu, vt, k, difl, difr, z,
            poles, givptr, givcol, ldgcol, perm, givnum, c, s, work, iwork, info )

call dlasda( icompg, smlsiz, n, sqre, d, e, u, ldu, vt, k, difl, difr, z,
            poles, givptr, givcol, ldgcol, perm, givnum, c, s, work, iwork, info )
```

Description

Using a divide and conquer approach, ?lasda computes the singular value decomposition (SVD) of a real upper bidiagonal n -by- m matrix B with diagonal d and off-diagonal e , where $m = n + sqre$.

The algorithm computes the singular values in the $SVD\ B = U*S*V^T$. The orthogonal matrices U and V^T are optionally computed in compact form. A related subroutine ?lasd0 computes the singular values and the singular vectors in explicit form.

Input Parameters

<i>icompg</i>	INTEGER. Specifies whether singular vectors are to be computed in compact form, as follows: = 0: Compute singular values only. = 1: Compute singular vectors of upper bidiagonal matrix in compact form.
<i>smlsiz</i>	INTEGER. The maximum size of the subproblems at the bottom of the computation tree.
<i>n</i>	INTEGER. The row dimension of the upper bidiagonal matrix. This is also the dimension of the main diagonal array d .
<i>sqre</i>	INTEGER. Specifies the column dimension of the bidiagonal matrix. If $sqre = 0$: the bidiagonal matrix has column dimension $m = n$

If $s_{qre} = 1$: the bidiagonal matrix has column dimension $m = n + 1$.

d REAL for slasda
DOUBLE PRECISION for dlasda.
Array, DIMENSION (n). On entry, *d* contains the main diagonal of the bidiagonal matrix.

e REAL for slasda
DOUBLE PRECISION for dlasda.
Array, DIMENSION ($m - 1$). Contains the subdiagonal entries of the bidiagonal matrix. On exit, *e* is destroyed.

ldu INTEGER. The leading dimension of arrays *u*, *vt*, *difl*, *difr*, *poles*, *givnum*, and *z*. $ldu \geq n$.

ldgcol INTEGER. The leading dimension of arrays *givcol* and *perm*. $ldgcol \geq n$.

work REAL for slasda
DOUBLE PRECISION for dlasda.
Workspace array, DIMENSION ($6n + (smlsiz+1)^2$).

iwork INTEGER.
Workspace array, *Dimension* must be at least $(7n)$.

Output Parameters

d On exit *d*, if *info* = 0, contains the singular values of the bidiagonal matrix.

u REAL for slasda
DOUBLE PRECISION for dlasda.
Array, DIMENSION (*ldu*, *smlsiz*) if *icompq* = 1.
Not referenced if *icompq* = 0.
If *icompq* = 1, on exit, *u* contains the left singular vector matrices of all subproblems at the bottom level.

vt REAL for slasda
DOUBLE PRECISION for dlasda.
Array, DIMENSION (*ldu*, *smlsiz+1*) if *icompq* = 1, and not referenced if *icompq* = 0. If *icompq* = 1, on exit, *vt* contains the right singular vector matrices of all subproblems at the bottom level.

<i>k</i>	INTEGER. Array, DIMENSION (<i>n</i>) if <i>icompg</i> = 1 and DIMENSION (1) if <i>icompg</i> = 0. If <i>icompg</i> = 1, on exit, <i>k</i>(<i>i</i>) is the dimension of the <i>i</i>-th secular equation on the computation tree.
<i>difl</i>	REAL for slasda DOUBLE PRECISION for dlasda. Array, DIMENSION (<i>ldu</i>, <i>nlvl</i>), where <i>nlvl</i> = floor(log₂(<i>n</i>/<i>smlsiz</i>)).
<i>difr</i>	REAL for slasda DOUBLE PRECISION for dlasda. Array, DIMENSION (<i>ldu</i>, 2 <i>nlvl</i>) if <i>icompg</i> = 1 and DIMENSION (<i>n</i>) if <i>icompg</i> = 0. If <i>icompg</i> = 1, on exit, <i>difl</i>(1:<i>n</i>, <i>i</i>) and <i>difr</i>(1:<i>n</i>, 2<i>i</i> - 1) record distances between singular values on the <i>i</i>-th level and singular values on the (<i>i</i> - 1)-th level, and <i>difr</i>(1:<i>n</i>, 2<i>i</i>) contains the normalizing factors for the right singular vector matrix. See ?lasd8 for details.
<i>z</i>	REAL for slasda DOUBLE PRECISION for dlasda. Array, DIMENSION (<i>ldu</i>, <i>nlvl</i>) if <i>icompg</i> = 1 and DIMENSION (<i>n</i>) if <i>icompg</i> = 0. The first <i>k</i> elements of <i>z</i>(1, <i>i</i>) contain the components of the deflation-adjusted updating row vector for subproblems on the <i>i</i>-th level.
<i>poles</i>	REAL for slasda DOUBLE PRECISION for dlasda Array, DIMENSION (<i>ldu</i>, 2*<i>nlvl</i>) if <i>icompg</i> = 1, and not referenced if <i>icompg</i> = 0. If <i>icompg</i> = 1, on exit, <i>poles</i>(1, 2<i>i</i> - 1) and <i>poles</i>(1, 2<i>i</i>) contain the new and old singular values involved in the secular equations on the <i>i</i>-th level.
<i>givptr</i>	INTEGER. Array, DIMENSION (<i>n</i>) if <i>icompg</i> = 1, and not referenced if <i>icompg</i> = 0. If <i>icompg</i> = 1, on exit, <i>givptr</i>(<i>i</i>) records the number of Givens rotations performed on the <i>i</i>-th problem on the computation tree.

<i>givcol</i>	INTEGER . Array, DIMENSION (<i>ldgcol</i>, 2*<i>nlvl</i>) if <i>icompg</i> = 1, and not referenced if <i>icompg</i> = 0. If <i>icompg</i> = 1, on exit, for each <i>i</i>, <i>givcol</i>(1, 2 <i>i</i> - 1) and <i>givcol</i>(1, 2 <i>i</i>) record the locations of Givens rotations performed on the <i>i</i>-th level on the computation tree.
<i>perm</i>	INTEGER . Array, DIMENSION (<i>ldgcol</i>, <i>nlvl</i>) if <i>icompg</i> = 1, and not referenced if <i>icompg</i> = 0. If <i>icompg</i> = 1, on exit, <i>perm</i> (1, <i>i</i>) records permutations done on the <i>i</i>-th level of the computation tree.
<i>givnum</i>	REAL for <i>slasda</i> DOUBLE PRECISION for <i>dlasda</i>. Array DIMENSION (<i>ldu</i>, 2*<i>nlvl</i>) if <i>icompg</i> = 1, and not referenced if <i>icompg</i> = 0. If <i>icompg</i> = 1, on exit, for each <i>i</i>, <i>givnum</i>(1, 2 <i>i</i> - 1) and <i>givnum</i>(1, 2 <i>i</i>) record the <i>c</i>- and <i>s</i>-values of Givens rotations performed on the <i>i</i>-th level on the computation tree.
<i>c</i>	REAL for <i>slasda</i> DOUBLE PRECISION for <i>dlasda</i>. Array, DIMENSION (<i>n</i>) if <i>icompg</i> = 1, and DIMENSION (1) if <i>icompg</i> = 0. If <i>icompg</i> = 1 and the <i>i</i>-th subproblem is not square, on exit, <i>c</i>(<i>i</i>) contains the <i>c</i>-value of a Givens rotation related to the right null space of the <i>i</i>-th subproblem.
<i>s</i>	REAL for <i>slasda</i> DOUBLE PRECISION for <i>dlasda</i>. Array, DIMENSION (<i>n</i>) if <i>icompg</i> = 1, and DIMENSION (1) if <i>icompg</i> = 0. If <i>icompg</i> = 1 and the <i>i</i>-th subproblem is not square, on exit, <i>s</i>(<i>i</i>) contains the <i>s</i>-value of a Givens rotation related to the right null space of the <i>i</i>-th subproblem.
<i>info</i>	INTEGER. = 0: successful exit. < 0: if <i>info</i> = -<i>i</i>, the <i>i</i>-th argument had an illegal value > 0: If <i>info</i> = 1, an singular value did not converge

?lasdq

Computes the SVD of a real bidiagonal matrix with diagonal d and off-diagonal e . Used by ?bdsdc.

Syntax

```
call slasdq( uplo, sqre, n, ncvt, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc,
            work, info )
```

```
call dlasdq( uplo, sqre, n, ncvt, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc,
            work, info )
```

Description

The routine ?lasdq computes the singular value decomposition (SVD) of a real (upper or lower) bidiagonal matrix with diagonal d and off-diagonal e , accumulating the transformations if desired. If B is the input bidiagonal matrix, the algorithm computes orthogonal matrices Q and P such that $B = Q^T S P^T$. The singular values s are overwritten on d .

The input matrix U is changed to $U^* Q$ if desired.

The input matrix VT is changed to $P^T VT$ if desired.

The input matrix C is changed to $Q^T C$ if desired.

Input Parameters

<i>uplo</i>	CHARACTER*1. On entry, <i>uplo</i> specifies whether the input bidiagonal matrix is upper or lower bidiagonal. If <i>uplo</i> = 'U' or 'u', B is upper bidiagonal; If <i>uplo</i> = 'L' or 'l', B is lower bidiagonal.
<i>sqre</i>	INTEGER. = 0: then the input matrix is n -by- n . = 1: then the input matrix is n -by- $(n+1)$ if <i>uplu</i> = 'U' and $(n+1)$ -by- n if <i>uplu</i> = 'L'. The bidiagonal matrix has $n = n_l + n_r + 1$ rows and $m = n + sqre \geq n$ columns.
<i>n</i>	INTEGER. On entry, <i>n</i> specifies the number of rows and columns in the matrix. <i>n</i> must be at least 0.

<i>ncvt</i>	INTEGER. On entry, <i>ncvt</i> specifies the number of columns of the matrix <i>VT</i> . <i>ncvt</i> must be at least 0.
<i>nru</i>	INTEGER. On entry, <i>nru</i> specifies the number of rows of the matrix <i>U</i> . <i>nru</i> must be at least 0.
<i>ncc</i>	INTEGER. On entry, <i>ncc</i> specifies the number of columns of the matrix <i>C</i> . <i>ncc</i> must be at least 0.
<i>d</i>	REAL for <i>slasdq</i> DOUBLE PRECISION for <i>dlasdq</i> . Array, DIMENSION (<i>n</i>). On entry, <i>d</i> contains the diagonal entries of the bidiagonal matrix.
<i>e</i>	REAL for <i>slasdq</i> DOUBLE PRECISION for <i>dlasdq</i> . Array, DIMENSION is (<i>n</i> -1) if <i>sqre</i> = 0 and <i>n</i> if <i>sqre</i> = 1. On entry, the entries of <i>e</i> contain the off-diagonal entries of the bidiagonal matrix.
<i>vt</i>	REAL for <i>slasdq</i> DOUBLE PRECISION for <i>dlasdq</i> . Array, DIMENSION (<i>ldvt</i> , <i>ncvt</i>). On entry, contains a matrix which on exit has been premultiplied by P^T , dimension <i>n</i> -by- <i>ncvt</i> if <i>sqre</i> = 0 and (<i>n</i> +1)-by- <i>ncvt</i> if <i>sqre</i> = 1 (not referenced if <i>ncvt</i> =0).
<i>ldvt</i>	INTEGER. On entry, <i>ldvt</i> specifies the leading dimension of <i>vt</i> as declared in the calling (sub) program. <i>ldvt</i> must be at least 1. If <i>ncvt</i> is nonzero, <i>ldvt</i> must also be at least <i>n</i> .
<i>u</i>	REAL for <i>slasdq</i> DOUBLE PRECISION for <i>dlasdq</i> . Array, DIMENSION (<i>ldu</i> , <i>n</i>). On entry, contains a matrix which on exit has been postmultiplied by <i>Q</i> , dimension <i>nru</i> -by- <i>n</i> if <i>sqre</i> = 0 and <i>nru</i> -by-(<i>n</i> +1) if <i>sqre</i> = 1 (not referenced if <i>nru</i> =0).
<i>ldu</i>	INTEGER. On entry, <i>ldu</i> specifies the leading dimension of <i>u</i> as declared in the calling (sub) program. <i>ldu</i> must be at least $\max(1, \text{ } nru \text{ })$.
<i>c</i>	REAL for <i>slasdq</i> DOUBLE PRECISION for <i>dlasdq</i> .

	Array, DIMENSION (<i>ldc</i> , <i>ncc</i>). On entry, contains an <i>n</i> -by- <i>ncc</i> matrix which on exit has been premultiplied by Q' , dimension <i>n</i> -by- <i>ncc</i> if <i>scre</i> = 0 and (<i>n</i> +1)-by- <i>ncc</i> if <i>scre</i> = 1 (not referenced if <i>ncc</i> =0).
<i>ldc</i>	INTEGER. On entry, <i>ldc</i> specifies the leading dimension of <i>c</i> as declared in the calling (sub) program. <i>ldc</i> must be at least 1. If <i>ncc</i> is non-zero, <i>ldc</i> must also be at least <i>n</i> .
<i>work</i>	REAL for slasdq DOUBLE PRECISION for dlasdq. Array, DIMENSION (4 <i>n</i>). This is a workspace array. Only referenced if one of <i>ncvt</i> , <i>nru</i> , or <i>ncc</i> is nonzero, and if <i>n</i> is at least 2.

Output Parameters

<i>d</i>	On normal exit, <i>d</i> contains the singular values in ascending order.
<i>e</i>	On normal exit, <i>e</i> will contain 0. If the algorithm does not converge, <i>d</i> and <i>e</i> will contain the diagonal and superdiagonal entries of a bidiagonal matrix orthogonally equivalent to the one given as input.
<i>vt</i>	On exit, the matrix has been premultiplied by P' .
<i>u</i>	On exit, the matrix has been postmultiplied by Q .
<i>c</i>	On exit, the matrix has been premultiplied by Q' .
<i>info</i>	INTEGER. On exit, a value of 0 indicates a successful exit. If <i>info</i> < 0, argument number - <i>info</i> is illegal. If <i>info</i> > 0, the algorithm did not converge, and <i>info</i> specifies how many superdiagonals did not converge.

?lasdt

Creates a tree of subproblems for bidiagonal divide and conquer. Used by ?bdsdc.

Syntax

```
call slasdt( n, lvl, nd, inode, ndiml, ndimr, msub )
```

```
call dlasdt( n, lvl, nd, inode, ndiml, ndimr, msub )
```

Description

The routine creates a tree of subproblems for bidiagonal divide and conquer.

Input Parameters

<i>n</i>	INTEGER. On entry, the number of diagonal elements of the bidiagonal matrix.
<i>msub</i>	INTEGER. On entry, the maximum row dimension each subproblem at the bottom of the tree can be of.

Output Parameters

<i>lvl</i>	INTEGER. On exit, the number of levels on the computation tree.
<i>nd</i>	INTEGER. On exit, the number of nodes on the tree.
<i>inode</i>	INTEGER. Array, DIMENSION (<i>n</i>). On exit, centers of subproblems.
<i>ndiml</i>	INTEGER . Array, DIMENSION (<i>n</i>). On exit, row dimensions of left children.
<i>ndimr</i>	INTEGER . Array, DIMENSION (<i>n</i>). On exit, row dimensions of right children.

zlaset

Initializes the off-diagonal elements and the diagonal elements of a matrix to given values.

Syntax

```
call slaset( uplo, m, n, alpha, beta, a, lda )
call dlaset( uplo, m, n, alpha, beta, a, lda )
call claset( uplo, m, n, alpha, beta, a, lda )
call zlaset( uplo, m, n, alpha, beta, a, lda )
```

Description

The routine initializes an m -by- n matrix A to β on the diagonal and α on the off-diagonals.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies the part of the matrix A to be set. If <i>uplo</i> = 'U', upper triangular part is set; the strictly lower triangular part of A is not changed. If <i>uplo</i> = 'L': lower triangular part is set; the strictly upper triangular part of A is not changed. Otherwise: All of the matrix A is set.
<i>m</i>	INTEGER. The number of rows of the matrix A . $m \geq 0$.
<i>n</i>	INTEGER. The number of columns of the matrix A . $n \geq 0$.
<i>alpha, beta</i>	REAL for slaset DOUBLE PRECISION for dlaset COMPLEX for claset COMPLEX*16 for zlaset. The constants to which the off-diagonal and diagonal elements are to be set, respectively.
<i>a</i>	REAL for slaset DOUBLE PRECISION for dlaset COMPLEX for claset COMPLEX*16 for zlaset.

Array, DIMENSION (lda, n).
 On entry, the m -by- n matrix A .
 lda INTEGER. The leading dimension of the array a .
 $lda \geq \max(1, m)$.

Output Parameters

a On exit, the leading m -by- n submatrix of A is set as follows:
 if $uplo = 'U'$, $A(i, j) = \alpha, 1 \leq i \leq j-1, 1 \leq j \leq n$,
 if $uplo = 'L'$, $A(i, j) = \alpha, j+1 \leq i \leq m, 1 \leq j \leq n$,
 otherwise, $A(i, j) = \alpha, 1 \leq i \leq m, 1 \leq j \leq n, i \neq j$,
 and, for all $uplo$, $A(i, i) = \beta, 1 \leq i \leq \min(m, n)$.

?lasq1

*Computes the singular values of a real square
 bidiagonal matrix. Used by ?bdsqr.*

Syntax

```
call slasq1( n, d, e, work, info )
call dlasq1( n, d, e, work, info )
```

Description

The routine ?lasq1 computes the singular values of a real n -by- n bidiagonal matrix with diagonal d and off-diagonal e . The singular values are computed to high relative accuracy, in the absence of denormalization, underflow and overflow.

Input Parameters

n INTEGER. The number of rows and columns in the matrix. $n \geq 0$.
 d REAL for slasq1
 DOUBLE PRECISION for dlasq1.
 Array, DIMENSION (n).
 On entry, d contains the diagonal elements of the bidiagonal matrix whose SVD is desired.

<i>e</i>	REAL for <code>slasq1</code> DOUBLE PRECISION for <code>dlasq1</code>. Array, DIMENSION (<i>n</i>). On entry, elements <code>e(1:n-1)</code> contain the off-diagonal elements of the bidiagonal matrix whose <i>SVD</i> is desired.
<i>work</i>	REAL for <code>slasq1</code> DOUBLE PRECISION for <code>dlasq1</code>. Workspace array, DIMENSION ($4n$).

Output Parameters

<i>d</i>	On normal exit, <i>d</i> contains the singular values in decreasing order.
<i>e</i>	On exit, <i>e</i> is overwritten.
<i>info</i>	INTEGER. = 0: successful exit; < 0: if <i>info</i> = -<i>i</i>, the <i>i</i>-th argument had an illegal value; > 0: the algorithm failed: = 1, a split was marked by a positive value in <i>e</i>; = 2, current block of <i>z</i> not diagonalized after $30n$ iterations (in inner while loop); = 3, termination criterion of outer while loop not met (program created more than <i>n</i> unreduced blocks).

?lasq2

*Computes all the eigenvalues of the symmetric positive definite tridiagonal matrix associated with the `qd` array *z* to high relative accuracy. Used by `?bdsqr` and `?stegr`.*

Syntax

```
call slasq2( n, z, info )
call dlasq2( n, z, info )
```

Description

The routine `?lasq2` computes all the eigenvalues of the symmetric positive definite tridiagonal matrix associated with the *qd* array *z* to high relative accuracy, in the absence of denormalization, underflow and overflow.

To see the relation of *z* to the tridiagonal matrix, let *L* be a unit lower bidiagonal matrix with subdiagonals *z*(2,4,6,...) and let *U* be an upper bidiagonal matrix with 1's above and diagonal *z*(1,3,5,...). The tridiagonal is *LU* or, if you prefer, the symmetric tridiagonal to which it is similar.

Input Parameters

n INTEGER. The number of rows and columns in the matrix.
 $n \geq 0$.

z REAL for `slasq2`
 DOUBLE PRECISION for `dlasq2`.
 Array, DIMENSION (4 * *n*).
 On entry, *z* holds the *qd* array.

Output Parameters

z On exit, entries 1 to *n* hold the eigenvalues in decreasing order, *z*(2*n*+1) holds the trace, and *z*(2*n*+2) holds the sum of the eigenvalues. If $n > 2$, then *z*(2*n*+3) holds the iteration count, *z*(2*n*+4) holds ndivs/n^2 , and *z*(2*n*+5) holds the percentage of shifts that failed.

info INTEGER.
 = 0: successful exit;
 < 0: if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*, if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+ *j*);
 > 0: the algorithm failed:
 = 1, a split was marked by a positive value in *e*;
 = 2, current block of *z* not diagonalized after 30**n* iterations (in inner while loop);
 = 3, termination criterion of outer while loop not met (program created more than *n* unreduced blocks).

Application Notes

The routine `?lasq2` defines a logical variable, `ieee`, which is `.TRUE.` on machines which follow ieee-754 floating-point standard in their handling of infinities and NaNs, and `.FALSE.` otherwise. This variable is passed to `?lazq3`.

?lasq3

Checks for deflation, computes a shift and calls `dqds`. Used by `?bdsqr`.

Syntax

```
call slasq3( i0, n0, z, pp, dmin, sigma, desig, qmax, nfail, iter, ndiv, ieee
)
call dlasq3( i0, n0, z, pp, dmin, sigma, desig, qmax, nfail, iter, ndiv, ieee
)
```

Description

The routine `?lasq3` checks for deflation, computes a shift, and calls `dqds`. In case of failure, it changes shifts, and tries again until output is positive.

Input Parameters

<code>i0</code>	INTEGER. First index.
<code>n0</code>	INTEGER. Last index.
<code>z</code>	REAL for <code>slasq3</code> DOUBLE PRECISION for <code>dlasq3</code> . Array, DIMENSION (4 <i>n</i>). <i>z</i> holds the <i>qd</i> array.
<code>pp</code>	INTEGER. <i>pp</i> =0 for ping, <i>pp</i> =1 for pong.
<code>desig</code>	REAL for <code>slasq3</code> DOUBLE PRECISION for <code>dlasq3</code> . Lower order part of <i>sigma</i> .
<code>qmax</code>	REAL for <code>slasq3</code> DOUBLE PRECISION for <code>dlasq3</code> . Maximum value of <i>q</i> .
<code>ieee</code>	LOGICAL. Flag for ieee or non-ieee arithmetic (passed to <code>?lasq5</code>).

Output Parameters

<i>dmin</i>	REAL for <code>slasq3</code> DOUBLE PRECISION for <code>dlasq3</code> . Minimum value of <i>d</i> .
<i>sigma</i>	REAL for <code>slasq3</code> DOUBLE PRECISION for <code>dlasq3</code> . Sum of shifts used in current segment.
<i>desig</i>	Lower order part of <i>sigma</i> .
<i>nfail</i>	INTEGER. Number of times shift was too big.
<i>iter</i>	INTEGER. Number of iterations.
<i>ndiv</i>	INTEGER. Number of divisions.

?lasq4

*Computes an approximation to the smallest eigenvalue using values of *d* from the previous transform. Used by ?bdsqr.*

Syntax

```
call slasq4( i0, n0, z, pp, n0in, dmin, dmin1, dmin2, dn, dn1, dn2, tau, ttype
)
call dlasq4( i0, n0, z, pp, n0in, dmin, dmin1, dmin2, dn, dn1, dn2, tau, ttype
)
```

Description

The routine computes an approximation *tau* to the smallest eigenvalue using values of *d* from the previous transform.

Input Parameters

<i>i0</i>	INTEGER. First index.
<i>n0</i>	INTEGER. Last index.
<i>z</i>	REAL for <code>slasq4</code> DOUBLE PRECISION for <code>dlasq4</code> . Array, DIMENSION (4 <i>n</i>). <i>z</i> holds the <i>qd</i> array.

<i>pp</i>	INTEGER. <i>pp</i> =0 for ping, <i>pp</i> =1 for pong.
<i>n0in</i>	INTEGER. The value of <i>n0</i> at start of eigtest.
<i>dmin</i>	REAL for slasq4 DOUBLE PRECISION for dlasq4. Minimum value of <i>d</i> .
<i>dmin1</i>	REAL for slasq4 DOUBLE PRECISION for dlasq4. Minimum value of <i>d</i> , excluding <i>d</i> (<i>n0</i>).
<i>dmin2</i>	REAL for slasq4 DOUBLE PRECISION for dlasq4. Minimum value of <i>d</i> , excluding <i>d</i> (<i>n0</i>) and <i>d</i> (<i>n0</i> -1).
<i>dn</i>	REAL for slasq4 DOUBLE PRECISION for dlasq4. Contains <i>d</i> (<i>n</i>).
<i>dn1</i>	REAL for slasq4 DOUBLE PRECISION for dlasq4. Contains <i>d</i> (<i>n</i> -1).
<i>dn2</i>	REAL for slasq4 DOUBLE PRECISION for dlasq4. Contains <i>d</i> (<i>n</i> -2).

Output Parameters

<i>tau</i>	REAL for slasq4 DOUBLE PRECISION for dlasq4. Shift.
<i>ttype</i>	INTEGER. Shift type.

?lasq5

Computes one dqds transform in ping-pong form.
Used by ?bdsqr and ?stegr.

Syntax

```
call slasq5( i0, n0, z, pp, tau, dmin, dmin1, dmin2, dn, dnm1, dnm2, ieee )
call dlasq5( i0, n0, z, pp, tau, dmin, dmin1, dmin2, dn, dnm1, dnm2, ieee )
```

Description

The routine computes one dqds transform in ping-pong form: one version for ieee machines, another for non-ieee machines.

Input Parameters

<i>i0</i>	INTEGER. First index.
<i>n0</i>	INTEGER. Last index.
<i>z</i>	REAL for slasq5 DOUBLE PRECISION for dlasq5. Array, DIMENSION (4 <i>n</i>). <i>z</i> holds the qd array. <i>emin</i> is stored in <i>z</i> (4* <i>n0</i>) to avoid an extra argument.
<i>pp</i>	INTEGER. <i>pp</i> =0 for ping, <i>pp</i> =1 for pong.
<i>tau</i>	REAL for slasq5 DOUBLE PRECISION for dlasq5. This is the shift.
<i>ieee</i>	LOGICAL. Flag for IEEE or non-IEEE arithmetic.

Output Parameters

<i>dmin</i>	REAL for slasq5 DOUBLE PRECISION for dlasq5. Minimum value of <i>d</i> .
<i>dmin1</i>	REAL for slasq5 DOUBLE PRECISION for dlasq5. Minimum value of <i>d</i> , excluding <i>d</i> (<i>n0</i>).
<i>dmin2</i>	REAL for slasq5 DOUBLE PRECISION for dlasq5. Minimum value of <i>d</i> , excluding <i>d</i> (<i>n0</i>) and <i>d</i> (<i>n0</i> -1).
<i>dn</i>	REAL for slasq5 DOUBLE PRECISION for dlasq5. Contains <i>d</i> (<i>n0</i>), the last value of <i>d</i> .
<i>dnm1</i>	REAL for slasq5 DOUBLE PRECISION for dlasq5. Contains <i>d</i> (<i>n0</i> -1).
<i>dnm2</i>	REAL for slasq5

DOUBLE PRECISION for `dlasq5`. Contains $d(n0-2)$.

?lasq6

*Computes one dqd transform in ping-pong form.
Used by ?bdsqr and ?stegr.*

Syntax

```
call slasq6( i0, n0, z, pp, dmin, dmin1, dmin2, dn, dnm1, dnm2 )
call dlasq6( i0, n0, z, pp, dmin, dmin1, dmin2, dn, dnm1, dnm2 )
```

Description

The routine ?lasq6 computes one dqd (shift equal to zero) transform in ping-pong form, with protection against underflow and overflow.

Input Parameters

<i>i0</i>	INTEGER. First index.
<i>n0</i>	INTEGER. Last index.
<i>z</i>	REAL for <code>slasq6</code> DOUBLE PRECISION for <code>dlasq6</code> . Array, DIMENSION (4 <i>n</i>). <i>z</i> holds the qd array. <i>emin</i> is stored in $z(4*n0)$ to avoid an extra argument.
<i>pp</i>	INTEGER. <i>pp</i> =0 for ping, <i>pp</i> =1 for pong.

Output Parameters

<i>dmin</i>	REAL for <code>slasq6</code> DOUBLE PRECISION for <code>dlasq6</code> . Minimum value of d .
<i>dmin1</i>	REAL for <code>slasq6</code> DOUBLE PRECISION for <code>dlasq6</code> . Minimum value of d , excluding $d(n0)$.
<i>dmin2</i>	REAL for <code>slasq6</code> DOUBLE PRECISION for <code>dlasq6</code> . Minimum value of d , excluding $d(n0)$ and $d(n0-1)$.
<i>dn</i>	REAL for <code>slasq6</code>

	DOUBLE PRECISION for dlasq6. Contains $d(n0)$, the last value of d .
$dnm1$	REAL for slasq6 DOUBLE PRECISION for dlasq6. Contains $d(n0-1)$.
$dnm2$	REAL for slasq6 DOUBLE PRECISION for dlasq6. Contains $d(n0-2)$.

?lasr

Applies a sequence of plane rotations to a general rectangular matrix.

Syntax

```
call slasr( side, pivot, direct, m, n, c, s, a, lda )
call dlasr( side, pivot, direct, m, n, c, s, a, lda )
call clasr( side, pivot, direct, m, n, c, s, a, lda )
call zlasr( side, pivot, direct, m, n, c, s, a, lda )
```

Description

The routine applies a sequence of plane rotations to a real/complex matrix A , from the left or the right.

$A := P^*A$, when $side = 'L'$ (Left-hand side)

$A := A^*P$, when $side = 'R'$ (Right-hand side)

where P is an orthogonal matrix consisting of a sequence of plane rotations with $z = m$ when $side = 'L'$ and $z = n$ when $side = 'R'$.

When $direct = 'F'$ (Forward sequence), then

$$P = P(z-1) * \dots * P(2) * P(1),$$

and when $direct = 'B'$ (Backward sequence), then

$$P = P(1) * P(2) * \dots * P(z-1),$$

where $P(k)$ is a plane rotation matrix defined by the 2-by-2 plane rotation:

$$R(k) = \begin{bmatrix} c(k) & s(k) \\ -s(k) & c(k) \end{bmatrix}$$

When $pivot = 'V'$ (Variable pivot), the rotation is performed for the plane $(k, k + 1)$, that is, $P(k)$ has the form

$$P(k) = \begin{bmatrix} 1 & & & & & \\ & \dots & & & & \\ & & 1 & & & \\ & & & c(k) & s(k) & \\ & & & -s(k) & c(k) & \\ & & & & & 1 \\ & & & & & \dots \\ & & & & & & 1 \end{bmatrix}$$

where $R(k)$ appears as a rank-2 modification to the identity matrix in rows and columns k and $k+1$.

When $pivot = 'T'$ (Top pivot), the rotation is performed for the plane $(1, k+1)$, so $P(k)$ has the form

$$P(k) = \begin{bmatrix} c(k) & & & s(k) & & \\ & 1 & & & & \\ & & \dots & & & \\ & & & 1 & & \\ -s(k) & & & & c(k) & \\ & & & & & 1 \\ & & & & & \dots \\ & & & & & & 1 \end{bmatrix}$$

where $R(k)$ appears in rows and columns k and $k+1$.

Similarly, when $pivot = 'B'$ (Bottom pivot), the rotation is performed for the plane (k, z) , giving $P(k)$ the form

$$P(k) = \begin{bmatrix} 1 & & & & & \\ & \dots & & & & \\ & & 1 & & & \\ & & & c(k) & & s(k) \\ & & & & 1 & \\ & & & & & \dots \\ & & & & & & 1 \\ & & & -s(k) & & & c(k) \end{bmatrix}$$

where $R(k)$ appears in rows and columns k and z . The rotations are performed without ever forming $P(k)$ explicitly.

Input Parameters

side CHARACTER*1. Specifies whether the plane rotation matrix P is applied to A on the left or the right.

	= 'L': left, compute $A := P^*A$ = 'R': right, compute $A := A^*P$
<i>direct</i>	CHARACTER*1. Specifies whether P is a forward or backward sequence of plane rotations. = 'F': forward, $P = P(z-1) * \dots * P(2) * P(1)$ = 'B': backward, $P = P(1) * P(2) * \dots * P(z-1)$
<i>pivot</i>	CHARACTER*1. Specifies the plane for which $P(k)$ is a plane rotation matrix. = 'V': Variable pivot, the plane $(k, k+1)$ = 'T': Top pivot, the plane $(1, k+1)$ = 'B': Bottom pivot, the plane (k, z)
<i>m</i>	INTEGER. The number of rows of the matrix A. If $m \leq 1$, an immediate return is effected.
<i>n</i>	INTEGER. The number of columns of the matrix A. If $n \leq 1$, an immediate return is effected.
<i>c, s</i>	REAL for slasr/clasr DOUBLE PRECISION for dlasr/zlasr. Arrays, DIMENSION $(m-1)$ if <i>side</i> = 'L', $(n-1)$ if <i>side</i> = 'R' . $c(k)$ and $s(k)$ contain the cosine and sine of the plane rotations respectively that define the 2-by-2 plane rotation part ($R(k)$) of the $P(k)$ matrix as described above in <i>Description</i>.
<i>a</i>	REAL for slasr DOUBLE PRECISION for dlasr COMPLEX for clasr COMPLEX*16 for zlasr. Array, DIMENSION (lda, n). The m-by-n matrix A.
<i>lda</i>	INTEGER. The leading dimension of the array a. $lda \geq \max(1, m)$.

Output Parameters

a On exit, *A* is overwritten by P^*A if *side* = 'R', or by A^*P if *side* = 'L'.

?lasrt

Sorts numbers in increasing or decreasing order.

Syntax

```
call slasrt( id, n, d, info )
```

```
call dlasrt( id, n, d, info )
```

Description

The routine ?lasrt sorts the numbers in *d* in increasing order (if *id* = 'I') or in decreasing order (if *id* = 'D'). It uses Quick Sort, reverting to Insertion Sort on arrays of size ≤ 20 . Dimension of *stack* limits *n* to about 2^{32} .

Input Parameters

id CHARACTER*1.
= 'I': sort *d* in increasing order;
= 'D': sort *d* in decreasing order.

n INTEGER. The length of the array *d*.

d REAL for slasrt
DOUBLE PRECISION for dlasrt.
On entry, the array to be sorted.

Output Parameters

d On exit, *d* has been sorted into increasing order
($d(1) \leq \dots \leq d(n)$) or into decreasing order
($d(1) \geq \dots \geq d(n)$), depending on *id*.

info INTEGER.
= 0: successful exit
< 0: if *info* = -*i*, the *i*-th argument had an illegal value.

?lassq

Updates a sum of squares represented in scaled form.

Syntax

```
call slassq( n, x, incx, scale, sumsq )
call dlassq( n, x, incx, scale, sumsq )
call classq( n, x, incx, scale, sumsq )
call zlassq( n, x, incx, scale, sumsq )
```

Description

The real routines `slassq`/`dlassq` return the values `scl` and `smSQ` such that

$$scl^2 * smSQ = x(1)^2 + \dots + x(n)^2 + scale^2 * sumsq,$$

where $x(i) = x(1 + (i - 1) * incx)$.

The value of `sumsq` is assumed to be non-negative and `scl` returns the value

$$scl = \max(scale, \text{abs}(x(i))).$$

Values `scale` and `sumsq` must be supplied in `scale` and `sumsq`, and `scl` and `smSQ` are overwritten on `scale` and `sumsq`, respectively.

The complex routines `classq`/`zlassq` return the values `scl` and `ssq` such that

$$scl^2 * ssq = x(1)^2 + \dots + x(n)^2 + scale^2 * sumsq,$$

where $x(i) = \text{abs}(x(1 + (i - 1) * incx))$.

The value of `sumsq` is assumed to be at least unity and the value of `ssq` will then satisfy 1.0

$$\leq ssq \leq sumsq + 2n$$

`scale` is assumed to be non-negative and `scl` returns the value

$$scl = \max(scale, \text{abs}(\text{real}(x(i))), \text{abs}(\text{aimag}(x(i)))).$$

Values `scale` and `sumsq` must be supplied in `scale` and `sumsq`, and `scl` and `ssq` are overwritten on `scale` and `sumsq`, respectively.

All routines `?lassq` make only one pass through the vector `x`.

Input Parameters

<i>n</i>	INTEGER. The number of elements to be used from the vector <i>x</i> .
<i>x</i>	REAL for <i>slassq</i> DOUBLE PRECISION for <i>dlassq</i> COMPLEX for <i>classq</i> COMPLEX*16 for <i>zlassq</i> . The vector for which a scaled sum of squares is computed: $x(i) = x(1+(i-1)*incx), 1 \leq i \leq n$.
<i>incx</i>	INTEGER. The increment between successive values of the vector <i>x</i> . <i>incx</i> > 0.
<i>scale</i>	REAL for <i>slassq/classq</i> DOUBLE PRECISION for <i>dlassq/zlassq</i> . On entry, the value <i>scale</i> in the equation above.
<i>sumsq</i>	REAL for <i>slassq/classq</i> DOUBLE PRECISION for <i>dlassq/zlassq</i> . On entry, the value <i>sumsq</i> in the equation above.

Output Parameters

<i>scale</i>	On exit, <i>scale</i> is overwritten with <i>scl</i> , the scaling factor for the sum of squares.
<i>sumsq</i>	For real flavors: On exit, <i>sumsq</i> is overwritten with the value <i>sumsq</i> in the equation above. For complex flavors: On exit, <i>sumsq</i> is overwritten with the value <i>ssq</i> in the equation above.

?lasv2

Computes the singular value decomposition of a 2-by-2 triangular matrix.

Syntax

```
call slasv2( f, g, h, ssmin, ssmax, snr, csr, snl, csl )
call dlasv2( f, g, h, ssmin, ssmax, snr, csr, snl, csl )
```

Description

The routine ?lasv2 computes the singular value decomposition of a 2-by-2 triangular matrix

$$\begin{bmatrix} f & g \\ 0 & h \end{bmatrix}$$

On return, $\text{abs}(ssmax)$ is the larger singular value, $\text{abs}(ssmin)$ is the smaller singular value, and (csl, snl) and (csr, snr) are the left and right singular vectors for $\text{abs}(ssmax)$, giving the decomposition

$$\begin{bmatrix} csl & snl \\ -snl & csl \end{bmatrix} \begin{bmatrix} f & g \\ 0 & h \end{bmatrix} \begin{bmatrix} csr & -snr \\ snr & csr \end{bmatrix} = \begin{bmatrix} ssmax & 0 \\ 0 & ssmin \end{bmatrix}$$

Input Parameters

f, g, h	REAL for slasv2 DOUBLE PRECISION for dlasv2. The (1,1), (1,2) and (2,2) elements of the 2-by-2 matrix, respectively.

Output Parameters

$ssmin, ssmax$	REAL for slasv2 DOUBLE PRECISION for dlasv2.

`abs(ssmin)` and `abs(ssmax)` is the smaller and the larger singular value, respectively.

snl, csl REAL for `slasv2`
 DOUBLE PRECISION for `dlasv2`.
 The vector (*csl, snl*) is a unit left singular vector for the singular value `abs(ssmax)`.

snr, csr REAL for `slasv2`
 DOUBLE PRECISION for `dlasv2`.
 The vector (*csr, snr*) is a unit right singular vector for the singular value `abs(ssmax)`.

Application Notes

Any input parameter may be aliased with any output parameter.

Barring over/underflow and assuming a guard digit in subtraction, all output quantities are correct to within a few units in the last place (ulps).

In ieee arithmetic, the code works correctly if one matrix element is infinite. Overflow will not occur unless the largest singular value itself overflows or is within a few ulps of overflow. (On machines with partial overflow, like the Cray, overflow may occur if the largest singular value is within a factor of 2 of overflow.) Underflow is harmless if underflow is gradual. Otherwise, results may correspond to a matrix modified by perturbations of size near the underflow threshold.

?laswp

Performs a series of row interchanges on a general rectangular matrix.

Syntax

```
call slaswp( n, a, lda, k1, k2, ipiv, incx )
call dlaswp( n, a, lda, k1, k2, ipiv, incx )
call claswp( n, a, lda, k1, k2, ipiv, incx )
call zlaswp( n, a, lda, k1, k2, ipiv, incx )
```

Description

The routine performs a series of row interchanges on the matrix *A*. One row interchange is initiated for each of rows *k1* through *k2* of *A*.

Input Parameters

<i>n</i>	INTEGER. The number of columns of the matrix <i>A</i> .
<i>a</i>	REAL for <i>slaswp</i> DOUBLE PRECISION for <i>dlaswp</i> COMPLEX for <i>claswp</i> COMPLEX*16 for <i>zlaswp</i> . Array, DIMENSION (<i>lda</i> , <i>n</i>). On entry, the matrix of column dimension <i>n</i> to which the row interchanges will be applied.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> .
<i>k1</i>	INTEGER. The first element of <i>ipiv</i> for which a row interchange will be done.
<i>k2</i>	INTEGER. The last element of <i>ipiv</i> for which a row interchange will be done.
<i>ipiv</i>	INTEGER. Array, DIMENSION (<i>k2</i> * <i>incx</i>). The vector of pivot indices. Only the elements in positions <i>k1</i> through <i>k2</i> of <i>ipiv</i> are accessed. <i>ipiv</i> (<i>k</i>) = <i>l</i> implies rows <i>k</i> and <i>l</i> are to be interchanged.
<i>incx</i>	INTEGER. The increment between successive values of <i>ipiv</i> . If <i>ipiv</i> is negative, the pivots are applied in reverse order.

Output Parameters

<i>a</i>	On exit, the permuted matrix.

?lasy2

Solves the Sylvester matrix equation where the matrices are of order 1 or 2.

Syntax

```
call slasy2( ltranl, ltranr, isgn, n1, n2, tl, ldtl, tr, ldtr, b, ldb, scale,
x, ldx, xnorm, info )
```

```
call dlasy2( ltranl, ltranr, isgn, n1, n2, tl, ldtl, tr, ldtr, b, ldb, scale,
x, ldx, xnorm, info )
```

Description

The routine solves for the $n1$ -by- $n2$ matrix X , $1 \leq n1, n2 \leq 2$, in

$$\text{op}(TL) * X + \text{isgn} * X * \text{op}(TR) = \text{scale} * B,$$

where

TL is $n1$ -by- $n1$,

TR is $n2$ -by- $n2$,

B is $n1$ -by- $n2$,

and $\text{isgn} = 1$ or -1 . $\text{op}(T) = T$ or T^T , where T^T denotes the transpose of T .

Input Parameters

<i>ltranl</i>	LOGICAL. On entry, <i>ltranl</i> specifies the $\text{op}(TL)$: = <code>.FALSE.</code> , $\text{op}(TL) = TL$, = <code>.TRUE.</code> , $\text{op}(TL) = (TL)^T$.
<i>ltranr</i>	LOGICAL. On entry, <i>ltranr</i> specifies the $\text{op}(TR)$: = <code>.FALSE.</code> , $\text{op}(TR) = TR$, = <code>.TRUE.</code> , $\text{op}(TR) = (TR)^T$.
<i>isgn</i>	INTEGER. On entry, <i>isgn</i> specifies the sign of the equation as described before. <i>isgn</i> may only be 1 or -1.
<i>n1</i>	INTEGER. On entry, <i>n1</i> specifies the order of matrix TL . <i>n1</i> may only be 0, 1 or 2.

<i>n2</i>	INTEGER. On entry, <i>n2</i> specifies the order of matrix <i>TR</i> . <i>n2</i> may only be 0, 1 or 2.
<i>t1</i>	REAL for slasy2 DOUBLE PRECISION for dlasy2. Array, DIMENSION (<i>ldt1</i> ,2). On entry, <i>t1</i> contains an <i>n1</i> -by- <i>n1</i> matrix <i>TL</i> .
<i>ldt1</i>	INTEGER. The leading dimension of the matrix <i>TL</i> . $ldt1 \geq \max(1, n1)$.
<i>tr</i>	REAL for slasy2 DOUBLE PRECISION for dlasy2. Array, DIMENSION (<i>ldtr</i> ,2). On entry, <i>tr</i> contains an <i>n2</i> -by- <i>n2</i> matrix <i>TR</i> .
<i>ldtr</i>	INTEGER. The leading dimension of the matrix <i>TR</i> . $ldtr \geq \max(1, n2)$.
<i>b</i>	REAL for slasy2 DOUBLE PRECISION for dlasy2. Array, DIMENSION (<i>ldb</i> ,2). On entry, the <i>n1</i> -by- <i>n2</i> matrix <i>B</i> contains the right-hand side of the equation.
<i>ldb</i>	INTEGER. The leading dimension of the matrix <i>B</i> . $ldb \geq \max(1, n1)$.
<i>ldx</i>	INTEGER. The leading dimension of the output matrix <i>X</i> . $ldx \geq \max(1, n1)$.

Output Parameters

<i>scale</i>	REAL for slasy2 DOUBLE PRECISION for dlasy2. On exit, <i>scale</i> contains the scale factor. <i>scale</i> is chosen less than or equal to 1 to prevent the solution overflowing.
<i>x</i>	REAL for slasy2 DOUBLE PRECISION for dlasy2. Array, DIMENSION (<i>ldx</i> ,2). On exit, <i>x</i> contains the <i>n1</i> -by- <i>n2</i> solution.
<i>xnorm</i>	REAL for slasy2

info DOUBLE PRECISION for `dlasy2`.
 On exit, *xnorm* is the infinity-norm of the solution.
 INTEGER. On exit, *info* is set to 0: successful exit. 1: *TL* and *TR* have too close eigenvalues, so *TL* or *TR* is perturbed to get a nonsingular equation.



NOTE. For higher speed, this routine does not check the inputs for errors.

?lasyf

Computes a partial factorization of a real/complex symmetric matrix, using the diagonal pivoting method.

Syntax

```
call slasyf( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call dlasyf( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call clasyf( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call zlasyf( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
```

Description

The routine `?lasyf` computes a partial factorization of a real/complex symmetric matrix *A* using the Bunch-Kaufman diagonal pivoting method. The partial factorization has the form:

$$A = \begin{bmatrix} I & U_{12} \\ 0 & U_{12}' \end{bmatrix} \begin{bmatrix} A_{11} & 0 \\ 0 & D \end{bmatrix} \begin{bmatrix} I & 0 \\ U_{12}' & U_{12}' \end{bmatrix} \text{ if } uplo = 'U', \text{ or}$$

$$A = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix} \begin{bmatrix} D & 0 \\ 0 & A_{22} \end{bmatrix} \begin{bmatrix} L_{12}' & L_{21}' \\ 0 & I \end{bmatrix} \text{ if } uplo = 'L'$$

where the order of *D* is at most *nb*.

The actual order is returned in the argument *kb*, and is either *nb* or *nb-1*, or *n* if $n \leq nb$.

This is an auxiliary routine called by `?sytrf`. It uses blocked code (calling Level 3 BLAS) to update the submatrix A_{11} (if *uplo* = 'U') or A_{22} (if *uplo* = 'L').

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix <i>A</i> is stored: = 'U': Upper triangular = 'L': Lower triangular
<i>n</i>	INTEGER. The order of the matrix <i>A</i> . $n \geq 0$.
<i>nb</i>	INTEGER. The maximum number of columns of the matrix <i>A</i> that should be factored. <i>nb</i> should be at least 2 to allow for 2-by-2 pivot blocks.
<i>a</i>	REAL for <code>slasyf</code> DOUBLE PRECISION for <code>dlasyf</code> COMPLEX for <code>clasyf</code> COMPLEX*16 for <code>zlasyf</code> . Array, DIMENSION (<i>lda</i> , <i>n</i>). If <i>uplo</i> = 'U', the leading <i>n</i> -by- <i>n</i> upper triangular part of <i>a</i> contains the upper triangular part of the matrix <i>A</i> , and the strictly lower triangular part of <i>a</i> is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i> -by- <i>n</i> lower triangular part of <i>a</i> contains the lower triangular part of the matrix <i>A</i> , and the strictly upper triangular part of <i>a</i> is not referenced.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $lda \geq \max(1, n)$.
<i>w</i>	REAL for <code>slasyf</code> DOUBLE PRECISION for <code>dlasyf</code> COMPLEX for <code>clasyf</code> COMPLEX*16 for <code>zlasyf</code> . Workspace array, DIMENSION (<i>ldw</i> , <i>nb</i>).
<i>ldw</i>	INTEGER. The leading dimension of the array <i>w</i> . $ldw \geq \max(1, n)$.

Output Parameters

<i>kb</i>	INTEGER. The number of columns of <i>A</i> that were actually factored <i>kb</i> is either <i>nb</i> -1 or <i>nb</i> , or <i>n</i> if $n \leq nb$.
<i>a</i>	On exit, <i>a</i> contains details of the partial factorization.
<i>ipiv</i>	INTEGER. Array, DIMENSION (<i>n</i>). Details of the interchanges and the block structure of <i>D</i> . If <i>uplo</i> = 'U', only the last <i>kb</i> elements of <i>ipiv</i> are set; if <i>uplo</i> = 'L', only the first <i>kb</i> elements are set. If <i>ipiv</i> (<i>k</i>) > 0, then rows and columns <i>k</i> and <i>ipiv</i> (<i>k</i>) were interchanged and <i>D</i> (<i>k</i> , <i>k</i>) is a 1-by-1 diagonal block. If <i>uplo</i> = 'U' and <i>ipiv</i> (<i>k</i>) = <i>ipiv</i> (<i>k</i> -1) < 0, then rows and columns <i>k</i> -1 and - <i>ipiv</i> (<i>k</i>) were interchanged and <i>D</i> (<i>k</i> -1: <i>k</i> , <i>k</i> -1: <i>k</i>) is a 2-by-2 diagonal block. If <i>uplo</i> = 'L' and <i>ipiv</i> (<i>k</i>) = <i>ipiv</i> (<i>k</i> +1) < 0, then rows and columns <i>k</i> +1 and - <i>ipiv</i> (<i>k</i>) were interchanged and <i>D</i> (<i>k</i> : <i>k</i> +1, <i>k</i> : <i>k</i> +1) is a 2-by-2 diagonal block.
<i>info</i>	INTEGER. = 0: successful exit > 0: if <i>info</i> = <i>k</i> , <i>D</i> (<i>k</i> , <i>k</i>) is exactly zero. The factorization has been completed, but the block diagonal matrix <i>D</i> is exactly singular.

?lahef

Computes a partial factorization of a complex Hermitian indefinite matrix, using the diagonal pivoting method.

Syntax

```
call clahef( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call zlahef( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
```

Description

The routine ?lahef computes a partial factorization of a complex Hermitian matrix *A*, using the Bunch-Kaufman diagonal pivoting method. The partial factorization has the form:

$$A = \begin{bmatrix} I & U_{12} \\ 0 & U_{12}' \end{bmatrix} \begin{bmatrix} A_{11} & 0 \\ 0 & D \end{bmatrix} \begin{bmatrix} I & 0 \\ U_{12}' & U_{12}' \end{bmatrix} \text{ if } uplo = 'U', \text{ or}$$

$$A = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix} \begin{bmatrix} D & 0 \\ 0 & A_{22} \end{bmatrix} \begin{bmatrix} L_{12}' & L_{21}' \\ 0 & I \end{bmatrix} \text{ if } uplo = 'L'$$

where the order of D is at most nb .

The actual order is returned in the argument kb , and is either nb or $nb-1$, or n if $n \leq nb$.

Note that U' denotes the conjugate transpose of U .

This is an auxiliary routine called by `?hetrf`. It uses blocked code (calling Level 3 BLAS) to update the submatrix A_{11} (if $uplo = 'U'$) or A_{22} (if $uplo = 'L'$).

Input Parameters

$uplo$	CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored: = 'U': upper triangular = 'L': lower triangular
n	INTEGER. The order of the matrix A . $n \geq 0$.
nb	INTEGER. The maximum number of columns of the matrix A that should be factored. nb should be at least 2 to allow for 2-by-2 pivot blocks.
a	COMPLEX for <code>claherf</code> COMPLEX*16 for <code>zlaherf</code> . Array, DIMENSION (lda , n). On entry, the Hermitian matrix A . If $uplo = 'U'$, the leading n -by- n upper triangular part of A contains the upper triangular part of the matrix A , and the strictly lower triangular part of A is not referenced.

If $uplo = 'L'$, the leading n -by- n lower triangular part of A contains the lower triangular part of the matrix A , and the strictly upper triangular part of A is not referenced.

lda INTEGER. The leading dimension of the array a . $lda \geq \max(1, n)$.

w COMPLEX for `clahef`
COMPLEX*16 for `zlahef`.
Workspace array, DIMENSION (ldw, nb).

ldw INTEGER. The leading dimension of the array w . $ldw \geq \max(1, n)$.

Output Parameters

kb INTEGER. The number of columns of A that were actually factored kb is either $nb-1$ or nb , or n if $n \leq nb$.

a On exit, A contains details of the partial factorization.

ipiv INTEGER.
Array, DIMENSION (n). Details of the interchanges and the block structure of D .
If $uplo = 'U'$, only the last kb elements of $ipiv$ are set;
if $uplo = 'L'$, only the first kb elements are set.
If $ipiv(k) > 0$, then rows and columns k and $ipiv(k)$ are interchanged and $D(k, k)$ is a 1-by-1 diagonal block.
If $uplo = 'U'$ and $ipiv(k) = ipiv(k-1) < 0$, then rows and columns $k-1$ and $-ipiv(k)$ are interchanged and $D(k-1:k, k-1:k)$ is a 2-by-2 diagonal block.
If $uplo = 'L'$ and $ipiv(k) = ipiv(k+1) < 0$, then rows and columns $k+1$ and $-ipiv(k)$ are interchanged and $D(k:k+1, k:k+1)$ is a 2-by-2 diagonal block.

info INTEGER.
= 0: successful exit
> 0: if $info = k$, $D(k, k)$ is exactly zero. The factorization has been completed, but the block diagonal matrix D is exactly singular.

?latbs

Solves a triangular banded system of equations.

Syntax

```
call slatbs( uplo, trans, diag, normin, n, kd, ab, ldab, x, scale, cnorm,
info )
```

```
call dlatbs( uplo, trans, diag, normin, n, kd, ab, ldab, x, scale, cnorm,
info )
```

```
call clatbs( uplo, trans, diag, normin, n, kd, ab, ldab, x, scale, cnorm,
info )
```

```
call zlatbs( uplo, trans, diag, normin, n, kd, ab, ldab, x, scale, cnorm,
info )
```

Description

The routine solves one of the triangular systems

$A*x = s*b$, or $A^T*x = s*b$, or $A^H*x = s*b$ (for complex flavors)

with scaling to prevent overflow, where A is an upper or lower triangular band matrix. Here A^T denotes the transpose of A , A^H denotes the conjugate transpose of A , x and b are n -element vectors, and s is a scaling factor, usually less than or equal to 1, chosen so that the components of x will be less than the overflow threshold. If the unscaled problem will not cause overflow, the Level 2 BLAS routine ?tbsv is called. If the matrix A is singular ($A(j, j)=0$ for some j), then s is set to 0 and a non-trivial solution to $A*x = 0$ is returned.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the matrix A is upper or lower triangular. = 'U': upper triangular = 'L': lower triangular
<i>trans</i>	CHARACTER*1. Specifies the operation applied to A . = 'N': solve $A*x = s*b$ (no transpose) = 'T': solve $A^T*x = s*b$ (transpose) = 'C': solve $A^H*x = s*b$ (conjugate transpose)

<i>diag</i>	CHARACTER*1. Specifies whether the matrix <i>A</i> is unit triangular = 'N': non-unit triangular = 'U': unit triangular
<i>normin</i>	CHARACTER*1. Specifies whether <i>cnorm</i> is set. = 'Y': <i>cnorm</i> contains the column norms on entry; = 'N': <i>cnorm</i> is not set on entry. On exit, the norms is computed and stored in <i>cnorm</i>.
<i>n</i>	INTEGER. The order of the matrix <i>A</i>. $n \geq 0$.
<i>kd</i>	INTEGER. The number of subdiagonals or superdiagonals in the triangular matrix <i>A</i>. $kb \geq 0$.
<i>ab</i>	REAL for slatbs DOUBLE PRECISION for dlatbs COMPLEX for clatbs COMPLEX*16 for zlatbs. Array, DIMENSION (<i>ldab</i>, <i>n</i>). The upper or lower triangular band matrix <i>A</i>, stored in the first <i>kb</i>+1 rows of the array. The <i>j</i>-th column of <i>A</i> is stored in the <i>j</i>-th column of the array <i>ab</i> as follows: if <i>uplo</i> = 'U', $ab(kd+1+i-j, j) = A(i, j)$ for $\max(1, j-kd) \leq i \leq j$; if <i>uplo</i> = 'L', $ab(1+i-j, j) = A(i, j)$ for $j \leq i \leq \min(n, j+kd)$.
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i>. $ldab \geq kb+1$.
<i>x</i>	REAL for slatbs DOUBLE PRECISION for dlatbs COMPLEX for clatbs COMPLEX*16 for zlatbs. Array, DIMENSION (<i>n</i>). On entry, the right hand side <i>b</i> of the triangular system.
<i>cnorm</i>	REAL for slatbs/clatbs DOUBLE PRECISION for dlatbs/zlatbs. Array, DIMENSION (<i>n</i>).

If `NORMIN = 'Y'`, `cnorm` is an input argument and `cnorm(j)` contains the norm of the off-diagonal part of the j -th column of A .

If `trans = 'N'`, `cnorm(j)` must be greater than or equal to the infinity-norm, and if `trans = 'T'` or `'C'`, `cnorm(j)` must be greater than or equal to the 1-norm.

Output Parameters

<code>scale</code>	REAL for <code>slatbs/clatbs</code> DOUBLE PRECISION for <code>dlatbs/zlatbs</code>. The scaling factor s for the triangular system as described above. If <code>scale = 0</code>, the matrix A is singular or badly scaled, and the vector x is an exact or approximate solution to $Ax = 0$.
<code>cnorm</code>	If <code>normin = 'N'</code>, <code>cnorm</code> is an output argument and <code>cnorm(j)</code> returns the 1-norm of the off-diagonal part of the j-th column of A.
<code>info</code>	INTEGER. = 0: successful exit < 0: if <code>info = -k</code>, the k-th argument had an illegal value

?latdf

Uses the LU factorization of the n -by- n matrix computed by ?getc2 and computes a contribution to the reciprocal Dif-estimate.

Syntax

```
call slatdf( ijob, n, z, ldz, rhs, rdsum, rdscal, ipiv, jpiv )
call dlatdf( ijob, n, z, ldz, rhs, rdsum, rdscal, ipiv, jpiv )
call clatdf( ijob, n, z, ldz, rhs, rdsum, rdscal, ipiv, jpiv )
call zlatdf( ijob, n, z, ldz, rhs, rdsum, rdscal, ipiv, jpiv )
```


Description

The routine `?latdf` uses the LU factorization of the n -by- n matrix Z computed by `?getc2` and computes a contribution to the reciprocal Dif-estimate by solving $Z^*x = b$ for x , and choosing the right-hand side b such that the norm of x is as large as possible. On entry $rhs = b$ holds the contribution from earlier solved sub-systems, and on return $rhs = x$.

The factorization of Z returned by `?getc2` has the form $Z = P^*L^*U^*Q$, where P and Q are permutation matrices. L is lower triangular with unit diagonal elements and U is upper triangular.

Input Parameters

ijob INTEGER.
ijob = 2: First compute an approximative null-vector e of Z using `?gecon`, e is normalized, and solve for $Z^*x = \pm e - f$ with the sign giving the greater value of 2-norm(x). This option is about 5 times as expensive as default.
ijob $\neq$ 2 (default): Local look ahead strategy where all entries of the right-hand side b is chosen as either +1 or -1.
 .

n INTEGER. The number of columns of the matrix Z .

z REAL for `slatdf/clatdf`
 DOUBLE PRECISION for `dlatdf/zlatdf`.
 Array, DIMENSION (*ldz*, *n*)
 On entry, the LU part of the factorization of the n -by- n matrix Z computed by `?getc2`: $Z = P^*L^*U^*Q$.

ldz INTEGER. The leading dimension of the array Z . $lda \geq \max(1, n)$.

rhs REAL for `slatdf/clatdf`
 DOUBLE PRECISION for `dlatdf/zlatdf`.
 Array, DIMENSION (*n*).
 On entry, *rhs* contains contributions from other subsystems.

rdsum REAL for `slatdf/clatdf`
 DOUBLE PRECISION for `dlatdf/zlatdf`.
 On entry, the sum of squares of computed contributions to the Dif-estimate under computation by `?tgsyL`, where the scaling factor *rdscal* has been factored out. If *trans* = 'T', *rdsum* is not touched.

	Note that <i>rdsum</i> only makes sense when <code>?tgssy2</code> is called by <code>?tgssyL</code>.
<i>rdscal</i>	REAL for <code>slatdf/clatdf</code> DOUBLE PRECISION for <code>dlatdf/zlatdf</code>. On entry, scaling factor used to prevent overflow in <i>rdsum</i>. If <i>trans</i> = <i>T'</i>, <i>rdscal</i> is not touched. Note that <i>rdscal</i> only makes sense when <code>?tgssy2</code> is called by <code>?tgssyL</code>.
<i>ipiv</i>	INTEGER. Array, DIMENSION (<i>n</i>). The pivot indices; for $1 \leq i \leq n$, row <i>i</i> of the matrix has been interchanged with row <i>ipiv</i>(<i>i</i>).
<i>jpiv</i>	INTEGER. Array, DIMENSION (<i>n</i>). The pivot indices; for $1 \leq j \leq n$, column <i>j</i> of the matrix has been interchanged with column <i>jpiv</i>(<i>j</i>).

Output Parameters

<i>rhs</i>	On exit, <i>rhs</i> contains the solution of the subsystem with entries according to the value of <i>ijob</i>.
<i>rdsum</i>	On exit, the corresponding sum of squares updated with the contributions from the current sub-system. If <i>trans</i> = <i>T'</i>, <i>rdsum</i> is not touched.
<i>rdscal</i>	On exit, <i>rdscal</i> is updated with respect to the current contributions in <i>rdsum</i>. If <i>trans</i> = <i>T'</i>, <i>rdscal</i> is not touched.

?latps

Solves a triangular system of equations with the matrix held in packed storage.

Syntax

```
call slatps( uplo, trans, diag, normin, n, ap, x, scale, cnorm, info )
call dlatps( uplo, trans, diag, normin, n, ap, x, scale, cnorm, info )
call clatps( uplo, trans, diag, normin, n, ap, x, scale, cnorm, info )
call zlatps( uplo, trans, diag, normin, n, ap, x, scale, cnorm, info )
```

Description

The routine ?latps solves one of the triangular systems

$A^*x = s*b$, or $A^T*x = s*b$, or $A^H*x = s*b$ (for complex flavors)

with scaling to prevent overflow, where A is an upper or lower triangular matrix stored in packed form. Here A^T denotes the transpose of A , A^H denotes the conjugate transpose of A , x and b are n -element vectors, and s is a scaling factor, usually less than or equal to 1, chosen so that the components of x will be less than the overflow threshold. If the unscaled problem does not cause overflow, the Level 2 BLAS routine ?tpsv is called. If the matrix A is singular ($A(j, j) = 0$ for some j), then s is set to 0 and a non-trivial solution to $A*x = 0$ is returned.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the matrix A is upper or lower triangular. = 'U': upper triangular = 'L': uower triangular
<i>trans</i>	CHARACTER*1. Specifies the operation applied to A . = 'N': solve $A*x = s*b$ (no transpose) = 'T': solve $A^T*x = s*b$ (transpose) = 'C': solve $A^H*x = s*b$ (conjugate transpose)
<i>diag</i>	CHARACTER*1. Specifies whether the matrix A is unit triangular. = 'N': non-unit triangular = 'U': unit triangular

<i>normin</i>	CHARACTER*1. Specifies whether <i>cnorm</i> is set. = 'Y': <i>cnorm</i> contains the column norms on entry; = 'N': <i>cnorm</i> is not set on entry. On exit, the norms will be computed and stored in <i>cnorm</i>.
<i>n</i>	INTEGER. The order of the matrix <i>A</i>. $n \geq 0$.
<i>ap</i>	REAL for slatps DOUBLE PRECISION for dlatps COMPLEX for clatps COMPLEX*16 for zlatps. Array, DIMENSION $(n(n+1)/2)$. The upper or lower triangular matrix <i>A</i>, packed columnwise in a linear array. The <i>j</i>-th column of <i>A</i> is stored in the array <i>ap</i> as follows: if <i>uplo</i> = 'U', $ap(i + (j-1)j/2) = A(i, j)$ for $1 \leq i \leq j$; if <i>uplo</i> = 'L', $ap(i + (j-1)(2n-j)/2) = A(i, j)$ for $j \leq i \leq n$.
<i>x</i>	REAL for slatps DOUBLE PRECISION for dlatps COMPLEX for clatps COMPLEX*16 for zlatps. Array, DIMENSION (n) On entry, the right hand side <i>b</i> of the triangular system.
<i>cnorm</i>	REAL for slatps/clatps DOUBLE PRECISION for dlatps/zlatps. Array, DIMENSION (n). If <i>normin</i> = 'Y', <i>cnorm</i> is an input argument and <i>cnorm</i>(<i>j</i>) contains the norm of the off-diagonal part of the <i>j</i>-th column of <i>A</i>. If <i>trans</i> = 'N', <i>cnorm</i>(<i>j</i>) must be greater than or equal to the infinity-norm, and if <i>trans</i> = 'T' or 'C', <i>cnorm</i>(<i>j</i>) must be greater than or equal to the 1-norm.

Output Parameters

<i>x</i>	On exit, <i>x</i> is overwritten by the solution vector <i>x</i> .

scale REAL for slatps/clatps
 DOUBLE PRECISION for dlatps/zlatps.
 The scaling factor s for the triangular system as described above.
 If $scale = 0$, the matrix A is singular or badly scaled, and the vector x is an exact or approximate solution to $A*x = 0$.

cnorm If $normin = 'N'$, *cnorm* is an output argument and *cnorm*(j) returns the 1-norm of the off-diagonal part of the j -th column of A .

info INTEGER.
 = 0: successful exit
 < 0: if $info = -k$, the k -th argument had an illegal value

?latrd

Reduces the first nb rows and columns of a symmetric/Hermitian matrix A to real tridiagonal form by an orthogonal/unitary similarity transformation.

Syntax

```
call slatrd( uplo, n, nb, a, lda, e, tau, w, ldw )
call dlatrd( uplo, n, nb, a, lda, e, tau, w, ldw )
call clatrd( uplo, n, nb, a, lda, e, tau, w, ldw )
call zlatrd( uplo, n, nb, a, lda, e, tau, w, ldw )
```

Description

The routine ?latrd reduces nb rows and columns of a real symmetric or complex Hermitian matrix A to symmetric/Hermitian tridiagonal form by an orthogonal/unitary similarity transformation $Q^T A Q$ for real flavors, $Q^H A Q$ for complex flavors, and returns the matrices v and w which are needed to apply the transformation to the unreduced part of A .

If $uplo = 'U'$, ?latrd reduces the last nb rows and columns of a matrix, of which the upper triangle is supplied;

if `uplo = 'L'`, `?latrd` reduces the first `nb` rows and columns of a matrix, of which the lower triangle is supplied.

This is an auxiliary routine called by `?sytrd/?hetrd`.

Input Parameters

<code>uplo</code>	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix <i>A</i> is stored: = 'U': upper triangular = 'L': lower triangular
<code>n</code>	INTEGER. The order of the matrix <i>A</i>.
<code>nb</code>	INTEGER. The number of rows and columns to be reduced.
<code>a</code>	REAL for <code>slatrd</code> DOUBLE PRECISION for <code>dlatrd</code> COMPLEX for <code>clatrd</code> COMPLEX*16 for <code>zlatrd</code>. Array, DIMENSION (<i>lda</i>, <i>n</i>). On entry, the symmetric/Hermitian matrix <i>A</i> If <code>uplo = 'U'</code>, the leading <i>n</i>-by-<i>n</i> upper triangular part of <i>a</i> contains the upper triangular part of the matrix <i>A</i>, and the strictly lower triangular part of <i>a</i> is not referenced. If <code>uplo = 'L'</code>, the leading <i>n</i>-by-<i>n</i> lower triangular part of <i>a</i> contains the lower triangular part of the matrix <i>A</i>, and the strictly upper triangular part of <i>a</i> is not referenced.
<code>lda</code>	INTEGER. The leading dimension of the array <i>a</i>. <i>lda</i> ≥ (1, <i>n</i>).
<code>ldw</code>	INTEGER. The leading dimension of the output array <i>w</i>. <i>ldw</i> ≥ max(1, <i>n</i>).

Output Parameters

<code>a</code>	On exit, if <code>uplo = 'U'</code>, the last <code>nb</code> columns have been reduced to tridiagonal form, with the diagonal elements overwriting the diagonal elements of <i>a</i>; the elements above

the diagonal with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors;
 if *uplo* = 'L', the first *nb* columns have been reduced to tridiagonal form, with the diagonal elements overwriting the diagonal elements of *a*; the elements below the diagonal with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors.

e REAL for slatrd/clatrd
 DOUBLE PRECISION for dlatrd/zlatrd.
 If *uplo* = 'U', *e*(*n-nb*:*n-1*) contains the superdiagonal elements of the last *nb* columns of the reduced matrix;
 if *uplo* = 'L', *e*(1:*nb*) contains the subdiagonal elements of the first *nb* columns of the reduced matrix.

tau REAL for slatrd
 DOUBLE PRECISION for dlatrd
 COMPLEX for clatrd
 COMPLEX*16 for zlatrd.
 Array, DIMENSION (*lda*, *n*).
 The scalar factors of the elementary reflectors, stored in *tau*(*n-nb*:*n-1*) if *uplo* = 'U', and in *tau*(1:*nb*) if *uplo* = 'L'.

w REAL for slatrd
 DOUBLE PRECISION for dlatrd
 COMPLEX for clatrd
 COMPLEX*16 for zlatrd.
 Array, DIMENSION (*lda*, *n*).
 The *n*-by-*nb* matrix *w* required to update the unreduced part of *A*.

Application Notes

If *uplo* = 'U', the matrix *Q* is represented as a product of elementary reflectors

$$Q = H(n) * H(n-1) * \dots * H(n-nb+1)$$

Each $H(i)$ has the form

$$H(i) = I - \tau u v^*$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(i:n) = 0$ and $v(i-1) = 1$; $v(1:i-1)$ is stored on exit in $a(1:i-1, i)$, and τ in $\tau(i-1)$.

If $uplo = 'L'$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(1) * H(2) * \dots * H(nb)$$

Each $H(i)$ has the form $H(i) = I - \tau v v'$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i) = 0$ and $v(i+1) = 1$; $v(i+1:n)$ is stored on exit in $a(i+1:n, i)$, and τ in $\tau(i)$.

The elements of the vectors v together form the n -by- nb matrix V which is needed, with W , to apply the transformation to the unreduced part of the matrix, using a symmetric/Hermitian rank-2k update of the form:

$$A := A - VW' - WV'.$$

The contents of a on exit are illustrated by the following examples with $n = 5$ and $nb = 2$:

if $uplo = 'U'$:

$$\begin{bmatrix} a & a & a & v_1 & v_1 \\ & a & a & v_1 & v_1 \\ & & a & 1 & v_1 \\ & & & d & 1 \\ & & & & d \end{bmatrix}$$

if $uplo = 'L'$:

$$\begin{bmatrix} d & & & & \\ & 1 & d & & \\ v_1 & 1 & a & & \\ v_1 & v_1 & a & a & \\ v_1 & v_1 & a & a & a \end{bmatrix}$$

where d denotes a diagonal element of the reduced matrix, a denotes an element of the original matrix that is unchanged, and v_i denotes an element of the vector defining $H(i)$.

?latrs

Solves a triangular system of equations with the scale factor set to prevent overflow.

Syntax

```
call slatrs( uplo, trans, diag, normin, n, a, lda, x, scale, cnorm, info )
call dlatrs( uplo, trans, diag, normin, n, a, lda, x, scale, cnorm, info )
call clatrs( uplo, trans, diag, normin, n, a, lda, x, scale, cnorm, info )
call zlatrs( uplo, trans, diag, normin, n, a, lda, x, scale, cnorm, info )
```

Description

The routine solves one of the triangular systems

$A^*x = s*b$, or $A^T*x = s*b$, or $A^H*x = s*b$ (for complex flavors)

with scaling to prevent overflow. Here A is an upper or lower triangular matrix, A^T denotes the transpose of A , A^H denotes the conjugate transpose of A , x and b are n -element vectors, and s is a scaling factor, usually less than or equal to 1, chosen so that the components of x will be less than the overflow threshold. If the unscaled problem will not cause overflow, the Level 2 BLAS routine `?trsv` is called. If the matrix A is singular ($A(j,j) = 0$ for some j), then s is set to 0 and a non-trivial solution to $A*x = 0$ is returned.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the matrix A is upper or lower triangular. = 'U': Upper triangular = 'L': Lower triangular
<i>trans</i>	CHARACTER*1. Specifies the operation applied to A . = 'N': solve $A*x = s*b$ (no transpose) = 'T': solve $A^T*x = s*b$ (transpose) = 'C': solve $A^H*x = s*b$ (conjugate transpose)
<i>diag</i>	CHARACTER*1. Specifies whether or not the matrix A is unit triangular. = 'N': non-unit triangular = 'U': non-unit triangular

<i>normin</i>	CHARACTER*1. Specifies whether <i>cnorm</i> has been set or not. = 'Y': <i>cnorm</i> contains the column norms on entry; = 'N': <i>cnorm</i> is not set on entry. 0 n exit, the norms will be computed and stored in <i>cnorm</i>.
<i>n</i>	INTEGER. The order of the matrix <i>A</i>. $n \geq 0$
<i>a</i>	REAL for slatrs DOUBLE PRECISION for dlatrs COMPLEX for clatrs COMPLEX*16 for zlatrs. Array, DIMENSION (<i>lda</i>, <i>n</i>). Contains the triangular matrix <i>A</i>. If <i>uplo</i> = 'U', the leading <i>n</i>-by-<i>n</i> upper triangular part of the array <i>a</i> contains the upper triangular matrix, and the strictly lower triangular part of <i>A</i> is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i>-by-<i>n</i> lower triangular part of the array <i>a</i> contains the lower triangular matrix, and the strictly upper triangular part of <i>A</i> is not referenced. If <i>diag</i> = 'U', the diagonal elements of <i>A</i> are also not referenced and are assumed to be 1.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i>. $lda \geq \max(1, n)$.
<i>x</i>	REAL for slatrs DOUBLE PRECISION for dlatrs COMPLEX for clatrs COMPLEX*16 for zlatrs. Array, DIMENSION (<i>n</i>). On entry, the right hand side <i>b</i> of the triangular system.
<i>cnorm</i>	REAL for slatrs/clatrs DOUBLE PRECISION for dlatrs/zlatrs. Array, DIMENSION (<i>n</i>). If <i>normin</i> = 'Y', <i>cnorm</i> is an input argument and <i>cnorm</i> (<i>j</i>) contains the norm of the off-diagonal part of the <i>j</i>-th column of <i>A</i>.

If $trans = 'N'$, $cnorm(j)$ must be greater than or equal to the infinity-norm, and if $trans = 'T'$ or $'C'$, $cnorm(j)$ must be greater than or equal to the 1-norm.

Output Parameters

x	On exit, x is overwritten by the solution vector x .
$scale$	REAL for slatrs/clatrs DOUBLE PRECISION for dlatrs/zlatrs. Array, DIMENSION (lda, n). The scaling factor s for the triangular system as described above. If $scale = 0$, the matrix A is singular or badly scaled, and the vector x is an exact or approximate solution to $A*x = 0$.
$cnorm$	If $normin = 'N'$, $cnorm$ is an output argument and $cnorm(j)$ returns the 1-norm of the off-diagonal part of the j -th column of A .
$info$	INTEGER. = 0: successful exit < 0: if $info = -k$, the k -th argument had an illegal value

Application Notes

A rough bound on x is computed; if that is less than overflow, `?trsv` is called, otherwise, specific code is used which checks for possible overflow or divide-by-zero at every operation.

A columnwise scheme is used for solving $Ax = b$. The basic algorithm if A is lower triangular is

```
x[1:n] := b[1:n]
for j = 1, ..., n
  x(j) := x(j) / A(j,j)
  x[j+1:n] := x[j+1:n] - x(j)*a[j+1:n,j]
end
```

Define bounds on the components of x after j iterations of the loop:

```
M(j) = bound on x[1:j]
G(j) = bound on x[j+1:n]
```

Initially, let $M(0) = 0$ and $G(0) = \max\{x(i), i=1, \dots, n\}$.

Then for iteration $j+1$ we have

$$M(j+1) \leq G(j) / |a(j+1, j+1)|$$

$$G(j+1) \leq G(j) + M(j+1) * |a[j+2:n, j+1]|$$

$$\leq G(j) (1 + cnorm(j+1) / |a(j+1, j+1)|),$$

where $cnorm(j+1)$ is greater than or equal to the infinity-norm of column $j+1$ of a , not counting the diagonal. Hence

$$G(j) \leq G(0) \prod_{1 \leq i \leq j} (1 + cnorm(i)/|A(i,i)|)$$

and

$$|x(j)| \leq (G(0)/|A(j,j)|) \prod_{1 \leq i \leq j} (1 + cnorm(i)/|A(i,i)|)$$

Since $|x(j)| \leq M(j)$, we use the Level 2 BLAS routine `?trsv` if the reciprocal of the largest $M(j)$, $j=1, \dots, n$, is larger than $\max(\text{underflow}, 1/\text{overflow})$.

The bound on $x(j)$ is also used to determine when a step in the columnwise method can be performed without fear of overflow. If the computed bound is greater than a large constant, x is scaled to prevent overflow, but if the bound overflows, x is set to 0, $x(j)$ to 1, and scale to 0, and a non-trivial solution to $Ax = 0$ is found.

Similarly, a row-wise scheme is used to solve $A^T x = b$ or $A^H x = b$. The basic algorithm for A upper triangular is

for $j = 1, \dots, n$

$x(j) := (b(j) - A[1:j-1, j]' * x[1:j-1]) / A(j, j)$

end

We simultaneously compute two bounds

$$G(j) = \text{bound on } (b(i) - A[1:i-1, i]' * x[1:i-1]), \quad 1 \leq i \leq j$$

$M(j)$ = bound on $x(i)$, $1 \leq i \leq j$

The initial values are $G(0) = 0$, $M(0) = \max\{b(i), i=1, \dots, n\}$, and we add the constraint $G(j) \geq G(j-1)$ and $M(j) \geq M(j-1)$ for $j \geq 1$.

Then the bound on $x(j)$ is

$$M(j) \leq M(j-1) * (1 + cnorm(j)) / |A(j, j)|$$

$$\leq M(0) \prod_{1 \leq i \leq j} (1 + cnorm(i)/|A(i, i)|)$$

and we can safely call ?trsv if $1/M(n)$ and $1/G(n)$ are both greater than $\max(\text{underflow}, 1/\text{overflow})$.

?latrz

Factors an upper trapezoidal matrix by means of orthogonal/unitary transformations.

Syntax

```
call slatz( m, n, l, a, lda, tau, work )
call dlatrz( m, n, l, a, lda, tau, work )
call clatz( m, n, l, a, lda, tau, work )
call zlatrz( m, n, l, a, lda, tau, work )
```

Description

The routine ?latrz factors the m -by- $(m+1)$ real/complex upper trapezoidal matrix

$$[A1 \ A2] = [A(1:m, 1:m) \ A(1:m, n-l+1:n)]$$

as $(R \ 0)^* Z$, by means of orthogonal/unitary transformations. Z is an $(m+1)$ -by- $(m+1)$ orthogonal/unitary matrix and R and $A1$ are m -by- m upper triangular matrices.

Input Parameters

m INTEGER. The number of rows of the matrix A . $m \geq 0$.

<i>n</i>	INTEGER. The number of columns of the matrix <i>A</i> . $n \geq 0$.
<i>l</i>	INTEGER. The number of columns of the matrix <i>A</i> containing the meaningful part of the Householder vectors. $n-m \geq l \geq 0$.
<i>a</i>	REAL for slatz DOUBLE PRECISION for dlatrz COMPLEX for clatz COMPLEX*16 for zlatrz. Array, DIMENSION (<i>lda</i> , <i>n</i>). On entry, the leading <i>m</i> -by- <i>n</i> upper trapezoidal part of the array <i>a</i> must contain the matrix to be factorized.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $lda \geq \max(1, m)$.
<i>work</i>	REAL for slatz DOUBLE PRECISION for dlatrz COMPLEX for clatz COMPLEX*16 for zlatrz. Workspace array, DIMENSION (<i>m</i>).

Output Parameters

<i>a</i>	On exit, the leading <i>m</i> -by- <i>m</i> upper triangular part of <i>a</i> contains the upper triangular matrix <i>R</i> , and elements $n-l+1$ to <i>n</i> of the first <i>m</i> rows of <i>a</i> , with the array <i>tau</i> , represent the orthogonal/unitary matrix <i>Z</i> as a product of <i>m</i> elementary reflectors.
<i>tau</i>	REAL for slatz DOUBLE PRECISION for dlatrz COMPLEX for clatz COMPLEX*16 for zlatrz. Array, DIMENSION (<i>m</i>). The scalar factors of the elementary reflectors.

Application Notes

The factorization is obtained by Householder's method. The *k*-th transformation matrix, $z(k)$, which is used to introduce zeros into the (*m* - *k* + 1)-th row of *A*, is given in the form

$$Z(k) = \begin{bmatrix} I & 0 \\ 0 & T(k) \end{bmatrix}$$

where

$$T(k) = I - \tau u(k) u(k)^T, \quad u(k) = \begin{bmatrix} 1 \\ 0 \\ z(k) \end{bmatrix}$$

τ is a scalar and $z(k)$ is an l -element vector. τ and $z(k)$ are chosen to annihilate the elements of the k -th row of $A2$.

The scalar τ is returned in the k -th element of τ and the vector $u(k)$ in the k -th row of $A2$, such that the elements of $z(k)$ are in $a(k, l+1), \dots, a(k, n)$.

The elements of r are returned in the upper triangular part of $A1$.

Z is given by

$$Z = Z(1) * Z(2) * \dots * Z(m).$$

slauu2

Computes the product $U*U^T(U*U^H)$ or $L^T*L(L^H*L)$, where U and L are upper or lower triangular matrices (unblocked algorithm).

Syntax

```
call slauu2( uplo, n, a, lda, info )
call dlauu2( uplo, n, a, lda, info )
call clauu2( uplo, n, a, lda, info )
call zlauu2( uplo, n, a, lda, info )
```

Description

The routine ?lauu2 computes the product U^*U^T or L^T*L for real flavors, and L^H*L or L^H*L for complex flavors. Here the triangular factor U or L is stored in the upper or lower triangular part of the array a .

If $uplo = 'U'$ or $'u'$, then the upper triangle of the result is stored, overwriting the factor U in A .

If $uplo = 'L'$ or $'l'$, then the lower triangle of the result is stored, overwriting the factor L in A .

This is the unblocked form of the algorithm, calling [BLAS Level 2 Routines](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the triangular factor stored in the array a is upper or lower triangular: = 'U': Upper triangular = 'L': Lower triangular
<i>n</i>	INTEGER. The order of the triangular factor U or L . $n \geq 0$.
<i>a</i>	REAL for slauu2 DOUBLE PRECISION for dlauu2 COMPLEX for clauu2 COMPLEX*16 for zlauu2. Array, DIMENSION (lda, n). On entry, the triangular factor U or L .
<i>lda</i>	INTEGER. The leading dimension of the array a . $lda \geq \max(1, n)$.

Output Parameters

<i>a</i>	On exit, if $uplo = 'U'$, then the upper triangle of a is overwritten with the upper triangle of the product $U^*U^T(U^*U^H)$; if $uplo = 'L'$, then the lower triangle of a is overwritten with the lower triangle of the product $L^T*L(L^H*L)$.
<i>info</i>	INTEGER. = 0: successful exit

< 0: if *info* = -*k*, the *k*-th argument had an illegal value

?lauum

Computes the product $U^*U^T(U^*U^H)$ or $L^T*L(L^H*L)$, where U and L are upper or lower triangular matrices (blocked algorithm).

Syntax

```
call slauum( uplo, n, a, lda, info )
call dlauum( uplo, n, a, lda, info )
call clauum( uplo, n, a, lda, info )
call zlauum( uplo, n, a, lda, info )
```

Description

The routine ?lauum computes the product U^*U^T or L^T*L for real flavors, and L^H*L or L^H*L for complex flavors. Here the triangular factor U or L is stored in the upper or lower triangular part of the array *a*.

If *uplo* = 'U' or 'u', then the upper triangle of the result is stored, overwriting the factor U in *A*.

If *uplo* = 'L' or 'l', then the lower triangle of the result is stored, overwriting the factor L in *A*.

This is the blocked form of the algorithm, calling [BLAS Level 3 Routines](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the triangular factor stored in the array <i>a</i> is upper or lower triangular: = 'U': Upper triangular = 'L': Lower triangular
<i>n</i>	INTEGER. The order of the triangular factor U or L . $n \geq 0$.
<i>a</i>	REAL for slauum DOUBLE PRECISION for dlauum COMPLEX for clauum

COMPLEX*16 for `zlauum`.

Array, DIMENSION (`lda`, `n`).

On entry, the triangular factor U or L .

`lda`

INTEGER. The leading dimension of the array `a`. $lda \geq \max(1, n)$.

Output Parameters

`a`

On exit,

if `uplo` = 'U', then the upper triangle of `a` is overwritten with the upper triangle of the product $U^*U^T(U^*U^H)$;

if `uplo` = 'L', then the lower triangle of `a` is overwritten with the lower triangle of the product $L^T*L(L^H*L)$.

`info`

INTEGER.

= 0: successful exit

< 0: if `info` = $-k$, the k -th argument had an illegal value

?lazq3

Checks for deflation, computes a shift and calls `dqds`.

Syntax

```
call slazq3( i0, n0, z, pp, dmin, sigma, desig, qmax, nfail, iter, ndiv, ieee,
  ttype, dmin1, dmin2, dn, dn1, dn2, tau )
```

```
call dlazq3( i0, n0, z, pp, dmin, sigma, desig, qmax, nfail, iter, ndiv, ieee,
  ttype, dmin1, dmin2, dn, dn1, dn2, tau )
```

Description

The routine `?lazq3` checks for deflation, computes a shift (`tau`) and calls `dqds`. In case of failure, it changes shifts, and tries again until output is positive.

This routine is a thread safe version of `?lasq3` routine, which passes `ttype`, `dmin1`, `dmin2`, `dn`, `dn1`, `dn2`, and `tau` through the argument list in place of declaring them in a `SAVE` statement.

Input Parameters

`i0`

INTEGER.

	First index.
<i>n0</i>	INTEGER.
	Last index.
<i>z</i>	REAL for slasq3 DOUBLE PRECISION for dlasq3. Array, DIMENSION (4*n). <i>z</i> holds the <i>qd</i> array.
<i>pp</i>	INTEGER. <i>pp</i> =0 for ping, <i>pp</i> =1 for pong.
<i>desig</i>	REAL for slazq3 DOUBLE PRECISION for dlazq3. Lower order part of <i>sigma</i> .
<i>qmax</i>	REAL for slazq3 DOUBLE PRECISION for dlazq3. Maximum value of <i>q</i> .
<i>ieee</i>	LOGICAL. Flag for IEEE or non-IEEE arithmetic (passed to the routine).
<i>ttype</i>	INTEGER. Shift type.
<i>dmin1</i>	REAL for slazq3 DOUBLE PRECISION for dlazq3. Minimum value of <i>d</i> , excluding <i>d</i> (<i>n0</i>) . Should be 0 on entry at the first iteration and should not be modified further.
<i>dmin2</i>	REAL for slazq3 DOUBLE PRECISION for dlazq3. Minimum value of <i>d</i> , excluding <i>d</i> (<i>n0</i>) and <i>d</i> (<i>n0</i> -1) . Should be 0 on entry at the first iteration and should not be modified further.
<i>dn</i>	REAL for slazq3 DOUBLE PRECISION for dlazq3. Contains <i>d</i> (<i>n</i>) . Should be 0 on entry at the first iteration and should not be modified further.
<i>dn1</i>	REAL for slazq3 DOUBLE PRECISION for dlazq3. Contains <i>d</i> (<i>n</i> -1) . Should be 0 on entry at the first iteration and should not be modified further.
<i>dn2</i>	REAL for slazq3 DOUBLE PRECISION for dlazq3.

Contains $d(n-2)$. Should be 0 on entry at the first iteration and should not be modified further.

tau REAL for slazq3
DOUBLE PRECISION for dlazq3.
Shift value

Output Parameters

dmin REAL for slazq3
DOUBLE PRECISION for dlazq3.
Minimum value of d .

sigma REAL for slazq3
DOUBLE PRECISION for dlazq3.
Sum of shifts used in current segment.

desig Lower order part of *sigma*.

nfail INTEGER.
Number of times shift was too big.

iter INTEGER.
Number of iterations.

ndiv INTEGER.
Number of divisions.

ttype Shift type.

dmin1 Minimum value of d , excluding $d(n0)$.

dmin2 Minimum value of d , excluding $d(n0)$ and $d(n0-1)$.

dn $d(n)$.

dn1 $d(n-1)$.

dn2 $d(n-2)$.

tau Shift value.

?lazq4

Computes an approximation to the smallest eigenvalue using values of d from the previous transform.

Syntax

```
call slazq4( i0, n0, z, pp, n0in, dmin, dmin1, dmin2, dn, dn1, dn2, tau,
           ttype, g )
```

```
call dlazq4( i0, n0, z, pp, n0in, dmin, dmin1, dmin2, dn, dn1, dn2, tau,
           ttype, g )
```

Description

The routine computes an approximation τ to the smallest eigenvalue using values of d from the previous transform.

This routine is a thread safe version of ?lasq4 routine, which passes g through the argument list in place of declaring g in a `SAVE` statement.

Input Parameters

<i>i0</i>	INTEGER. First index.
<i>n0</i>	INTEGER. Last index.
<i>z</i>	REAL for slazq4 DOUBLE PRECISION for dlazq4. Array, DIMENSION (4*n). <i>z</i> holds the qd array.
<i>pp</i>	INTEGER. <i>pp</i> =0 for ping, <i>pp</i> =1 for pong.
<i>n0in</i>	INTEGER. The value of <i>n0</i> at start of eigtest.
<i>dmin</i>	REAL for slazq4 DOUBLE PRECISION for dlazq4. Minimum value of d .
<i>dmin1</i>	REAL for slazq4 DOUBLE PRECISION for dlazq4.

<i>dmin2</i>	Minimum value of d, excluding $d(n0)$. REAL for slazq4 DOUBLE PRECISION for dlazq4. Minimum value of d, excluding $d(n0)$ and $d(n0-1)$.
<i>dn</i>	REAL for slazq4 DOUBLE PRECISION for dlazq4. Contains $d(n)$.
<i>dn1</i>	REAL for slazq4 DOUBLE PRECISION for dlazq4. Contains $d(n-1)$.
<i>dn2</i>	REAL for slazq4 DOUBLE PRECISION for dlazq4. Contains $d(n-2)$.
<i>g</i>	REAL for slazq4 DOUBLE PRECISION for dlazq4. Shift coefficient.

Output Parameters

<i>tau</i>	REAL for slazq4 DOUBLE PRECISION for dlazq4. Shift value.
<i>ttype</i>	INTEGER. Shift type.
<i>g</i>	Shift coefficient.

?org2l/?ung2l

Generates all or part of the orthogonal/unitary matrix Q from a QL factorization determined by ?geqlf (unblocked algorithm).

Syntax

```
call sorg2l( m, n, k, a, lda, tau, work, info )
call dorg2l( m, n, k, a, lda, tau, work, info )
call cung2l( m, n, k, a, lda, tau, work, info )
call zung2l( m, n, k, a, lda, tau, work, info )
```

Description

The routine ?org2l/?ung2l generates an m -by- n real/complex matrix Q with orthonormal columns, which is defined as the last n columns of a product of k elementary reflectors of order m :

$Q = H(k) * \dots * H(2) * H(1)$ as returned by ?geqlf.

Input Parameters

m	INTEGER. The number of rows of the matrix Q . $m \geq 0$.
n	INTEGER. The number of columns of the matrix Q . $m \geq n \geq 0$.
k	INTEGER. The number of elementary reflectors whose product defines the matrix Q . $n \geq k \geq 0$.
a	REAL for sorg2l DOUBLE PRECISION for dorg2l COMPLEX for cung2l COMPLEX*16 for zung2l. Array, DIMENSION (lda,n). On entry, the $(n-k+i)$ -th column must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by ?geqlf in the last k columns of its array argument A.

lda INTEGER. The first dimension of the array *a*. $lda \geq \max(1, m)$.

tau REAL for sorg2l
DOUBLE PRECISION for dorg2l
COMPLEX for cung2l
COMPLEX*16 for zung2l.
Array, DIMENSION (*k*).
tau(*i*) must contain the scalar factor of the elementary reflector $H(i)$, as returned by ?geqlf.

work REAL for sorg2l
DOUBLE PRECISION for dorg2l
COMPLEX for cung2l
COMPLEX*16 for zung2l.
Workspace array, DIMENSION (*n*).

Output Parameters

a On exit, the m -by- n matrix Q .

info INTEGER.
= 0: successful exit
< 0: if *info* = -*i*, the *i*-th argument has an illegal value

?org2r/?ung2r

Generates all or part of the orthogonal/unitary matrix Q from a QR factorization determined by ?geqrf (unblocked algorithm).

Syntax

```
call sorg2r( m, n, k, a, lda, tau, work, info )
call dorg2r( m, n, k, a, lda, tau, work, info )
call cung2r( m, n, k, a, lda, tau, work, info )
call zung2r( m, n, k, a, lda, tau, work, info )
```


Description

The routine `?org2r/?ung2r` generates an m -by- n real/complex matrix Q with orthonormal columns, which is defined as the first n columns of a product of k elementary reflectors of order m

$$Q = H(1) * H(2) * \dots * H(k)$$

as returned by `?geqrf`.

Input Parameters

m	INTEGER. The number of rows of the matrix Q . $m \geq 0$.
n	INTEGER. The number of columns of the matrix Q . $m \geq n \geq 0$.
k	INTEGER. The number of elementary reflectors whose product defines the matrix Q . $n \geq k \geq 0$.
a	REAL for <code>sorg2r</code> DOUBLE PRECISION for <code>dorg2r</code> COMPLEX for <code>cung2r</code> COMPLEX*16 for <code>zung2r</code> . Array, DIMENSION (lda, n). On entry, the i -th column must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by <code>?geqrf</code> in the first k columns of its array argument a .
lda	INTEGER. The first DIMENSION of the array a . $lda \geq \max(1, m)$.
τ	REAL for <code>sorg2r</code> DOUBLE PRECISION for <code>dorg2r</code> COMPLEX for <code>cung2r</code> COMPLEX*16 for <code>zung2r</code> . Array, DIMENSION (k). $\tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by <code>?geqrf</code> .
$work$	REAL for <code>sorg2r</code> DOUBLE PRECISION for <code>dorg2r</code>

COMPLEX for `cung2r`
 COMPLEX*16 for `zung2r`.
 Workspace array, DIMENSION (n).

Output Parameters

`a` On exit, the m -by- n matrix Q .
`info` INTEGER.
 = 0: successful exit
 < 0: if `info` = $-i$, the i -th argument has an illegal value

?orgl2/?ungl2

Generates all or part of the orthogonal/unitary matrix Q from an LQ factorization determined by ?gelqf (unblocked algorithm).

Syntax

```
call sorgl2( m, n, k, a, lda, tau, work, info )
call dorgl2( m, n, k, a, lda, tau, work, info )
call cungl2( m, n, k, a, lda, tau, work, info )
call zungl2( m, n, k, a, lda, tau, work, info )
```

Description

The routine ?orgl2/?ungl2 generates a m -by- n real/complex matrix Q with orthonormal rows, which is defined as the first m rows of a product of k elementary reflectors of order n

$Q = H(k) * \dots * H(2) * H(1)$ for real flavors, or $Q = (H(k))^H * \dots * (H(2))^H * (H(1))^H$ for complex flavors as returned by ?gelqf.

Input Parameters

`m` INTEGER. The number of rows of the matrix Q . $m \geq 0$.
`n` INTEGER. The number of columns of the matrix Q . $n \geq m$.
`k` INTEGER. The number of elementary reflectors whose product defines the matrix Q . $m \geq k \geq 0$.

<i>a</i>	REAL for sorgl2 DOUBLE PRECISION for dorgl2 COMPLEX for cungl2 COMPLEX*16 for zungl2. Array, DIMENSION (<i>lda</i>, <i>n</i>). On entry, the <i>i</i>-th row must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by ?gelqf in the first <i>k</i> rows of its array argument <i>a</i>.
<i>lda</i>	INTEGER. The first dimension of the array <i>a</i>. $lda \geq \max(1, m)$.
<i>tau</i>	REAL for sorgl2 DOUBLE PRECISION for dorgl2 COMPLEX for cungl2 COMPLEX*16 for zungl2. Array, DIMENSION (<i>k</i>). <i>tau</i>(<i>i</i>) must contain the scalar factor of the elementary reflector $H(i)$, as returned by ?gelqf.
<i>work</i>	REAL for sorgl2 DOUBLE PRECISION for dorgl2 COMPLEX for cungl2 COMPLEX*16 for zungl2. Workspace array, DIMENSION (<i>m</i>).

Output Parameters

<i>a</i>	On exit, the <i>m</i> -by- <i>n</i> matrix Q .
<i>info</i>	INTEGER. = 0: successful exit < 0: if <i>info</i> = -<i>i</i>, the <i>i</i>-th argument has an illegal value.

?orgr2/?ungr2

Generates all or part of the orthogonal/unitary matrix Q from an RQ factorization determined by ?gerqf (unblocked algorithm).

Syntax

```
call sorgr2( m, n, k, a, lda, tau, work, info )
call dorgr2( m, n, k, a, lda, tau, work, info )
call cungr2( m, n, k, a, lda, tau, work, info )
call zungr2( m, n, k, a, lda, tau, work, info )
```

Description

The routine ?orgr2/?ungr2 generates an m -by- n real matrix Q with orthonormal rows, which is defined as the last m rows of a product of k elementary reflectors of order n

$Q = H(1) * H(2) * \dots * H(k)$ for real flavors, or $Q = (H(1))^H * (H(2))^H * \dots * (H(k))^H$ for complex flavors as returned by ?gerqf.

Input Parameters

m	INTEGER. The number of rows of the matrix Q . $m \geq 0$.
n	INTEGER. The number of columns of the matrix Q . $n \geq m$
k	INTEGER. The number of elementary reflectors whose product defines the matrix Q . $m \geq k \geq 0$.
a	REAL for sorgr2 DOUBLE PRECISION for dorgr2 COMPLEX for cungr2 COMPLEX*16 for zungr2. Array, DIMENSION (lda, n). On entry, the ($m-k+1$)-th row must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by ?gerqf in the last k rows of its array argument a .

lda INTEGER. The first dimension of the array *a*. $lda \geq \max(1, m)$.

tau REAL for `sorghr2`
 DOUBLE PRECISION for `dorghr2`
 COMPLEX for `cungr2`
 COMPLEX*16 for `zungr2`.
 Array, `DIMENSION (k).tau(i)` must contain the scalar factor of the elementary reflector $H(i)$, as returned by `?gerqf`.

work REAL for `sorghr2`
 DOUBLE PRECISION for `dorghr2`
 COMPLEX for `cungr2`
 COMPLEX*16 for `zungr2`.
 Workspace array, `DIMENSION (m)`.

Output Parameters

a On exit, the m -by- n matrix Q .

info INTEGER.
 = 0: successful exit
 < 0: if *info* = $-i$, the i -th argument has an illegal value

?orm2l/?unm2l

Multiplies a general matrix by the orthogonal/unitary matrix from a QL factorization determined by ?geqlf (unblocked algorithm).

Syntax

```
call sorm2l( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call dorm2l( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call cunm2l( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call zunm2l( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
```

Description

The routine `?orm2l/?unm2l` overwrites the general real/complex m -by- n matrix C with Q^*C if *side* = 'L' and *trans* = 'N', or

$Q^T * C / Q^H * C$ if *side* = 'L' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors), or

$C * Q$ if *side* = 'R' and *trans* = 'N', or

$C * Q^T / C * Q^H$ if *side* = 'R' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors).

Here Q is a real orthogonal or complex unitary matrix defined as the product of k elementary reflectors

$Q = H(k) * \dots * H(2) * H(1)$ as returned by `?geqlf`.

Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

Input Parameters

<i>side</i>	CHARACTER*1. = 'L': apply Q or Q^T / Q^H from the left = 'R': apply Q or Q^T / Q^H from the right
<i>trans</i>	CHARACTER*1. = 'N': apply Q (no transpose) = 'T': apply Q^T (transpose, for real flavors) = 'C': apply Q^H (conjugate transpose, for complex flavors)
<i>m</i>	INTEGER. The number of rows of the matrix C . $m \geq 0$.
<i>n</i>	INTEGER. The number of columns of the matrix C . $n \geq 0$.
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q . If <i>side</i> = 'L', $m \geq k \geq 0$; if <i>side</i> = 'R', $n \geq k \geq 0$.
<i>a</i>	REAL for <code>sorm21</code> DOUBLE PRECISION for <code>dorm21</code> COMPLEX for <code>cunm21</code> COMPLEX*16 for <code>zunm21</code> . Array, DIMENSION (<i>lda</i> , <i>k</i>). The i -th column must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by <code>?geqlf</code> in the last k columns of its array argument <i>a</i> . The array <i>a</i> is modified by the routine but restored on exit.

<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i>. If <i>side</i> = 'L', $lda \geq \max(1, m)$ if <i>side</i> = 'R', $lda \geq \max(1, n)$.
<i>tau</i>	REAL for sorm2l DOUBLE PRECISION for dorm2l COMPLEX for cunm2l COMPLEX*16 for zunm2l. Array, DIMENSION (<i>k</i>). <i>tau</i>(<i>i</i>) must contain the scalar factor of the elementary reflector $H(i)$, as returned by ?geqlf.
<i>c</i>	REAL for sorm2l DOUBLE PRECISION for dorm2l COMPLEX for cunm2l COMPLEX*16 for zunm2l. Array, DIMENSION (<i>ldc</i>, <i>n</i>). On entry, the <i>m</i>-by-<i>n</i> matrix <i>C</i>.
<i>ldc</i>	INTEGER. The leading dimension of the array <i>c</i>. $ldc \geq \max(1, m)$.
<i>work</i>	REAL for sorm2l DOUBLE PRECISION for dorm2l COMPLEX for cunm2l COMPLEX*16 for zunm2l. Workspace array, DIMENSION: (<i>n</i>) if <i>side</i> = 'L', (<i>m</i>) if <i>side</i> = 'R'.

Output Parameters

<i>c</i>	On exit, <i>c</i> is overwritten by Q^*C or Q^T*C / Q^H*C, or $C*Q$, or $C*Q^T / C*Q^H$.
<i>info</i>	INTEGER. = 0: successful exit < 0: if <i>info</i> = -<i>i</i>, the <i>i</i>-th argument had an illegal value

?orm2r/?unm2r

Multiplies a general matrix by the orthogonal/unitary matrix from a QR factorization determined by ?geqrf (unblocked algorithm).

Syntax

```
call sorm2r( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call dorm2r( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call cunm2r( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call zunm2r( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
```

Description

The routine ?orm2r/?unm2r overwrites the general real/complex m -by- n matrix C with

Q^*C if $side = 'L'$ and $trans = 'N'$, or

$Q^T C / Q^H C$ if $side = 'L'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors), or

$C Q$ if $side = 'R'$ and $trans = 'N'$, or

$C Q^T / C Q^H$ if $side = 'R'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors).

Here Q is a real orthogonal or complex unitary matrix defined as the product of k elementary reflectors

$Q = H(1) * H(2) * \dots * H(k)$ as returned by ?geqrf.

Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

<i>side</i>	CHARACTER*1. = 'L': apply Q or Q^T / Q^H from the left = 'R': apply Q or Q^T / Q^H from the right
<i>trans</i>	CHARACTER*1. = 'N': apply Q (no transpose) = 'T': apply Q^T (transpose, for real flavors) = 'C': apply Q^H (conjugate transpose, for complex flavors)

<i>m</i>	INTEGER. The number of rows of the matrix <i>C</i> . $m \geq 0$.
<i>n</i>	INTEGER. The number of columns of the matrix <i>C</i> . $n \geq 0$.
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix <i>Q</i> . If <i>side</i> = 'L', $m \geq k \geq 0$; if <i>side</i> = 'R', $n \geq k \geq 0$.
<i>a</i>	REAL for sorm2r DOUBLE PRECISION for dorm2r COMPLEX for cunm2r COMPLEX*16 for zunm2r. Array, DIMENSION (<i>lda</i> , <i>k</i>). The <i>i</i> -th column must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by ?geqrf in the first <i>k</i> columns of its array argument <i>a</i> . The array <i>a</i> is modified by the routine but restored on exit.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . If <i>side</i> = 'L', $lda \geq \max(1, m)$; if <i>side</i> = 'R', $lda \geq \max(1, n)$.
<i>tau</i>	REAL for sorm2r DOUBLE PRECISION for dorm2r COMPLEX for cunm2r COMPLEX*16 for zunm2r. Array, DIMENSION (<i>k</i>). <i>tau</i> (<i>i</i>) must contain the scalar factor of the elementary reflector $H(i)$, as returned by ?geqrf.
<i>c</i>	REAL for sorm2r DOUBLE PRECISION for dorm2r COMPLEX for cunm2r COMPLEX*16 for zunm2r. Array, DIMENSION (<i>ldc</i> , <i>n</i>). On entry, the <i>m</i> -by- <i>n</i> matrix <i>C</i> .
<i>ldc</i>	INTEGER. The leading dimension of the array <i>c</i> . $ldc \geq \max(1, m)$.
<i>work</i>	REAL for sorm2r

DOUBLE PRECISION for dorm2r
 COMPLEX for cunm2r
 COMPLEX*16 for zunm2r.
 Workspace array, DIMENSION
 (n) if *side* = 'L',
 (m) if *side* = 'R'.

Output Parameters

<i>c</i>	On exit, <i>c</i> is overwritten by Q^*C or Q^T*C / Q^H*C , or $C*Q$, or $C*Q^T / C*Q^H$.
<i>info</i>	INTEGER. = 0: successful exit < 0: if <i>info</i> = -i, the i-th argument had an illegal value

?orml2/?unml2

Multiplies a general matrix by the orthogonal/unitary matrix from a LQ factorization determined by ?gelqf (unblocked algorithm).

Syntax

```
call sorml2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call dorml2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call cunml2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call zunml2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
```

Description

The routine ?orml2/?unml2 overwrites the general real/complex *m*-by-*n* matrix *C* with

Q^*C if *side* = 'L' and *trans* = 'N', or

Q^T*C / Q^H*C if *side* = 'L' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors), or

$C*Q$ if *side* = 'R' and *trans* = 'N', or

$C*Q^T / C*Q^H$ if *side* = 'R' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors).

Here Q is a real orthogonal or complex unitary matrix defined as the product of k elementary reflectors

$Q = H(k) * \dots * H(2) * H(1)$ for real flavors, or $Q = (H(k))^H * \dots * (H(2))^H * (H(1))^H$ for complex flavors as returned by ?gelqf.

Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

<i>side</i>	CHARACTER*1. = 'L': apply Q or Q^T / Q^H from the left = 'R': apply Q or Q^T / Q^H from the right
<i>trans</i>	CHARACTER*1. = 'N': apply Q (no transpose) = 'T': apply Q^T (transpose, for real flavors) = 'C': apply Q^H (conjugate transpose, for complex flavors)
<i>m</i>	INTEGER. The number of rows of the matrix C . $m \geq 0$.
<i>n</i>	INTEGER. The number of columns of the matrix C . $n \geq 0$.
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q . If $side = 'L'$, $m \geq k \geq 0$; if $side = 'R'$, $n \geq k \geq 0$.
<i>a</i>	REAL for sorml2 DOUBLE PRECISION for dorml2 COMPLEX for cunml2 COMPLEX*16 for zunml2. Array, DIMENSION (<i>lda</i> , <i>m</i>) if $side = 'L'$, (<i>lda</i> , <i>n</i>) if $side = 'R'$ The i -th row must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by ?gelqf in the first k rows of its array argument a . The array a is modified by the routine but restored on exit.
<i>lda</i>	INTEGER. The leading dimension of the array a . $lda \geq \max(1, k)$.

<i>tau</i>	REAL for sorml2 DOUBLE PRECISION for dorml2 COMPLEX for cunml2 COMPLEX*16 for zunml2. Array, DIMENSION (<i>k</i>). <i>tau</i>(<i>i</i>) must contain the scalar factor of the elementary reflector $H(i)$, as returned by ?gelqf.
<i>c</i>	REAL for sorml2 DOUBLE PRECISION for dorml2 COMPLEX for cunml2 COMPLEX*16 for zunml2. Array, DIMENSION (<i>ldc</i>, <i>n</i>) On entry, the <i>m</i>-by-<i>n</i> matrix <i>C</i>.
<i>ldc</i>	INTEGER. The leading dimension of the array <i>c</i>. $ldc \geq \max(1, m)$.
<i>work</i>	REAL for sorml2 DOUBLE PRECISION for dorml2 COMPLEX for cunml2 COMPLEX*16 for zunml2. Workspace array, DIMENSION (<i>n</i>) if <i>side</i> = 'L', (<i>m</i>) if <i>side</i> = 'R'

Output Parameters

<i>c</i>	On exit, <i>c</i> is overwritten by Q^*C or $Q^T * C / Q^H * C$, or C^*Q, or C^*Q^T / C^*Q^H.
<i>info</i>	INTEGER. = 0: successful exit < 0: if <i>info</i> = -<i>i</i>, the <i>i</i>-th argument had an illegal value

?ormr2/?unmr2

Multiplies a general matrix by the orthogonal/unitary matrix from a RQ factorization determined by ?gerqf (unblocked algorithm).

Syntax

```
call sormr2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call dormr2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call cunmr2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call zunmr2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
```

Description

The routine ?ormr2/?unmr2 overwrites the general real/complex m -by- n matrix C with

Q^*C if $side = 'L'$ and $trans = 'N'$, or

$Q^T * C / Q^H * C$ if $side = 'L'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors), or

$C * Q$ if $side = 'R'$ and $trans = 'N'$, or

$C * Q^T / C * Q^H$ if $side = 'R'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors).

Here Q is a real orthogonal or complex unitary matrix defined as the product of k elementary reflectors

$Q = H(1) * H(2) * \dots * H(k)$ for real flavors, or $Q = (H(1))^H * (H(2))^H * \dots * (H(k))^H$ as returned by ?gerqf.

Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

<i>side</i>	CHARACTER*1. = 'L': apply Q or Q^T / Q^H from the left = 'R': apply Q or Q^T / Q^H from the right
<i>trans</i>	CHARACTER*1. = 'N': apply Q (no transpose) = 'T': apply Q^T (transpose, for real flavors)

= 'C': apply Q^H (conjugate transpose, for complex flavors)

m INTEGER. The number of rows of the matrix *C*. $m \geq 0$.

n INTEGER. The number of columns of the matrix *C*. $n \geq 0$.

k INTEGER. The number of elementary reflectors whose product defines the matrix *Q*.
 If *side* = 'L', $m \geq k \geq 0$;
 if *side* = 'R', $n \geq k \geq 0$.

a REAL for sormr2
 DOUBLE PRECISION for dormr2
 COMPLEX for cunmr2
 COMPLEX*16 for zunmr2.
 Array, DIMENSION
 (*lda*, *m*) if *side* = 'L',
 (*lda*, *n*) if *side* = 'R'
 The *i*-th row must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by ?gerqf in the last *k* rows of its array argument *a*. The array *a* is modified by the routine but restored on exit.

lda INTEGER.
 The leading dimension of the array *a*. $lda \geq \max(1, k)$.

tau REAL for sormr2
 DOUBLE PRECISION for dormr2
 COMPLEX for cunmr2
 COMPLEX*16 for zunmr2.
 Array, DIMENSION (*k*).
tau(*i*) must contain the scalar factor of the elementary reflector $H(i)$, as returned by ?gerqf.

c REAL for sormr2
 DOUBLE PRECISION for dormr2
 COMPLEX for cunmr2
 COMPLEX*16 for zunmr2.
 Array, DIMENSION (*ldc*, *n*).
 On entry, the *m*-by-*n* matrix *C*.

ldc INTEGER. The leading dimension of the array *c*. $ldc \geq \max(1, m)$.

work REAL for sormr2
 DOUBLE PRECISION for dormr2
 COMPLEX for cunmr2
 COMPLEX*16 for zunmr2.
 Workspace array, DIMENSION
 (*n*) if *side* = 'L',
 (*m*) if *side* = 'R'

Output Parameters

c On exit, *c* is overwritten by Q^*C or $Q^T * C$ / $Q^H * C$, or $C * Q$, or $C * Q^T$ / $C * Q^H$.

info INTEGER.
 = 0: successful exit
 < 0: if *info* = -*i*, the *i*-th argument had an illegal value

?ormr3/?unmr3

Multiplies a general matrix by the orthogonal/unitary matrix from a RZ factorization determined by ?tzzrf (unblocked algorithm).

Syntax

```
call sormr3( side, trans, m, n, k, l, a, lda, tau, c, ldc, work, info )
call dormr3( side, trans, m, n, k, l, a, lda, tau, c, ldc, work, info )
call cunmr3( side, trans, m, n, k, l, a, lda, tau, c, ldc, work, info )
call zunmr3( side, trans, m, n, k, l, a, lda, tau, c, ldc, work, info )
```

Description

The routine ?ormr3/?unmr3 overwrites the general real/complex *m*-by-*n* matrix *C* with

Q^*C if *side* = 'L' and *trans* = 'N', or

$Q^T * C$ / $Q^H * C$ if *side* = 'L' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors), or

$C*Q$ if $side = 'R'$ and $trans = 'N'$, or

$C*Q^T / C*Q^H$ if $side = 'R'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors).

Here Q is a real orthogonal or complex unitary matrix defined as the product of k elementary reflectors

$Q = H(1)*H(2)*\dots*H(k)$ as returned by ?tzrzf.

Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

<i>side</i>	CHARACTER*1. = 'L': apply Q or Q^T / Q^H from the left = 'R': apply Q or Q^T / Q^H from the right
<i>trans</i>	CHARACTER*1. = 'N': apply Q (no transpose) = 'T': apply Q^T (transpose, for real flavors) = 'C': apply Q^H (conjugate transpose, for complex flavors)
<i>m</i>	INTEGER. The number of rows of the matrix C . $m \geq 0$.
<i>n</i>	INTEGER. The number of columns of the matrix C . $n \geq 0$.
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q . If $side = 'L'$, $m \geq k \geq 0$; if $side = 'R'$, $n \geq k \geq 0$.
<i>l</i>	INTEGER. The number of columns of the matrix A containing the meaningful part of the Householder reflectors. If $side = 'L'$, $m \geq l \geq 0$, if $side = 'R'$, $n \geq l \geq 0$.
<i>a</i>	REAL for sormr3 DOUBLE PRECISION for dormr3 COMPLEX for cunmr3 COMPLEX*16 for zunmr3. Array, DIMENSION (lda, m) if $side = 'L'$, (lda, n) if $side = 'R'$

The i -th row must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by ?tzzrzf in the last k rows of its array argument a . The array a is modified by the routine but restored on exit.

<i>lda</i>	INTEGER. The leading dimension of the array a . $lda \geq \max(1, k)$.
<i>tau</i>	REAL for sormr3 DOUBLE PRECISION for dormr3 COMPLEX for cunmr3 COMPLEX*16 for zunmr3. Array, DIMENSION (k). $\tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by ?tzzrzf.
<i>c</i>	REAL for sormr3 DOUBLE PRECISION for dormr3 COMPLEX for cunmr3 COMPLEX*16 for zunmr3. Array, DIMENSION (ldc, n). On entry, the m -by- n matrix C .
<i>ldc</i>	INTEGER. The leading dimension of the array c . $ldc \geq \max(1, m)$.
<i>work</i>	REAL for sormr3 DOUBLE PRECISION for dormr3 COMPLEX for cunmr3 COMPLEX*16 for zunmr3. Workspace array, DIMENSION (n) if $side = 'L'$, (m) if $side = 'R'$.

Output Parameters

<i>c</i>	On exit, c is overwritten by Q^*C or Q^T*C / Q^H*C , or $C*Q$, or $C*Q^T / C*Q^H$.
<i>info</i>	INTEGER. = 0: successful exit < 0: if $info = -i$, the i -th argument had an illegal value

?pbtf2

Computes the Cholesky factorization of a symmetric/ Hermitian positive-definite band matrix (unblocked algorithm).

Syntax

```
call spbtf2( uplo, n, kd, ab, ldab, info )
call dpbtf2( uplo, n, kd, ab, ldab, info )
call cpbtf2( uplo, n, kd, ab, ldab, info )
call zpbtf2( uplo, n, kd, ab, ldab, info )
```

Description

The routine computes the Cholesky factorization of a real symmetric or complex Hermitian positive definite band matrix A .

The factorization has the form

$A = U^T * U$ for real flavors, $A = U^H * U$ for complex flavors if $uplo = 'U'$, or

$A = L * L^T$ for real flavors, $A = L * L^H$ for complex flavors if $uplo = 'L'$,

where U is an upper triangular matrix, and L is lower triangular. This is the unblocked version of the algorithm, calling [BLAS Level 2 Routines](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix A is stored: = 'U': upper triangular = 'L': lower triangular
<i>n</i>	INTEGER. The order of the matrix A . $n \geq 0$.
<i>kd</i>	INTEGER. The number of super-diagonals of the matrix A if $uplo = 'U'$, or the number of sub-diagonals if $uplo = 'L'$. $kd \geq 0$.
<i>ab</i>	REAL for spbtf2

DOUBLE PRECISION for dpbtf2

COMPLEX for cpbtf2

COMPLEX*16 for zpbtf2.

Array, DIMENSION (*ldab*, *n*).

On entry, the upper or lower triangle of the symmetric/Hermitian band matrix *A*, stored in the first *kd*+1 rows of the array. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:

if *uplo* = 'U', $ab(kd+1+i-j, j) = A(i, j)$ for $\max(1, j-kd) \leq i \leq j$;

if *uplo* = 'L', $ab(1+i-j, j) = A(i, j)$ for $j \leq i \leq \min(n, j+kd)$.

ldab

INTEGER. The leading dimension of the array *ab*. *ldab* $\geq kd+1$.

Output Parameters

ab

On exit, If *info* = 0, the triangular factor *U* or *L* from the Cholesky factorization $A=U^T*U$ ($A=U^H*U$), or $A=L*L^T$ ($A=L*L^H$) of the band matrix *A*, in the same storage format as *A*.

info

INTEGER.

= 0: successful exit

< 0: if *info* = -*k*, the *k*-th argument had an illegal value

> 0: if *info* = *k*, the leading minor of order *k* is not positive definite, and the factorization could not be completed.

?potf2

Computes the Cholesky factorization of a symmetric/Hermitian positive-definite matrix (unblocked algorithm).

Syntax

```
call spotf2( uplo, n, a, lda, info )
call dpotf2( uplo, n, a, lda, info )
call cpotf2( uplo, n, a, lda, info )
call zpotf2( uplo, n, a, lda, info )
```

Description

The routine ?potf2 computes the Cholesky factorization of a real symmetric or complex Hermitian positive definite matrix A . The factorization has the form

$A = U^T * U$ for real flavors, $A = U^H * U$ for complex flavors if $uplo = 'U'$, or

$A = L * L^T$ for real flavors, $A = L * L^H$ for complex flavors if $uplo = 'L'$,

where U is an upper triangular matrix, and L is lower triangular.

This is the unblocked version of the algorithm, calling [BLAS Level 2 Routines](#)

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix A is stored. = 'U': upper triangular = 'L': lower triangular
<i>n</i>	INTEGER. The order of the matrix A . $n \geq 0$.
<i>a</i>	REAL for spotf2 DOUBLE PRECISION or dpotf2 COMPLEX for cpotf2 COMPLEX*16 for zpotf2. Array, DIMENSION (lda, n). On entry, the symmetric/Hermitian matrix A .

If $uplo = 'U'$, the leading n -by- n upper triangular part of a contains the upper triangular part of the matrix A , and the strictly lower triangular part of a is not referenced.

If $uplo = 'L'$, the leading n -by- n lower triangular part of a contains the lower triangular part of the matrix A , and the strictly upper triangular part of a is not referenced.

lda INTEGER. The leading dimension of the array a .

$lda \geq \max(1, n)$.

Output Parameters

a On exit, If $info = 0$, the factor U or L from the Cholesky factorization $A = U^T * U$ ($A = U^H * U$), or $A = L * L^T$ ($A = L * L^H$).

$info$ INTEGER.

= 0: successful exit

< 0: if $info = -k$, the k -th argument had an illegal value

> 0: if $info = k$, the leading minor of order k is not positive definite, and the factorization could not be completed.

?ptts2

Solves a tridiagonal system of the form $A * X = B$ using the $L * D * L^H / L * D * L^H$ factorization computed by ?pttrf.

Syntax

```
call sptts2( n, nrhs, d, e, b, ldb )
call dptts2( n, nrhs, d, e, b, ldb )
call cptts2( iuplo, n, nrhs, d, e, b, ldb )
call zptts2( iuplo, n, nrhs, d, e, b, ldb )
```

Description

The routine ?ptts2 solves a tridiagonal system of the form

$$A * X = B$$

Real flavors `sptts2/dptts2` use the $L^*D^*L^T$ factorization of A computed by `spttrf/dpttrf`, and complex flavors `cptts2/zptts2` use the U^H*D*U or $L^*D^*L^H$ factorization of A computed by `cpttrf/zpttrf`.

D is a diagonal matrix specified in the vector d , U (or L) is a unit bidiagonal matrix whose superdiagonal (subdiagonal) is specified in the vector e , and X and B are n -by- $nrhs$ matrices.

Input Parameters

<i>iuplo</i>	INTEGER. Used with complex flavors only. Specifies the form of the factorization, and whether the vector e is the superdiagonal of the upper bidiagonal factor U or the subdiagonal of the lower bidiagonal factor L. = 1: $A = U^H*D*U$, e is the superdiagonal of U; = 0: $A = L^*D^*L^H$, e is the subdiagonal of L
<i>n</i>	INTEGER. The order of the tridiagonal matrix A. $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right hand sides, that is, the number of columns of the matrix B. $nrhs \geq 0$.
<i>d</i>	REAL for <code>sptts2/cptts2</code> DOUBLE PRECISION for <code>dptts2/zptts2</code>. Array, DIMENSION (n). The n diagonal elements of the diagonal matrix D from the factorization of A.
<i>e</i>	REAL for <code>sptts2</code> DOUBLE PRECISION for <code>dptts2</code> COMPLEX for <code>cptts2</code> COMPLEX*16 for <code>zptts2</code>. Array, DIMENSION ($n-1$). Contains the ($n-1$) subdiagonal elements of the unit bidiagonal factor L from the $L^*D^*L^T$ (for real flavors) or $L^*D^*L^H$ (for complex flavors when $iuplo = 0$) factorization of A. For complex flavors when $iuplo = 1$, e contains the ($n-1$) superdiagonal elements of the unit bidiagonal factor U from the factorization $A = U^H*D*U$.
<i>B</i>	REAL for <code>sptts2/cptts2</code> DOUBLE PRECISION for <code>dptts2/zptts2</code>.

Array, DIMENSION (ldb , $nrhs$).

On entry, the right hand side vectors B for the system of linear equations.

ldb

INTEGER. The leading dimension of the array B . $ldb \geq \max(1, n)$.

Output Parameters

b

On exit, the solution vectors, x .

?rscl

Multiplies a vector by the reciprocal of a real scalar.

Syntax

```
call srscl( n, sa, sx, incx )
```

```
call drscl( n, sa, sx, incx )
```

```
call csrscl( n, sa, sx, incx )
```

```
call zdrscl( n, sa, sx, incx )
```

Description

The routine ?rscl multiplies an n -element real/complex vector x by the real scalar $1/a$. This is done without overflow or underflow as long as the final result x/a does not overflow or underflow.

Input Parameters

n

INTEGER. The number of components of the vector x .

sa

REAL for srscl/csrscl

DOUBLE PRECISION for drscl/zdrscl.

The scalar a which is used to divide each component of the vector x . sa must be ≥ 0 , or the subroutine will divide by zero.

sx

REAL for srscl

DOUBLE PRECISION for drscl

COMPLEX for csrscl

COMPLEX*16 for `zdrsl`.
 Array, DIMENSION $(1+(n-1) * |incx|)$.
 The n -element vector x .
`incx` INTEGER. The increment between successive values of the vector sx .
 If $incx > 0$, $sx(1)=x(1)$, and $sx(1+(i-1)*incx)=x(i)$, $1 < i \leq n$.

Output Parameters

`sx` On exit, the result x/a .

?sygs2/?hegs2

Reduces a symmetric/Hermitian definite generalized eigenproblem to standard form, using the factorization results obtained from ?potrf (unblocked algorithm).

Syntax

```
call ssygs2( itype, uplo, n, a, lda, b, ldb, info )
call dsygs2( itype, uplo, n, a, lda, b, ldb, info )
call chgs2( itype, uplo, n, a, lda, b, ldb, info )
call zhegs2( itype, uplo, n, a, lda, b, ldb, info )
```

Description

The routine ?sygs2/?hegs2 reduces a real symmetric-definite or a complex Hermitian-definite generalized eigenproblem to standard form.

If $itype = 1$, the problem is

$$A*x = \lambda*B*x$$

and A is overwritten by $inv(U')*A*inv(U)$, or $inv(L)*A*inv(L')$.

If $itype = 2$ or 3 , the problem is

$$A*B*x = \lambda*x, \text{ or } B*A*x = \lambda*x,$$

and A is overwritten by U^*A^*U' or L'^*A^*L . Here $U'(L')$ is the transpose (conjugate transpose) of $U(L)$.

B must be previously factorized as U'^*U or L^*L' by ?potrf.

Input Parameters

itype INTEGER.
 = 1: compute $\text{inv}(U')^*A^*\text{inv}(U)$, or $\text{inv}(L)^*A^*\text{inv}(L')$;
 = 2 or 3: compute U^*A^*U' , or L'^*A^*L .

uplo CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix A is stored, and how B has been factorized.
 = 'U': upper triangular
 = 'L': lower triangular

n INTEGER. The order of the matrices A and B . $n \geq 0$.

a REAL for ssygs2
 DOUBLE PRECISION for dsygs2
 COMPLEX for chegs2
 COMPLEX*16 for zhegs2.
 Array, DIMENSION (lda, n).
 On entry, the symmetric/Hermitian matrix A .
 If *uplo* = 'U', the leading n -by- n upper triangular part of a contains the upper triangular part of the matrix A , and the strictly lower triangular part of a is not referenced.
 If *uplo* = 'L', the leading n -by- n lower triangular part of a contains the lower triangular part of the matrix A , and the strictly upper triangular part of a is not referenced.

lda INTEGER.
 The leading dimension of the array a . $lda \geq \max(1, n)$.

b REAL for ssygs2
 DOUBLE PRECISION for dsygs2
 COMPLEX for chegs2
 COMPLEX*16 for zhegs2.
 Array, DIMENSION (ldb, n).
 The triangular factor from the Cholesky factorization of B as returned by ?potrf.

ldb INTEGER. The leading dimension of the array *b*. $ldb \geq \max(1, n)$.

Output Parameters

a On exit, If *info* = 0, the transformed matrix, stored in the same format as *A*.

info INTEGER.
 = 0: successful exit.
 < 0: if *info* = -*i*, the *i*-th argument had an illegal value.

?sytd2/?hetd2

Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation(unblocked algorithm).

Syntax

```
call ssytd2( uplo, n, a, lda, d, e, tau, info )
call dsytd2( uplo, n, a, lda, d, e, tau, info )
call chetd2( uplo, n, a, lda, d, e, tau, info )
call zhetd2( uplo, n, a, lda, d, e, tau, info )
```

Description

The routine ?sytd2/?hetd2 reduces a real symmetric/complex Hermitian matrix *A* to real symmetric tridiagonal form *T* by an orthogonal/unitary similarity transformation: $Q^T A Q = T$ ($Q^H A Q = T$).

Input Parameters

uplo CHARACTER*1.
 Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix *A* is stored:
 = 'U': upper triangular
 = 'L': lower triangular

n INTEGER. The order of the matrix *A*. $n \geq 0$.

a REAL for ssytd2
 DOUBLE PRECISION for dsytd2
 COMPLEX for chetd2
 COMPLEX*16 for zhetd2.
 Array, DIMENSION (*lda*, *n*).
 On entry, the symmetric/Hermitian matrix *A*.
 If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced.
 If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda INTEGER. The leading dimension of the array *a*. $lda \geq \max(1, n)$.

Output Parameters

a On exit, if *uplo* = 'U', the diagonal and first superdiagonal of *a* are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements above the first superdiagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors;
 if *uplo* = 'L', the diagonal and first subdiagonal of *a* are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements below the first subdiagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors.

d REAL for ssytd2/chetd2
 DOUBLE PRECISION for dsytd2/zhetd2.
 Array, DIMENSION (*n*).
 The diagonal elements of the tridiagonal matrix *T*:
 $d(i) = a(i, i)$.

e REAL for ssytd2/chetd2
 DOUBLE PRECISION for dsytd2/zhetd2.
 Array, DIMENSION (*n*-1).
 The off-diagonal elements of the tridiagonal matrix *T*:
 $e(i) = a(i, i+1)$ if *uplo* = 'U',

	$e(i) = a(i+1,i)$ if <code>uplo = 'L'</code> .
<code>tau</code>	REAL for ssytd2 DOUBLE PRECISION for dsytd2 COMPLEX for chetd2 COMPLEX*16 for zhetd2. Array, DIMENSION (<i>n</i>). The first <i>n</i>-1 elements contain scalar factors of the elementary reflectors. <code>tau(n)</code> is used as workspace.
<code>info</code>	INTEGER. = 0: successful exit < 0: if <code>info = -i</code>, the <i>i</i>-th argument had an illegal value.

?sytf2

Computes the factorization of a real/complex symmetric indefinite matrix, using the diagonal pivoting method (unblocked algorithm).

Syntax

```
call ssytf2( uplo, n, a, lda, ipiv, info )
call dsytf2( uplo, n, a, lda, ipiv, info )
call csytf2( uplo, n, a, lda, ipiv, info )
call zsytf2( uplo, n, a, lda, ipiv, info )
```

Description

The routine `?sytf2` computes the factorization of a real/complex symmetric matrix *A* using the Bunch-Kaufman diagonal pivoting method:

$$A = U^* D^* U^T (A = U^* D^* U^H), \text{ or } A = L^* D^* L^T (A = L^* D^* L^H),$$

where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, and *D* is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the unblocked version of the algorithm, calling [BLAS Level 2 Routines](#).

Input Parameters

<code>uplo</code>	CHARACTER*1.

Specifies whether the upper or lower triangular part of the symmetric matrix A is stored
 = 'U': upper triangular
 = 'L': lower triangular

n INTEGER. The order of the matrix A . $n \geq 0$.

a REAL for ssytf2
 DOUBLE PRECISION for dsytf2
 COMPLEX for csytf2
 COMPLEX*16 for zsytf2.
 Array, DIMENSION (lda, n).
 On entry, the symmetric matrix A .
 If *uplo* = 'U', the leading n -by- n upper triangular part of a contains the upper triangular part of the matrix A , and the strictly lower triangular part of a is not referenced.
 If *uplo* = 'L', the leading n -by- n lower triangular part of a contains the lower triangular part of the matrix A , and the strictly upper triangular part of a is not referenced.

lda INTEGER.
 The leading dimension of the array a . $lda \geq \max(1, n)$.

Output Parameters

a On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L .

ipiv INTEGER.
 Array, DIMENSION (n).
 Details of the interchanges and the block structure of D
 If $ipiv(k) > 0$, then rows and columns k and $ipiv(k)$ are interchanged and $D(k,k)$ is a 1-by-1 diagonal block.
 If *uplo* = 'U' and $ipiv(k) = ipiv(k-1) < 0$, then rows and columns $k-1$ and $-ipiv(k)$ are interchanged and $D(k,k)$ is a 2-by-2 diagonal block.
 If *uplo* = 'L' and $ipiv(k) = ipiv(k+1) < 0$, then rows and columns $k+1$ and $-ipiv(k)$ were interchanged and $D(k:k+1, k:k+1)$ is a 2-by-2 diagonal block.

info INTEGER.
 = 0: successful exit

< 0: if *info* = -*k*, the *k*-th argument has an illegal value
 > 0: if *info* = *k*, *D*(*k*,*k*) is exactly zero. The factorization are completed, but the block diagonal matrix *D* is exactly singular, and division by zero will occur if it is used to solve a system of equations.

zhetf2

Computes the factorization of a complex Hermitian matrix, using the diagonal pivoting method (unblocked algorithm).

Syntax

```
call chetf2( uplo, n, a, lda, ipiv, info )
call zhetf2( uplo, n, a, lda, ipiv, info )
```

Description

The routine computes the factorization of a complex Hermitian matrix *A* using the Bunch-Kaufman diagonal pivoting method:

$$A = U^* D^* U' \text{ or } A = L^* D^* L'$$

where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, *U'* is the conjugate transpose of *U*, and *D* is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the unblocked version of the algorithm, calling [BLAS Level 2 Routines](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian matrix <i>A</i> is stored: = 'U': Upper triangular = 'L': Lower triangular
<i>n</i>	INTEGER. The order of the matrix <i>A</i> . $n \geq 0$.
<i>A</i>	COMPLEX for chetf2 COMPLEX*16 for zhetf2. Array, DIMENSION (<i>lda</i> , <i>n</i>).

On entry, the Hermitian matrix A .

If $uplo = 'U'$, the leading n -by- n upper triangular part of A contains the upper triangular part of the matrix A , and the strictly lower triangular part of A is not referenced.

If $uplo = 'L'$, the leading n -by- n lower triangular part of A contains the lower triangular part of the matrix A , and the strictly upper triangular part of A is not referenced.

lda

INTEGER. The leading dimension of the array a . $lda \geq \max(1, n)$.

Output Parameters

a

On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L .

ipiv

INTEGER. Array, DIMENSION (n).

Details of the interchanges and the block structure of D

If $ipiv(k) > 0$, then rows and columns k and $ipiv(k)$ were interchanged and $D(k, k)$ is a 1-by-1 diagonal block.

If $uplo = 'U'$ and $ipiv(k) = ipiv(k-1) < 0$, then rows and columns $k-1$ and $-ipiv(k)$ were interchanged and $D(k-1:k, k-1:k)$ is a 2-by-2 diagonal block.

If $uplo = 'L'$ and $ipiv(k) = ipiv(k+1) < 0$, then rows and columns $k+1$ and $-ipiv(k)$ were interchanged and $D(k:k+1, k:k+1)$ is a 2-by-2 diagonal block.

info

INTEGER.

= 0: successful exit

< 0: if $info = -k$, the k -th argument had an illegal value

> 0: if $info = k$, $D(k, k)$ is exactly zero. The factorization has been completed, but the block diagonal matrix D is exactly singular, and division by zero will occur if it is used to solve a system of equations.

?tgex2

Swaps adjacent diagonal blocks in an upper (quasi) triangular matrix pair by an orthogonal/unitary equivalence transformation.

Syntax

```
call stgex2( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, j1, n1, n2,
work, lwork, info )

call dtgex2( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, j1, n1, n2,
work, lwork, info )

call ctgex2( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, j1, info )
call ztgex2( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, j1, info )
```

Description

The real routines `stgex2/dtgex2` swap adjacent diagonal blocks (A_{11} , B_{11}) and (A_{22} , B_{22}) of size 1-by-1 or 2-by-2 in an upper (quasi) triangular matrix pair (A , B) by an orthogonal equivalence transformation. (A , B) must be in generalized real Schur canonical form (as returned by `sgges/dgges`), that is, A is block upper triangular with 1-by-1 and 2-by-2 diagonal blocks. B is upper triangular.

The complex routines `ctgex2/ztgex2` swap adjacent diagonal 1-by-1 blocks (A_{11} , B_{11}) and (A_{22} , B_{22}) in an upper triangular matrix pair (A , B) by an unitary equivalence transformation.

(A , B) must be in generalized Schur canonical form, that is, A and B are both upper triangular.

All routines optionally update the matrices Q and Z of generalized Schur vectors:

$$Q(\text{in}) * A(\text{in}) * Z(\text{in})' = Q(\text{out}) * A(\text{out}) * Z(\text{out})'$$

$Q(\text{in}) * B(\text{in}) * Z(\text{in})' = Q(\text{out}) * B(\text{out}) * Z(\text{out})'$, where Z' is the transpose (conjugate transpose) of Z .

Input Parameters

<code>wantq</code>	LOGICAL. If <code>wantq = .TRUE.</code> : update the left transformation matrix Q ; If <code>wantq = .FALSE.</code> : do not update Q .
<code>wantz</code>	LOGICAL.

If *wantz* = .TRUE. : update the right transformation matrix *Z*;
 If *wantz* = .FALSE. : do not update *Z*.

n INTEGER. The order of the matrices *A* and *B*. $n \geq 0$.

a, b REAL for stgex2 DOUBLE PRECISION for dtgex2
 COMPLEX for ctgex2
 COMPLEX*16 for ztgex2.
Arrays, DIMENSION (*lda*, *n*) and (*ldb*, *n*), respectively.
On entry, the matrices *A* and *B* in the pair (*A*, *B*).

lda INTEGER. The leading dimension of the array *a*. $lda \geq \max(1, n)$.

ldb INTEGER. The leading dimension of the array *b*. $ldb \geq \max(1, n)$.

q, z REAL for stgex2 DOUBLE PRECISION for dtgex2
 COMPLEX for ctgex2
 COMPLEX*16 for ztgex2.
Arrays, DIMENSION (*ldq*, *n*) and (*ldz*, *n*), respectively.
On entry, if *wantq* = .TRUE., *q* contains the
orthogonal/unitary matrix *Q*, and if *wantz* = .TRUE., *z*
contains the orthogonal/unitary matrix *Z*.

ldq INTEGER. The leading dimension of the array *q*. $ldq \geq 1$.
 If *wantq* = .TRUE., $ldq \geq n$.

ldz INTEGER. The leading dimension of the array *z*. $ldz \geq 1$.
 If *wantz* = .TRUE., $ldz \geq n$.

j1 INTEGER.
The index to the first block (*A11*, *B11*). $1 \leq j1 \leq n$.

n1 INTEGER. Used with real flavors only. The order of the first
block (*A11*, *B11*). $n1 = 0, 1$ or 2 .

n2 INTEGER. Used with real flavors only. The order of the
second block (*A22*, *B22*). $n2 = 0, 1$ or 2 .

work REAL for stgex2
 DOUBLE PRECISION for dtgex2.

Workspace array, `DIMENSION (max(1, lwork))`. Used with real flavors only.

lwork

INTEGER. The dimension of the array *work*.

$lwork \geq \max(n * (n2 + n1), 2 * (n2 + n1)^2)$

Output Parameters

a

On exit, the updated matrix *A*.

B

On exit, the updated matrix *B*.

Q

On exit, the updated matrix *Q*.

Not referenced if *wantq* = .FALSE..

z

On exit, the updated matrix *Z*.

Not referenced if *wantz* = .FALSE..

info

INTEGER.

=0: Successful exit For *stgex2/dtgex2*: If *info* = 1, the transformed matrix (*A*, *B*) would be too far from generalized Schur form; the blocks are not swapped and (*A*, *B*) and (*Q*, *Z*) are unchanged. The problem of swapping is too ill-conditioned. If *info* = -16: *lwork* is too small.

Appropriate value for *lwork* is returned in *work*(1).

For *ctgex2/ztgex2*:

If *info* = 1, the transformed matrix pair (*A*, *B*) would be too far from generalized Schur form; the problem is ill-conditioned.

?tgsy2

Solves the generalized Sylvester equation (unblocked algorithm).

Syntax

```
call stgsy2( trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f,
            ldf, scale, rdsum, rdscal, iwork, pq, info )

call dtgsy2( trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f,
            ldf, scale, rdsum, rdscal, iwork, pq, info )

call ctgsy2( trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f,
            ldf, scale, rdsum, rdscal, iwork, pq, info )

call ztgsy2( trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f,
            ldf, scale, rdsum, rdscal, iwork, pq, info )
```

Description

The routine ?tgsy2 solves the generalized Sylvester equation:

$$A^*R - L^*B = \text{scale}^*C \quad (1)$$

$$D^*R - L^*E = \text{scale}^*F$$

using Level 1 and 2 BLAS, where R and L are unknown m -by- n matrices, (A, D) , (B, E) and (C, F) are given matrix pairs of size m -by- m , n -by- n and m -by- n , respectively. For stgsy2/dtgsy2, pairs (A, D) and (B, E) must be in generalized Schur canonical form, that is, A, B are upper quasi triangular and D, E are upper triangular. For ctgsy2/ztgsy2, matrices A, B, D and E are upper triangular (that is, (A, D) and (B, E) in generalized Schur form).

The solution (R, L) overwrites (C, F) .

$0 \leq \text{scale} \leq 1$ is an output scaling factor chosen to avoid overflow.

In matrix notation, solving equation (1) corresponds to solve

$$Z^*x = \text{scale}^*b$$

where Z is defined as

$$Z = \begin{bmatrix} \text{kron}(I_n, A) & -\text{kron}(B', I_n) \\ \text{kron}(I_n, D) & -\text{kron}(E', I_n) \end{bmatrix} \quad (2)$$

Here I_k is the identity matrix of size k and x' is the transpose (conjugate transpose) of x .
 $\text{kron}(X, Y)$ denotes the Kronecker product between the matrices X and Y .

If $trans = 'T'$, solve the transposed (conjugate transposed) system

$$Z' * y = scale * b$$

for y , which is equivalent to solve for R and L in

$$A' * R + D' * L = scale * C \quad (3)$$

$$R * B' + L * E' = scale * (-F)$$

This case is used to compute an estimate of $\text{Dif}[(A, D), (B, E)] = \text{sigma\_min}(Z)$ using reverse communication with [?lacon](#).

[?tgssy2](#) also (for $ijob \geq 1$) contributes to the computation in [?tgssyl](#) of an upper bound on the separation between two matrix pairs. Then the input (A, D) , (B, E) are sub-pencils of the matrix pair (two matrix pairs) in [?tgssyl](#). See [?tgssyl](#) for details.

Input Parameters

<i>trans</i>	CHARACTER*1. If $trans = 'N'$, solve the generalized Sylvester equation (1); If $trans = 'T'$: solve the 'transposed' system (3).
<i>ijob</i>	INTEGER. Specifies what kind of functionality is to be performed. If $ijob = 0$: solve (1) only. If $ijob = 1$: a contribution from this subsystem to a Frobenius norm-based estimate of the separation between two matrix pairs is computed (look ahead strategy is used); If $ijob = 2$: a contribution from this subsystem to a Frobenius norm-based estimate of the separation between two matrix pairs is computed (?gecon on sub-systems is used). Not referenced if $trans = 'T'$.

<i>m</i>	INTEGER. On entry, <i>m</i> specifies the order of <i>A</i> and <i>D</i> , and the row dimension of <i>C</i> , <i>F</i> , <i>R</i> and <i>L</i> .
<i>n</i>	INTEGER. On entry, <i>n</i> specifies the order of <i>B</i> and <i>E</i> , and the column dimension of <i>C</i> , <i>F</i> , <i>R</i> and <i>L</i> .
<i>a</i> , <i>b</i>	REAL for stgsy2 DOUBLE PRECISION for dtgsy2 COMPLEX for ctgsy2 COMPLEX*16 for ztgsy2. Arrays, DIMENSION (<i>lda</i> , <i>m</i>) and (<i>ldb</i> , <i>n</i>), respectively. On entry, <i>a</i> contains an upper (quasi) triangular matrix <i>A</i> , and <i>b</i> contains an upper (quasi) triangular matrix <i>B</i> .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $lda \geq \max(1, m)$.
<i>ldb</i>	INTEGER. The leading dimension of the array <i>b</i> . $ldb \geq \max(1, n)$.
<i>c</i> , <i>f</i>	REAL for stgsy2 DOUBLE PRECISION for dtgsy2 COMPLEX for ctgsy2 COMPLEX*16 for ztgsy2. Arrays, DIMENSION (<i>ldc</i> , <i>n</i>) and (<i>ldf</i> , <i>n</i>), respectively. On entry, <i>c</i> contains the right-hand-side of the first matrix equation in (1), and <i>f</i> contains the right-hand-side of the second matrix equation in (1).
<i>ldc</i>	INTEGER. The leading dimension of the array <i>c</i> . $ldc \geq \max(1, m)$.
<i>d</i> , <i>e</i>	REAL for stgsy2 DOUBLE PRECISION for dtgsy2 COMPLEX for ctgsy2 COMPLEX*16 for ztgsy2. Arrays, DIMENSION (<i>ldd</i> , <i>m</i>) and (<i>lde</i> , <i>n</i>), respectively. On entry, <i>d</i> contains an upper triangular matrix <i>D</i> , and <i>e</i> contains an upper triangular matrix <i>E</i> .
<i>ldd</i>	INTEGER. The leading dimension of the array <i>d</i> . $ldd \geq \max(1, m)$.

<i>lde</i>	INTEGER. The leading dimension of the array <i>e</i> . $lde \geq \max(1, n)$.
<i>ldf</i>	INTEGER. The leading dimension of the array <i>f</i> . $ldf \geq \max(1, m)$.
<i>rdsum</i>	REAL for stgsy2/ctgsy2 DOUBLE PRECISION for dtgsy2/ztgsy2. On entry, the sum of squares of computed contributions to the Dif-estimate under computation by ?tgsyL, where the scaling factor <i>rdscal</i> has been factored out.
<i>rdscal</i>	REAL for stgsy2/ctgsy2 DOUBLE PRECISION for dtgsy2/ztgsy2. On entry, scaling factor used to prevent overflow in <i>rdsum</i> .
<i>iwork</i>	INTEGER. Used with real flavors only. Workspace array, DIMENSION ($m+n+2$).

Output Parameters

<i>c</i>	On exit, if <i>ijob</i> = 0, <i>c</i> is overwritten by the solution <i>R</i> .
<i>f</i>	On exit, if <i>ijob</i> = 0, <i>f</i> is overwritten by the solution <i>L</i> .
<i>scale</i>	REAL for stgsy2/ctgsy2 DOUBLE PRECISION for dtgsy2/ztgsy2. On exit, $0 \leq scale \leq 1$. If $0 < scale < 1$, the solutions <i>R</i> and <i>L</i> (<i>C</i> and <i>F</i> on entry) hold the solutions to a slightly perturbed system, but the input matrices <i>A</i> , <i>B</i> , <i>D</i> and <i>E</i> are not changed. If <i>scale</i> = 0, <i>R</i> and <i>L</i> hold the solutions to the homogeneous system with <i>C</i> = <i>F</i> = 0. Normally <i>scale</i> = 1.
<i>rdsum</i>	On exit, the corresponding sum of squares updated with the contributions from the current sub-system. If <i>trans</i> = 'T', <i>rdsum</i> is not touched. Note that <i>rdsum</i> only makes sense when ?tgsy2 is called by ?tgsyl.
<i>rdscal</i>	On exit, <i>rdscal</i> is updated with respect to the current contributions in <i>rdsum</i> . If <i>trans</i> = 'T', <i>rdscal</i> is not touched.

	Note that <i>rdscal</i> only makes sense when <i>?tgsyl</i> is called by <i>?tgsyl</i> .
<i>pq</i>	INTEGER. Used with real flavors only. On exit, the number of subsystems (of size 2-by-2, 4-by-4 and 8-by-8) solved by the routine <i>stgsy2/dtgsy2</i> .
<i>info</i>	INTEGER. On exit, if <i>info</i> is set to = 0: Successful exit < 0: If <i>info</i> = <i>-i</i> , the <i>i</i> -th argument has an illegal value. > 0: The matrix pairs (<i>A</i> , <i>D</i>) and (<i>B</i> , <i>E</i>) have common or very close eigenvalues.

?trti2

Computes the inverse of a triangular matrix (unblocked algorithm).

Syntax

```
call strti2( uplo, diag, n, a, lda, info )
call dtrti2( uplo, diag, n, a, lda, info )
call ctrti2( uplo, diag, n, a, lda, info )
call ztrti2( uplo, diag, n, a, lda, info )
```

Description

The routine *?trti2* computes the inverse of a real/complex upper or lower triangular matrix. This is the *Level 2 BLAS* version of the algorithm.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the matrix <i>A</i> is upper or lower triangular. = 'U': upper triangular = 'L': lower triangular
<i>diag</i>	CHARACTER*1. Specifies whether or not the matrix <i>A</i> is unit triangular. = 'N': non-unit triangular = 'U': unit triangular

n INTEGER. The order of the matrix *A*. $n \geq 0$.

a REAL for `strti2`
 DOUBLE PRECISION for `dtrti2`
 COMPLEX for `ctrti2`
 COMPLEX*16 for `ztrti2`.
 Array, DIMENSION (*lda*, *n*).
 On entry, the triangular matrix *A*.
 If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of the array *a* contains the upper triangular matrix, and the strictly lower triangular part of *a* is not referenced.
 If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of the array *a* contains the lower triangular matrix, and the strictly upper triangular part of *a* is not referenced.
 If *diag* = 'U', the diagonal elements of *a* are also not referenced and are assumed to be 1.

lda INTEGER. The leading dimension of the array *a*. $lda \geq \max(1, n)$.

Output Parameters

a On exit, the (triangular) inverse of the original matrix, in the same storage format.

info INTEGER.
 = 0: successful exit
 < 0: if *info* = *-k*, the *k*-th argument had an illegal value

clag2z

Converts a complex single precision matrix to a complex double precision matrix.

Syntax

```
call clag2z( m, n, sa, ldsa, a, lda, info )
```

Description

The routine converts a complex single precision matrix *SA* to a complex double precision matrix *A*.

Note that while it is possible to overflow while converting from double to single, it is not possible to overflow when converting from single to double.

This is a helper routine so there is no argument checking.

Input Parameters

<i>m</i>	INTEGER. The number of lines of the matrix <i>A</i> ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in the matrix <i>A</i> ($n \geq 0$).
<i>ldsa</i>	INTEGER. The leading dimension of the array <i>sa</i> ; $ldsa \geq \max(1, m)$.
<i>a</i>	DOUBLE PRECISION array, DIMENSION (<i>lda</i> , <i>n</i>). On entry, contains the <i>m</i> -by- <i>n</i> coefficient matrix <i>A</i> .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> ; $lda \geq \max(1, m)$.

Output Parameters

<i>sa</i>	REAL array, DIMENSION (<i>ldsa</i> , <i>n</i>). On exit, contains the <i>m</i> -by- <i>n</i> coefficient matrix <i>SA</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful.

dlag2s

Converts a double precision matrix to a single precision matrix.

Syntax

```
call dlag2s( m, n, a, lda, sa, ldsa, info )
```

Description

The routine converts a double precision matrix *SA* to a single precision matrix *A*.

RMAX is the overflow for the single precision arithmetic. `dlag2s` checks that all the entries of *A* are between -RMAX and RMAX. If not, the conversion is aborted and a flag is raised.

This is a helper routine so there is no argument checking.

Input Parameters

<i>m</i>	INTEGER. The number of lines of the matrix <i>A</i> ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in the matrix <i>A</i> ($n \geq 0$).
<i>a</i>	DOUBLE PRECISION array, DIMENSION (<i>lda</i> , <i>n</i>). On entry, contains the <i>m</i> -by- <i>n</i> coefficient matrix <i>A</i> .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> ; $lda \geq \max(1, m)$.
<i>ldsa</i>	INTEGER. The leading dimension of the array <i>sa</i> ; $ldsa \geq \max(1, m)$.

Output Parameters

<i>sa</i>	REAL array, DIMENSION (<i>ldsa</i> , <i>n</i>). On exit, if <i>info</i> = 0, contains the <i>m</i> -by- <i>n</i> coefficient matrix <i>SA</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>k</i> , the (<i>i</i> , <i>j</i>) entry of the matrix <i>A</i> has overflowed when moving from double precision to single precision. <i>k</i> is given by $k = (i-1)*lda+j$.

slag2d

Converts a single precision matrix to a double precision matrix.

Syntax

```
call slag2d( m, n, sa, ldsa, a, lda, info )
```

Description

The routine converts a single precision matrix *SA* to a double precision matrix *A*.

Note that while it is possible to overflow while converting from double to single, it is not possible to overflow when converting from single to double.

This is a helper routine so there is no argument checking.

Input Parameters

<i>m</i>	INTEGER. The number of lines of the matrix <i>A</i> ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in the matrix <i>A</i> ($n \geq 0$).
<i>ldsa</i>	INTEGER. The leading dimension of the array <i>sa</i> ; $ldsa \geq \max(1, m)$.
<i>a</i>	DOUBLE PRECISION array, DIMENSION (<i>lda</i> , <i>n</i>). On entry, contains the <i>m</i> -by- <i>n</i> coefficient matrix <i>A</i> .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> ; $lda \geq \max(1, m)$.

Output Parameters

<i>sa</i>	REAL array, DIMENSION (<i>ldsa</i> , <i>n</i>). On exit, if <i>info</i> = 0, contains the <i>m</i> -by- <i>n</i> coefficient matrix <i>SA</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful.

zlag2c

Converts a complex double precision matrix to a complex single precision matrix.

Syntax

```
call zlag2c( m, n, a, lda, sa, ldsa, info )
```

Description

The routine converts a double precision complex matrix *SA* to a single precision complex matrix *A*.

RMAX is the overflow for the single precision arithmetic. *zlag2c* checks that all the entries of *A* are between $-RMAX$ and *RMAX*. If not, the conversion is aborted and a flag is raised.

This is a helper routine so there is no argument checking.

Input Parameters

<i>m</i>	INTEGER. The number of lines of the matrix <i>A</i> ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in the matrix <i>A</i> ($n \geq 0$).
<i>a</i>	DOUBLE PRECISION array, DIMENSION (<i>lda</i> , <i>n</i>). On entry, contains the <i>m</i> -by- <i>n</i> coefficient matrix <i>A</i> .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> ; $lda \geq \max(1, m)$.
<i>ldsa</i>	INTEGER. The leading dimension of the array <i>sa</i> ; $ldsa \geq \max(1, m)$.

Output Parameters

<i>sa</i>	REAL array, DIMENSION (<i>ldsa</i> , <i>n</i>). On exit, if <i>info</i> = 0, contains the <i>m</i> -by- <i>n</i> coefficient matrix <i>SA</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>k</i> , the (<i>i</i> , <i>j</i>) entry of the matrix <i>A</i> has overflowed when moving from double precision to single precision. <i>k</i> is given by $k = (i-1)*lda+j$.

Utility Functions and Routines

This section describes LAPACK utility functions and routines. Summary information about these routines is given in the following table:

Table 5-2 LAPACK Utility Routines

Routine Name	Data Types	Description
<i>ilaver</i>		Returns the version of the Lapack library.
<i>ilaenv</i>		Environmental enquiry function which returns values for tuning algorithmic performance.
<i>iparmq</i>		Environmental enquiry function which returns values for tuning algorithmic performance.

Routine Name	Data Types	Description
ieeeck		Checks if the infinity and NaN arithmetic is safe. Called by ilaenv .
lsame		Tests two characters for equality regardless of case.
lsamen		Tests two character strings for equality regardless of case.
?labad	s, d	Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.
?lamch	s, d	Determines machine parameters for floating-point arithmetic.
?lamc1	s, d	Called from ?lamc2 . Determines machine parameters given by <i>beta</i> , <i>t</i> , <i>rnd</i> , <i>ieee1</i> .
?lamc2	s, d	Used by ?lamch . Determines machine parameters specified in its arguments list.
?lamc3	s, d	Called from ?lamc1 - ?lamc5 . Intended to force <i>a</i> and <i>b</i> to be stored prior to doing the addition of <i>a</i> and <i>b</i> .
?lamc4	s, d	This is a service routine for ?lamc2 .
?lamc5	s, d	Called from ?lamc2 . Attempts to compute the largest machine floating-point number, without overflow.
second/dsecnd		Return user time for a process.
xerbla		Error handling routine called by LAPACK routines.

ilaver

Returns the version of the Lapack library.

Syntax

```
call ilaver( vers_major, vers_minor, vers_patch )
```

Description

This routine returns the version of the Lapack library.

Output Parameters

<code>vers_major</code>	INTEGER. Returns the major version of the LAPACK library.
<code>vers_minor</code>	INTEGER. Returns the minor version from the major version of the LAPACK library.
<code>vers_patch</code>	INTEGER. Returns the patch version from the minor version of the LAPACK library.

ilaenv

Environmental enquiry function that returns values for tuning algorithmic performance.

Syntax

```
value = ilaenv( ispec, name, opts, n1, n2, n3, n4 )
```

Description

Enquiry function `ilaenv` is called from the LAPACK routines to choose problem-dependent parameters for the local environment. See *ispec* below for a description of the parameters.

This version provides a set of parameters that should give good, but not optimal, performance on many of the currently available computers. Users are encouraged to modify this subroutine to set the tuning parameters for their particular machine using the option and problem size information in the arguments.

This routine will not function correctly if it is converted to all lower case. Converting it to all upper case is allowed.

Input Parameters

<code>ispec</code>	INTEGER. Specifies the parameter to be returned as the value of <code>ilaenv</code> : = 1: the optimal blocksize; if this value is 1, an unblocked algorithm will give the best performance.

= 2: the minimum block size for which the block routine should be used; if the usable block size is less than this value, an unblocked routine should be used.
 = 3: the crossover point (in a block routine, for n less than this value, an unblocked routine should be used)
 = 4: the number of shifts, used in the nonsymmetric eigenvalue routines (deprecated)
 = 5: the minimum column dimension for blocking to be used; rectangular blocks must have dimension at least k -by- m , where k is given by `ilaenv(2,...)` and m by `ilaenv(5,...)`
 = 6: the crossover point for the SVD (when reducing an m -by- n matrix to bidiagonal form, if $\max(m,n)/\min(m,n)$ exceeds this value, a QR factorization is used first to reduce the matrix to a triangular form.)
 = 7: the number of processors
 = 8: the crossover point for the multishift QR and QZ methods for nonsymmetric eigenvalue problems (deprecated).
 = 9: maximum size of the subproblems at the bottom of the computation tree in the divide-and-conquer algorithm (used by `?gelss` and `?gesdd`)
 =10: ieee NaN arithmetic can be trusted not to trap
 =11: infinity arithmetic can be trusted not to trap
 $12 \leq ispec \leq 16$: `?hseqr` or one of its subroutines, see `iparmq` for detailed explanation.

name CHARACTER*(*). The name of the calling subroutine, in either upper case or lower case.

opts CHARACTER*(*). The character options to the subroutine *name*, concatenated into a single character string. For example, *uplo* = 'U', *trans* = 'T', and *diag* = 'N' for a triangular routine would be specified as *opts* = 'UTN'.

n1, n2, n3, n4 INTEGER. Problem dimensions for the subroutine *name*; these may not all be required.

Output Parameters

value INTEGER.

If *value* ≥ 0 : the value of the parameter specified by *ispec*;
 If *value* = $-k < 0$: the *k*-th argument had an illegal value.

Application Notes

The following conventions have been used when calling `ilaenv` from the LAPACK routines:

1. *opts* is a concatenation of all of the character options to subroutine *name*, in the same order that they appear in the argument list for *name*, even if they are not used in determining the value of the parameter specified by *ispec*.
2. The problem dimensions *n1*, *n2*, *n3*, *n4* are specified in the order that they appear in the argument list for *name*. *n1* is used first, *n2* second, and so on, and unused problem dimensions are passed a value of -1.
3. The parameter value returned by `ilaenv` is checked for validity in the calling subroutine. For example, `ilaenv` is used to retrieve the optimal blocksize for `strtri` as follows:

```
nb = ilaenv( 1, 'strtri', uplo // diag, n, -1, -1, -1> )
if( nb.le.1 ) nb = max( 1, n )
```

Below is an example of `ilaenv` usage in C language:

```
#include <stdio.h>
#include "mkl.h"
int main(void)
{
    int size = 1000;
    int ispec = 1;
    int dummy = -1;
    int blockSize1 = ilaenv(&ispec, "dsytrd", "U", &size, &dummy, &dummy, &dummy);
    int blockSize2 = ilaenv(&ispec, "dormtr", "LUN", &size, &size, &dummy, &dummy);
    printf("DSYTRD blocksize = %d\n", blockSize1);
    printf("DORMTR blocksize = %d\n", blockSize2);
    return 0;
}
```


iparmq

Environmental enquiry function which returns values for tuning algorithmic performance.

Syntax

```
value = iparmq( ispec, name, opts, n, ilo, ihi, lwork )
```

Description

This function sets problem and machine dependent parameters useful for ?hseqr and its subroutines. It is called whenever ilaenv is called with $12 \leq ispec \leq 16$.

Input Parameters

ispec INTEGER.
Specifies the parameter to be returned as the value of iparmq:

- = 12: (*inmin*) Matrices of order *nmin* or less are sent directly to ?lahqr, the implicit double shift QR algorithm. *nmin* must be at least 11.
- = 13: (*inwin*) Size of the deflation window. This is best set greater than or equal to the number of simultaneous shifts *ns*. Larger matrices benefit from larger deflation windows.
- = 14: (*inibl*) Determines when to stop nibbling and invest in an (expensive) multi-shift QR sweep. If the aggressive early deflation subroutine finds *ld* converged eigenvalues from an order *nw* deflation window and $ld > (nw * nibble) / 100$, then the next QR sweep is skipped and early deflation is applied immediately to the remaining active diagonal block. Setting iparmq(*ispec*=14)=0 causes TTQRE to skip a multi-shift QR sweep whenever early deflation finds a converged eigenvalue. Setting iparmq(*ispec*=14) greater than or equal to 100 prevents TTQRE from skipping a multi-shift QR sweep.
- = 15: (*nshfts*) The number of simultaneous shifts in a multi-shift QR iteration.
- = 16: (*iacc22*) iparmq is set to 0, 1 or 2 with the following meanings.

0: During the multi-shift QR sweep, `?laqr5` does not accumulate reflections and does not use matrix-matrix multiply to update the far-from-diagonal matrix entries.
 1: During the multi-shift QR sweep, `?laqr5` and/or `?laqr3` accumulates reflections and uses matrix-matrix multiply to update the far-from-diagonal matrix entries.
 2: During the multi-shift QR sweep, `?laqr5` accumulates reflections and takes advantage of 2-by-2 block structure during matrix-matrix multiplies.
 (If `?trrm` is slower than `?gemm`, then `iparmq(ispec=16)=1` may be more efficient than `iparmq(ispec=16)=2` despite the greater level of arithmetic work implied by the latter choice.)

<i>name</i>	CHARACTER*(*) . The name of the calling subroutine.
<i>opts</i>	CHARACTER*(*) . This is a concatenation of the string arguments to TTQRE.
<i>n</i>	INTEGER. <i>n</i> is the order of the Hessenberg matrix <i>H</i> .
<i>ilo, ihi</i>	INTEGER. It is assumed that <i>H</i> is already upper triangular in rows and columns 1: <i>ilo</i> -1 and <i>ihi</i> +1: <i>n</i> .
<i>lwork</i>	INTEGER. The amount of workspace available.

Output Parameters

<i>value</i>	INTEGER. If <i>value</i> ≥ 0: the value of the parameter specified by <i>iparmq</i> ; If <i>value</i> = -k < 0: the k-th argument had an illegal value.

Application Notes

The following conventions have been used when calling `ilaenv` from the LAPACK routines:

1. *opts* is a concatenation of all of the character options to subroutine *name*, in the same order that they appear in the argument list for *name*, even if they are not used in determining the value of the parameter specified by *ispec*.

2. The problem dimensions $n1$, $n2$, $n3$, $n4$ are specified in the order that they appear in the argument list for *name*. $n1$ is used first, $n2$ second, and so on, and unused problem dimensions are passed a value of -1.
3. The parameter value returned by *ilaenv* is checked for validity in the calling subroutine. For example, *ilaenv* is used to retrieve the optimal blocksize for *strtri* as follows:

```
nb = ilaenv( 1, 'strtri', uplo // diag, n, -1, -1, -1 )
if( nb.le.1 ) nb = max( 1, n )
```

ieeeck

Checks if the infinity and NaN arithmetic is safe.
Called by *ilaenv*.

Syntax

```
ival = ieeck( ispec, zero, one )
```

Description

The function *ieeeck* is called from *ilaenv* to verify that infinity and possibly NaN arithmetic is safe, that is, will not trap.

Input Parameters

<i>ispec</i>	INTEGER. Specifies whether to test just for infinity arithmetic or both for infinity and NaN arithmetic: If <i>ispec</i> = 0: Verify infinity arithmetic only. If <i>ispec</i> = 1: Verify infinity and NaN arithmetic.
<i>zero</i>	REAL. Must contain the value 0.0 This is passed to prevent the compiler from optimizing away this code.
<i>one</i>	REAL. Must contain the value 1.0 This is passed to prevent the compiler from optimizing away this code.

Output Parameters

<i>ival</i>	INTEGER.

If `ival = 0`: Arithmetic failed to produce the correct answers.
If `ival = 1`: Arithmetic produced the correct answers.

lsame

Tests two characters for equality regardless of case.

Syntax

```
val = lsame(ca, cb)
```

Description

This logical function returns `.TRUE.` if `ca` is the same letter as `cb` regardless of case.

Input Parameters

`ca, cb` CHARACTER*1. Specify the single characters to be compared.

Output Parameters

`val` LOGICAL. Result of the comparison.

lsamen

Tests two character strings for equality regardless of case.

Syntax

```
val = lsamen(n, ca, cb)
```

Description

This logical function tests if the first `n` letters of the string `ca` are the same as the first `n` letters of `cb`, regardless of case. The function `lsamen` returns `.TRUE.` if `ca` and `cb` are equivalent except for case and `.FALSE.` otherwise. `lsamen` also returns `.FALSE.` if `len(ca)` or `len(cb)` is less than `n`.

Input Parameters

<i>n</i>	INTEGER. The number of characters in <i>ca</i> and <i>cb</i> to be compared.
<i>ca, cb</i>	CHARACTER* (*). Specify two character strings of length at least <i>n</i> to be compared. Only the first <i>n</i> characters of each string will be accessed.

Output Parameters

<i>val</i>	LOGICAL. Result of the comparison.

?labad

Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.

Syntax

```
call slabad( small, large )
call dlabad( small, large )
```

Description

This routine takes as input the values computed by `slamch/dlamch` for underflow and overflow, and returns the square root of each of these values if the log of *large* is sufficiently large. This subroutine is intended to identify machines with a large exponent range, such as the Crays, and redefine the underflow and overflow limits to be the square roots of the values computed by `?lamch`. This subroutine is needed because `?lamch` does not compensate for poor arithmetic in the upper half of the exponent range, as is found on a Cray.

Input Parameters

<i>small</i>	REAL for slabad DOUBLE PRECISION for dlabad. The underflow threshold as computed by <code>?lamch</code> .
<i>large</i>	REAL for slabad DOUBLE PRECISION for dlabad. The overflow threshold as computed by <code>?lamch</code> .

Output Parameters

<i>small</i>	On exit, if $\log_{10}(\textit{large})$ is sufficiently large, the square root of <i>small</i> , otherwise unchanged.
<i>large</i>	On exit, if $\log_{10}(\textit{large})$ is sufficiently large, the square root of <i>large</i> , otherwise unchanged.

?lamch

Determines machine parameters for floating-point arithmetic.

Syntax

```
val = slamch( cmach )
```

```
val = dlamch( cmach )
```

Description

The function ?lamch determines single precision and double precision machine parameters.

Input Parameters

<i>cmach</i>	CHARACTER*1. Specifies the value to be returned by ?lamch: = 'E' or 'e', <i>val</i> = <i>eps</i> = 'S' or 's', <i>val</i> = <i>sfmin</i> = 'B' or 'b', <i>val</i> = <i>base</i> = 'P' or 'p', <i>val</i> = <i>eps*base</i> = 'n' or 'n', <i>val</i> = <i>t</i> = 'R' or 'r', <i>val</i> = <i>rnd</i> = 'm' or 'm', <i>val</i> = <i>emin</i> = 'U' or 'u', <i>val</i> = <i>rmin</i> = 'L' or 'l', <i>val</i> = <i>emax</i> = 'O' or 'o', <i>val</i> = <i>rmax</i> where <i>eps</i> = relative machine precision; <i>sfmin</i> = safe minimum, such that $1/\textit{sfmin}$ does not overflow; <i>base</i> = base of the machine; <i>prec</i> = <i>eps*base</i>;

```

t = number of (base) digits in the mantissa;
rnd = 1.0 when rounding occurs in addition, 0.0 otherwise;
emin = minimum exponent before (gradual) underflow;
rmin = underflow_threshold - base**(emin-1);
emax = largest exponent before overflow;
rmax = overflow_threshold - (base**emax)*(1-eps).

```

Output Parameters

<i>val</i>	REAL for slamch DOUBLE PRECISION for dlamch Value returned by the function.

?lamc1

*Called from ?lamc2. Determines machine parameters given by *beta*, *t*, *rnd*, *ieee1*.*

Syntax

```

call slamc1( beta, t, rnd, ieee1 )
call dlamc1( beta, t, rnd, ieee1 )

```

Description

The routine ?lamc1 determines machine parameters given by *beta*, *t*, *rnd*, *ieee1*.

Output Parameters

<i>beta</i>	INTEGER. The base of the machine.
<i>t</i>	INTEGER. The number of (<i>beta</i>) digits in the mantissa.
<i>rnd</i>	LOGICAL. Specifies whether proper rounding (<i>rnd</i> = .TRUE.) or chopping (<i>rnd</i> = .FALSE.) occurs in addition. This may not be a reliable guide to the way in which the machine performs its arithmetic.
<i>ieee1</i>	LOGICAL. Specifies whether rounding appears to be done in the <i>ieee</i> 'round to nearest' style.

?lamc2

Used by ?lamch. Determines machine parameters specified in its arguments list.

Syntax

```
call slamc2( beta, t, rnd, eps, emin, rmin, emax, rmax )
call dlamc2( beta, t, rnd, eps, emin, rmin, emax, rmax )
```

Description

The routine ?lamc2 determines machine parameters specified in its arguments list.

Output Parameters

<i>beta</i>	INTEGER. The base of the machine.
<i>t</i>	INTEGER. The number of (<i>beta</i>) digits in the mantissa.
<i>rnd</i>	LOGICAL. Specifies whether proper rounding (<i>rnd</i> = .TRUE.) or chopping (<i>rnd</i> = .FALSE.) occurs in addition. This may not be a reliable guide to the way in which the machine performs its arithmetic.
<i>eps</i>	REAL for slamc2 DOUBLE PRECISION for dlamc2 The smallest positive number such that $fl(1.0 - eps) < 1.0$, where <i>fl</i> denotes the computed value.
<i>emin</i>	INTEGER. The minimum exponent before (gradual) underflow occurs.
<i>rmin</i>	REAL for slamc2 DOUBLE PRECISION for dlamc2 The smallest normalized number for the machine, given by $base^{emin-1}$, where <i>base</i> is the floating point value of <i>beta</i> .
<i>emax</i>	INTEGER. The maximum exponent before overflow occurs.
<i>rmax</i>	REAL for slamc2 DOUBLE PRECISION for dlamc2

The largest positive number for the machine, given by $base^{emax(1 - eps)}$, where *base* is the floating point value of *beta*.

?lamc3

*Called from ?lamc1-?lamc5. Intended to force *a* and *b* to be stored prior to doing the addition of *a* and *b*.*

Syntax

```
val = slamc3( a, b )
```

```
val = dlamc3( a, b )
```

Description

The routine is intended to force *A* and *B* to be stored prior to doing the addition of *A* and *B*, for use in situations where optimizers might hold one of these in a register.

Input Parameters

a, b REAL for slamc3
 DOUBLE PRECISION for dlamc3
 The values *a* and *b*.

Output Parameters

val REAL for slamc3
 DOUBLE PRECISION for dlamc3
 The result of adding values *a* and *b*.

?lamc4

This is a service routine for ?lamc2.

Syntax

```
call slamc4( emin, start, base )
```

```
call dlamc4( emin, start, base )
```

Description

This is a service routine for `?lamc2`.

Input Parameters

<i>start</i>	REAL for <code>slamc4</code> DOUBLE PRECISION for <code>dlamc4</code> The starting point for determining <i>emin</i> .
<i>base</i>	INTEGER. The base of the machine.

Output Parameters

<i>emin</i>	INTEGER. The minimum exponent before (gradual) underflow, computed by setting <i>a</i> = <i>start</i> and dividing by <i>base</i> until the previous <i>a</i> can not be recovered.

?lamc5

Called from ?lamc2. Attempts to compute the largest machine floating-point number, without overflow.

Syntax

```
call slamc5( beta, p, emin, ieee, emax, rmax)
call dlamc5( beta, p, emin, ieee, emax, rmax)
```

Description

The routine `?lamc5` attempts to compute *rmax*, the largest machine floating-point number, without overflow. It assumes that *emax* + *abs(emin)* sum approximately to a power of 2. It will fail on machines where this assumption does not hold, for example, the Cyber 205 (*emin* = -28625, *emax* = 28718). It will also fail if the value supplied for *emin* is too large (that is, too close to zero), probably with overflow.

Input Parameters

<i>beta</i>	INTEGER. The base of floating-point arithmetic.
<i>p</i>	INTEGER. The number of base <i>beta</i> digits in the mantissa of a floating-point value.

<i>emin</i>	INTEGER. The minimum exponent before (gradual) underflow.
<i>ieee</i>	LOGICAL. A logical flag specifying whether or not the arithmetic system is thought to comply with the IEEE standard.

Output Parameters

<i>emax</i>	INTEGER. The largest exponent before overflow.
<i>rmax</i>	REAL for <code>slamc5</code> DOUBLE PRECISION for <code>dlamc5</code> The largest machine floating-point number.

second/dsecnd

Return user time for a process.

Syntax

```
val = second()
val = dsecnd()
```

Description

The functions `second/dsecnd` return the user time for a process in seconds. These versions get the time from the system function `etime`. The difference is that `dsecnd` returns the result with double precision.

Output Parameters

<i>val</i>	REAL for <code>second</code> DOUBLE PRECISION for <code>dsecnd</code> User time for a process.

xerbla

Error handling routine called by BLAS, LAPACK, VML routines.

Syntax

```
call xerbla( sname, info )
```

Description

The routine `xerbla` is an error handler for the BLAS, LAPACK, and VML routines. It is called by a BLAS, LAPACK, or VML routine if an input parameter has an invalid value.

A message is printed and execution stops.

Installers may consider modifying the `stop` statement to call system-specific exception-handling facilities.

Input Parameters

<i>sname</i>	CHARACTER*6 The name of the routine which called <code>xerbla</code> .
<i>info</i>	INTEGER. The position of the invalid parameter in the parameter list of the calling routine.

ScaLAPACK Routines

6

This chapter describes the Intel® Math Kernel Library implementation of routines from the ScaLAPACK package for distributed-memory architectures. Routines are supported for both real and complex dense and band matrices to perform the tasks of solving systems of linear equations, solving linear least-squares problems, eigenvalue and singular value problems, as well as performing a number of related computational tasks. All routines are available in both single precision and double precision.



NOTE. ScaLAPACK routines are provided only with Intel® MKL versions for Linux* and Windows* OSs.

Sections in this chapter include descriptions of ScaLAPACK [computational routines](#) that perform distinct computational tasks, as well as [driver routines](#) for solving standard types of problems in one call.

Generally, ScaLAPACK runs on a network of computers using MPI as a message-passing layer and a set of prebuilt communication subprograms (BLACS), as well as a set of BLAS optimized for the target architecture. Intel MKL version of ScaLAPACK is optimized for Intel® processors. For the detailed system and environment requirements see *Intel® MKL Release Notes* and *Intel® MKL User's Guide*.

For full reference on ScaLAPACK routines and related information see [\[SLUG\]](#).

Overview

The model of the computing environment for ScaLAPACK is represented as a one-dimensional array of processes (for operations on band or tridiagonal matrices) or also a two-dimensional process grid (for operations on dense matrices). To use ScaLAPACK, all global matrices or vectors should be distributed on this array or grid prior to calling the ScaLAPACK routines.

ScaLAPACK uses the two-dimensional block-cyclic data distribution as a layout for dense matrix computations. This distribution provides good work balance between available processors, as well as gives the opportunity to use BLAS Level 3 routines for optimal local computations. Information about the data distribution that is required to establish the mapping between each global array and its corresponding process and memory location is contained in the so called *array descriptor* associated with each global array. An example of an array descriptor structure is given in [Table 6-1](#).

Table 6-1 Content of the array descriptor for dense matrices

Array Element #	Name	Definition
1	<i>dtype</i>	Descriptor type (=1 for dense matrices)
2	<i>ctxt</i>	BLACS context handle for the process grid

Array Element #	Name	Definition
3	<i>m</i>	Number of rows in the global array
4	<i>n</i>	Number of columns in the global array
5	<i>mb</i>	Row blocking factor
6	<i>nb</i>	Column blocking factor
7	<i>rsrc</i>	Process row over which the first row of the global array is distributed
8	<i>csrc</i>	Process column over which the first column of the global array is distributed
9	<i>lld</i>	Leading dimension of the local array

The number of rows and columns of a global dense matrix that a particular process in a grid receives after data distributing is denoted by $LOC_r()$ and $LOC_c()$, respectively. To compute these numbers, you can use the ScaLAPACK tool routine `numroc`.

After the block-cyclic distribution of global data is done, you may choose to perform an operation on a submatrix of the global matrix A , which is contained in the global subarray $\text{sub}(A)$, defined by the following 6 values (for dense matrices):

<i>m</i>	The number of rows of $\text{sub}(A)$
<i>n</i>	The number of columns of $\text{sub}(A)$
<i>a</i>	A pointer to the local array containing the entire global array A
<i>ia</i>	The row index of $\text{sub}(A)$ in the global array
<i>ja</i>	The column index of $\text{sub}(A)$ in the global array
<i>desca</i>	The array descriptor for the global array

Routine Naming Conventions

For each routine introduced in this chapter, you can use the ScaLAPACK name. The naming convention for ScaLAPACK routines is similar to that used for LAPACK routines (see [Routine Naming Conventions in Chapter 4](#)). A general rule is that each routine name in ScaLAPACK, which has an LAPACK equivalent, is simply the LAPACK name prefixed by initial letter *p*.

ScaLAPACK names have the structure $p?yyzzz$ or $p?yyzz$, which is described below.

The initial letter *p* is a distinctive prefix of ScaLAPACK routines and is present in each such routine.

The second symbol *?* indicates the data type:

<i>s</i>	real, single precision
<i>d</i>	real, double precision
<i>c</i>	complex, single precision

z complex, double precision

The second and third letters *yy* indicate the matrix type as:

ge	general
gb	general band
gg	a pair of general matrices (for a generalized problem)
dt	general tridiagonal (diagonally dominant-like)
db	general band (diagonally dominant-like)
po	symmetric or Hermitian positive-definite
pb	symmetric or Hermitian positive-definite band
pt	symmetric or Hermitian positive-definite tridiagonal
sy	symmetric
st	symmetric tridiagonal (real)
he	Hermitian
or	orthogonal
tr	triangular (or quasi-triangular)
tz	trapezoidal
un	unitary

For computational routines, the last three letters **zzz** indicate the computation performed and have the same meaning as for LAPACK routines.

For driver routines, the last two letters **zz** or three letters **zzz** have the following meaning:

sv	a <i>simple</i> driver for solving a linear system
svx	an <i>expert</i> driver for solving a linear system
ls	a driver for solving a linear least squares problem
ev	a simple driver for solving a symmetric eigenvalue problem
evx	an expert driver for solving a symmetric eigenvalue problem
svd	a driver for computing a singular value decomposition
gvx	an expert driver for solving a generalized symmetric definite eigenvalue problem

Simple driver here means that the driver just solves the general problem, whereas an *expert* driver is more versatile and can also optionally perform some related computations (such, for example, as refining the solution and computing error bounds after the linear system is solved).

Computational Routines

In the sections that follow, the descriptions of ScaLAPACK computational routines are given. These routines perform distinct computational tasks that can be used for:

- [Solving Systems of Linear Equations](#)
- [Orthogonal Factorizations and LLS Problems](#)
- [Symmetric Eigenproblems](#)
- [Nonsymmetric Eigenproblems](#)
- [Singular Value Decomposition](#)
- [Generalized Symmetric-Definite Eigenproblems](#)

See also the respective [driver routines](#).

Linear Equations

ScaLAPACK supports routines for the systems of equations with the following types of matrices:

- general
- general banded
- general diagonally dominant-like banded (including general tridiagonal)
- symmetric or Hermitian positive-definite
- symmetric or Hermitian positive-definite banded
- symmetric or Hermitian positive-definite tridiagonal

A *diagonally dominant-like* matrix is defined as a matrix for which it is known in advance that pivoting is not required in the *LU* factorization of this matrix.

For the above matrix types, the library includes routines for performing the following computations: *factoring* the matrix; *equilibrating* the matrix; *solving* a system of linear equations; *estimating the condition number* of a matrix; *refining* the solution of linear equations and computing its error bounds; *inverting* the matrix. Note that for some of the listed matrix types only part of the computational routines are provided (for example, routines that refine the solution are not provided for band or tridiagonal matrices). See [Table 6-2](#) for full list of available routines.

To solve a particular problem, you can either call two or more computational routines or call a corresponding [driver routine](#) that combines several tasks in one call. Thus, to solve a system of linear equations with a general matrix, you can first call `p?getrf` (*LU* factorization) and then

`p?getrs`(computing the solution). Then, you might wish to call `p?gerfs` to refine the solution and get the error bounds. Alternatively, you can just use the driver routine `p?gesvx` which performs all these tasks in one call.

Table 6-2 lists the ScaLAPACK computational routines for factorizing, equilibrating, and inverting matrices, estimating their condition numbers, solving systems of equations with real matrices, refining the solution, and estimating its error.

Table 6-2 Computational Routines for Systems of Linear Equations

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
general (partial pivoting)	<code>p?getrf</code>	<code>p?geequ</code>	<code>p?getrs</code>	<code>p?gecon</code>	<code>p?gerfs</code>	<code>p?getri</code>
general band (partial pivoting)	<code>p?gbtrf</code>		<code>p?gbtrs</code>			
general band (no pivoting)	<code>p?dbtrf</code>		<code>p?dbtrs</code>			
general tridiagonal (no pivoting)	<code>p?dttrf</code>		<code>p?dttrs</code>			
symmetric/Hermitian positive-definite	<code>p?potrf</code>	<code>p?poequ</code>	<code>p?potrs</code>	<code>p?pocon</code>	<code>p?porfs</code>	<code>p?potri</code>
symmetric/Hermitian positive-definite, band	<code>p?pbtrf</code>		<code>p?pbtrs</code>			
symmetric/Hermitian positive-definite, tridiagonal	<code>p?pttrf</code>		<code>p?pttrs</code>			
triangular			<code>p?trtrs</code>	<code>p?trcon</code>	<code>p?trrfs</code>	<code>p?trtri</code>

In this table ? stands for *s* (single precision real), *d* (double precision real), *c* (single precision complex), or *z* (double precision complex).

Routines for Matrix Factorization

This section describes the ScaLAPACK routines for matrix factorization. The following factorizations are supported:

- LU factorization of general matrices
- LU factorization of diagonally dominant-like matrices
- Cholesky factorization of real symmetric or complex Hermitian positive-definite matrices

You can compute the factorizations using full and band storage of matrices.

p?getrf

Computes the LU factorization of a general m-by-n distributed matrix.

Syntax

```
call psgetrf(m, n, a, ia, ja, desca, ipiv, info)
call pdgetrf(m, n, a, ia, ja, desca, ipiv, info)
call pcgetrf(m, n, a, ia, ja, desca, ipiv, info)
call pzgetrf(m, n, a, ia, ja, desca, ipiv, info)
```

Description

The routine forms the LU factorization of a general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ as

$$A = P * L * U$$

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U is upper triangular (upper trapezoidal if $m < n$). L and U are stored in $\text{sub}(A)$.

The routine uses partial pivoting, with row interchanges.

Input Parameters

m	(global) INTEGER. The number of rows in the distributed submatrix $\text{sub}(A)$; $m \geq 0$.
n	(global) INTEGER. The number of columns in the distributed submatrix $\text{sub}(A)$; $n \geq 0$.
a	(local) REAL for psgetrf DOUBLE PRECISION for pdgetrf COMPLEX for pcgetrf DOUBLE COMPLEX for pzgetrf. Pointer into the local memory to an array of local dimension $(lld\_a, LOCc(ja+n-1))$. Contains the local pieces of the distributed matrix $\text{sub}(A)$ to be factored.

ia, ja (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix $A(ia:ia+n-1, ja:ja+n-1)$, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

Output Parameters

a Overwritten by local pieces of the factors *L* and *U* from the factorization $A = P * L * U$. The unit diagonal elements of *L* are not stored.

ipiv (local) INTEGER array.
The dimension of *ipiv* is $(LOCr(m\_a) + mb\_a)$.
This array contains the pivoting information: local row *i* was interchanged with global row *ipiv*(*i*). This array is tied to the distributed matrix *A*.

info (global) INTEGER.
If *info*=0, the execution is successful.
info < 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = $-(i*100+j)$; if the *i*-th argument is a scalar and had an illegal value, then *info* = $-i$.
If *info* = *i*, u_{ii} is 0. The factorization has been completed, but the factor *U* is exactly singular. Division by zero will occur if you use the factor *U* for solving a system of linear equations.

p?gbtrf

Computes the LU factorization of a general n -by- n banded distributed matrix.

Syntax

```
call psgbtrf(n, bwl, bwu, a, ja, desca, ipiv, af, laf, work, lwork, info)
call pdgbtrf(n, bwl, bwu, a, ja, desca, ipiv, af, laf, work, lwork, info)
call pcgbtrf(n, bwl, bwu, a, ja, desca, ipiv, af, laf, work, lwork, info)
call pzgbtrf(n, bwl, bwu, a, ja, desca, ipiv, af, laf, work, lwork, info)
```

Description

The routine computes the LU factorization of a general n -by- n real/complex banded distributed matrix $A(1:n, ja:ja+n-1)$ using partial pivoting with row interchanges.

The resulting factorization is not the same factorization as returned from the LAPACK routine [?gbtrf](#). Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form

$$A(1:n, ja:ja+n-1) = P * L * U * Q$$

where P and Q are permutation matrices, and L and U are banded lower and upper triangular matrices, respectively. The matrix Q represents reordering of columns for the sake of parallelism, while P represents reordering of rows for numerical stability using classic partial pivoting.

Input Parameters

n	(global) INTEGER. The number of rows and columns in the distributed submatrix $A(1:n, ja:ja+n-1)$; $n \geq 0$.
bwl	(global) INTEGER. The number of sub-diagonals within the band of A ($0 \leq bwl \leq n-1$).
bwu	(global) INTEGER. The number of super-diagonals within the band of A ($0 \leq bwu \leq n-1$).
a	(local) REAL for psgbtrf

DOUBLE PRECISION for pdgbtrf
 COMPLEX for pcgbtrf
 DOUBLE COMPLEX for pzgbtrf.
 Pointer into the local memory to an array of local dimension
 (lld_a , $LOC_c(ja+n-1)$) where
 $lld\_a \geq 2*bwl + 2*bwu + 1$.
 Contains the local pieces of the n -by- n distributed banded
 matrix $A(1:n, ja:ja+n-1)$ to be factored.

ja (global) INTEGER. The index in the global array *A* that points
 to the start of the matrix to be operated on (which may be
 either all of *A* or a submatrix of *A*).

desca (global and local) INTEGER array, dimension ($dlen\_$). The
 array descriptor for the distributed matrix *A*.
 If $desca(dtype\_)$ = 501, then $dlen\_ \geq 7$;
 else if $desca(dtype\_)$ = 1, then $dlen\_ \geq 9$.

laf (local) INTEGER. The dimension of the array *af*.
 Must be $laf \geq$
 $(NB+bwu) * (bwl+bwu) + 6 * (bwl+bwu) * (bwl+2*bwu)$.
 If *laf* is not large enough, an error code will be returned
 and the minimum acceptable size will be returned in *af*(1).

work (local) Same type as *a*. Workspace array of dimension *lwork*
 .

lwork (local or global) INTEGER. The size of the *work* array ($lwork$
 ≥ 1) . If *lwork* is too small, the minimal acceptable size will
 be returned in *work*(1) and an error code is returned.

Output Parameters

a On exit, this array contains details of the factorization. Note
 that additional permutations are performed on the matrix,
 so that the factors returned are different from those returned
 by LAPACK.

ipiv (local) INTEGER array.
 The dimension of *ipiv* must be $\geq desca(NB)$.

	Contains pivot indices for local factorizations. Note that you <i>should not alter</i> the contents of this array between factorization and solve.
<i>af</i>	(local) REAL for psgbtrf DOUBLE PRECISION for pdgbtrf COMPLEX for pcgbtrf DOUBLE COMPLEX for pzgbtrf. Array, dimension (<i>laf</i>). Auxiliary Fillin space. Fillin is created during the factorization routine p?gbtrf and this is stored in <i>af</i>. Note that if a linear system is to be solved using p?gbtrs after the factorization routine, <i>af</i> must not be altered after the factorization.
<i>work</i> (1)	On exit, <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. If <i>info</i>=0, the execution is successful. <i>info</i> < 0: If the <i>i</i>th argument is an array and the <i>j</i>th entry had an illegal value, then <i>info</i> = -(<i>i</i>*100+<i>j</i>); if the <i>i</i>th argument is a scalar and had an illegal value, then <i>info</i> = -<i>i</i>. <i>info</i> > 0: If <i>info</i> = <i>k</i> ≤ NPROCS, the submatrix stored on processor <i>info</i> and factored locally was not nonsingular, and the factorization was not completed. If <i>info</i> = <i>k</i> > NPROCS, the submatrix stored on processor <i>info</i>-NPROCS representing interactions with other processors was not nonsingular, and the factorization was not completed.

p?dbtrf

Computes the LU factorization of a n -by- n diagonally dominant-like banded distributed matrix.

Syntax

```
call psdbtrf(n, bwl, bwu, a, ja, desca, af, laf, work, lwork, info)
call pddbtrf(n, bwl, bwu, a, ja, desca, af, laf, work, lwork, info)
call pcdbrtf(n, bwl, bwu, a, ja, desca, af, laf, work, lwork, info)
call pzdbtrf(n, bwl, bwu, a, ja, desca, af, laf, work, lwork, info)
```

Description

The routine computes the LU factorization of a n -by- n real/complex diagonally dominant-like banded distributed matrix $A(1:n, ja:ja+n-1)$ without pivoting.

Note that the resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

Input Parameters

n	(global) INTEGER. The number of rows and columns in the distributed submatrix $A(1:n, ja:ja+n-1)$; $n \geq 0$.
bwl	(global) INTEGER. The number of sub-diagonals within the band of A $(0 \leq bwl \leq n-1)$.
bwu	(global) INTEGER. The number of super-diagonals within the band of A $(0 \leq bwu \leq n-1)$.
a	(local) REAL for psdbtrf DOUBLE PRECISION for pddbtrf COMPLEX for pcdbrtf DOUBLE COMPLEX for pzdbtrf. Pointer into the local memory to an array of local dimension $(lld\_a, LOCC(ja+n-1))$.

	Contains the local pieces of the n -by- n distributed banded matrix $A(1:n, ja:ja+n-1)$ to be factored.
<i>ja</i>	(global) INTEGER. The index in the global array <i>a</i> that points to the start of the matrix to be operated on (which may be either all of <i>a</i> or a submatrix of <i>a</i>).
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> . If <i>desca</i> (<i>dtype_</i>) = 501, then <i>dlen_</i> ≥ 7; else if <i>desca</i> (<i>dtype_</i>) = 1, then <i>dlen_</i> ≥ 9.
<i>laf</i>	(local) INTEGER. The dimension of the array <i>af</i> . Must be $laf \geq NB * (bwl + bwu) + (\max(bwl, bwu))^2$. If <i>laf</i> is not large enough, an error code will be returned and the minimum acceptable size will be returned in <i>af</i> (1).
<i>work</i>	(local) Same type as <i>a</i> . Workspace array of dimension <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER. The size of the <i>work</i> array, must be $lwork \geq (\max(bwl, bwu))^2$. If <i>lwork</i> is too small, the minimal acceptable size will be returned in <i>work</i> (1) and an error code is returned.

Output Parameters

<i>a</i>	On exit, this array contains details of the factorization. Note that additional permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.
<i>af</i>	(local) REAL for psdbtrf DOUBLE PRECISION for pddbtrf COMPLEX for pcdbrf DOUBLE COMPLEX for pzdbtrf. Array, dimension (<i>laf</i>). Auxiliary Fillin space. Fillin is created during the factorization routine p?dbtrf and this is stored in <i>af</i> .

Note that if a linear system is to be solved using `p?dbtrs` after the factorization routine, `af` must not be altered after the factorization.

`work(1)` On exit, `work(1)` contains the minimum value of `lwork` required for optimum performance.

`info` (global) INTEGER.
 If `info=0`, the execution is successful.
`info < 0`:
 If the `i`th argument is an array and the `j`-th entry had an illegal value, then `info = -(i*100+j)`; if the `i`-th argument is a scalar and had an illegal value, then `info = -i`. `info > 0`:
 If `info = k ≤ NPROCS`, the submatrix stored on processor `info` and factored locally was not diagonally dominant-like, and the factorization was not completed. If `info = k > NPROCS`, the submatrix stored on processor `info-NPROCS` representing interactions with other processors was not nonsingular, and the factorization was not completed.

p?potrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite distributed matrix.

Syntax

```
call pspotrf(uplo, n, a, ia, ja, desca, info)
call pdpotrf(uplo, n, a, ia, ja, desca, info)
call pcpotrf(uplo, n, a, ia, ja, desca, info)
call pzpotrf(uplo, n, a, ia, ja, desca, info)
```

Description

This routine computes the Cholesky factorization of a real symmetric or complex Hermitian positive-definite distributed n -by- n matrix $A(ia:ia+n-1, ja:ja+n-1)$, denoted below as $\text{sub}(A)$.

The factorization has the form

$\text{sub}(A) = U^H * U$ if $\text{uplo} = 'U'$, or

$\text{sub}(A) = L * L^H$ if $\text{uplo} = 'L'$

where L is a lower triangular matrix and U is upper triangular.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Indicates whether the upper or lower triangular part of $\text{sub}(A)$ is stored. Must be 'U' or 'L'. If $\text{uplo} = 'U'$, the array <i>a</i> stores the upper triangular part of the matrix $\text{sub}(A)$ that is factored as $U^H * U$. If $\text{uplo} = 'L'$, the array <i>a</i> stores the lower triangular part of the matrix $\text{sub}(A)$ that is factored as $L * L^H$.
<i>n</i>	(global) INTEGER. The order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).
<i>a</i>	(local) REAL for pspotrf DOUBLE PRECISION for pdpotrf COMPLEX for pcspotrf DOUBLE COMPLEX for pzpotrf. Pointer into the local memory to an array of dimension (<i>lld_a</i> , <i>LOCc(ja+n-1)</i>). On entry, this array contains the local pieces of the n -by- n symmetric/Hermitian distributed matrix $\text{sub}(A)$ to be factored. Depending on <i>uplo</i> , the array <i>a</i> contains either the upper or the lower triangular part of the matrix $\text{sub}(A)$ (see <i>uplo</i>).
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>A</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .

Output Parameters

<i>a</i>	The upper or lower triangular part of <i>a</i> is overwritten by the Cholesky factor U or L , as specified by <i>uplo</i> .

info

(global) INTEGER.

If *info*=0, the execution is successful;

info < 0: if the *i*-th argument is an array, and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

If *info* = *k* > 0, the leading minor of order *k*, $A(ia:ia+k-1, ja:ja+k-1)$, is not positive-definite, and the factorization could not be completed.

p?pbtrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite banded distributed matrix.

Syntax

```
call pspbtrf(uplo, n, bw, a, ja, desca, af, laf, work, lwork, info)
call pdpbtrf(uplo, n, bw, a, ja, desca, af, laf, work, lwork, info)
call pcpbtrf(uplo, n, bw, a, ja, desca, af, laf, work, lwork, info)
call pzpbtfrf(uplo, n, bw, a, ja, desca, af, laf, work, lwork, info)
```

Description

This routine computes the Cholesky factorization of an *n*-by-*n* real symmetric or complex Hermitian positive-definite banded distributed matrix $A(1:n, ja:ja+n-1)$.

The resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form:

$A(1:n, ja:ja+n-1) = P * U^H * P^T$, if *uplo*='U', or

$A(1:n, ja:ja+n-1) = P * L * L^H * P^T$, if *uplo*='L',

where *P* is a permutation matrix and *U* and *L* are banded upper and lower triangular matrices, respectively.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', upper triangle of $A(1:n, ja:ja+n-1)$ is stored; If <i>uplo</i> = 'L', lower triangle of $A(1:n, ja:ja+n-1)$ is stored.
<i>n</i>	(global) INTEGER. The order of the distributed submatrix $A(1:n, ja:ja+n-1)$. ($n \geq 0$).
<i>bw</i>	(global) INTEGER. The number of superdiagonals of the distributed matrix if <i>uplo</i> = 'U', or the number of subdiagonals if <i>uplo</i> = 'L'. ($bw \geq 0$).
<i>a</i>	(local) REAL for pspbtrf DOUBLE PRECISION for pdpbtrf COMPLEX for pcgbtrf DOUBLE COMPLEX for pzpbtrf. Pointer into the local memory to an array of dimension $(lld\_a, LOCC(ja+n-1))$. On entry, this array contains the local pieces of the upper or lower triangle of the symmetric/Hermitian band distributed matrix $A(1:n, ja:ja+n-1)$ to be factored.
<i>ja</i>	(global) INTEGER. The index in the global array <i>A</i> that points to the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i>. If <i>desca</i>(<i>dtype_</i>) = 501, then <i>dlen_</i> $\geq$ 7; else if <i>desca</i>(<i>dtype_</i>) = 1, then <i>dlen_</i> $\geq$ 9.
<i>laf</i>	(local) INTEGER. The dimension of the array <i>af</i>. Must be <i>laf</i> $\geq$ (NB+2*<i>bw</i>) * <i>bw</i>. If <i>laf</i> is not large enough, an error code will be returned and the minimum acceptable size will be returned in <i>af</i>(1).

work (local) Same type as *a*. Workspace array of dimension *lwork*.

lwork (local or global) INTEGER. The size of the *work* array, must be $lwork \geq bw^2$.

Output Parameters

a On exit, if *info*=0, contains the permuted triangular factor *U* or *L* from the Cholesky factorization of the band matrix $A(1:n, ja:ja+n-1)$, as specified by *uplo*.

af (local)
 REAL for pspbtrf
 DOUBLE PRECISION for pdpbtrf
 COMPLEX for pcpbtrf
 DOUBLE COMPLEX for pzpbtrf.
 Array, dimension (*laf*). Auxiliary Fillin space. Fillin is created during the factorization routine *p?pbtrf* and this is stored in *af*. Note that if a linear system is to be solved using *p?pbtrs* after the factorization routine, *af* must not be altered.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info (global) INTEGER.
 If *info*=0, the execution is successful.
info < 0:
 If the *i*th argument is an array and the *j*th entry had an illegal value, then $info = -(i*100+j)$; if the *i*th argument is a scalar and had an illegal value, then $info = -i$.
info > 0:
 If $info = k \leq NPROCS$, the submatrix stored on processor *info* and factored locally was not positive definite, and the factorization was not completed.
 If $info = k > NPROCS$, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not nonsingular, and the factorization was not completed.

p?pttrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite tridiagonal distributed matrix.

Syntax

```
call pspttrf(n, d, e, ja, desca, af, laf, work, lwork, info)
call pdpttrf(n, d, e, ja, desca, af, laf, work, lwork, info)
call pcpttrf(n, d, e, ja, desca, af, laf, work, lwork, info)
call pzpttrf(n, d, e, ja, desca, af, laf, work, lwork, info)
```

Description

This routine computes the Cholesky factorization of an n -by- n real symmetric or complex hermitian positive-definite tridiagonal distributed matrix $A(1:n, ja:ja+n-1)$.

The resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form:

$$A(1:n, ja:ja+n-1) = P * L * D * L^H * P^T, \text{ or}$$

$$A(1:n, ja:ja+n-1) = P * U^H * D * U * P^T,$$

where P is a permutation matrix, and U and L are tridiagonal upper and lower triangular matrices, respectively.

Input Parameters

n	(global) INTEGER. The order of the distributed submatrix $A(1:n, ja:ja+n-1)$ ($n \geq 0$).
d, e	(local) REAL for pspttrf DOUBLE PRECISION for pdpttrf COMPLEX for pcpttrf DOUBLE COMPLEX for pzpttrf. Pointers into the local memory to arrays of dimension $(desca(nb\_))$ each.

	On entry, the array <i>d</i> contains the local part of the global vector storing the main diagonal of the distributed matrix <i>A</i> .
	On entry, the array <i>e</i> contains the local part of the global vector storing the upper diagonal of the distributed matrix <i>A</i> .
<i>ja</i>	(global) INTEGER. The index in the global array <i>A</i> that points to the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> . If <i>desca</i> (<i>dtype_</i>) = 501, then <i>dlen_</i> ≥ 7; else if <i>desca</i> (<i>dtype_</i>) = 1, then <i>dlen_</i> ≥ 9.
<i>laf</i>	(local) INTEGER. The dimension of the array <i>af</i> . Must be <i>laf</i> ≥ NB+2. If <i>laf</i> is not large enough, an error code will be returned and the minimum acceptable size will be returned in <i>af</i> (1).
<i>work</i>	(local) Same type as <i>d</i> and <i>e</i> . Workspace array of dimension <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER. The size of the <i>work</i> array, must be at least <i>lwork</i> ≥ 8*NPCOL.

Output Parameters

<i>d, e</i>	On exit, overwritten by the details of the factorization.
<i>af</i>	(local) REAL for pspttrf DOUBLE PRECISION for pdpttrf COMPLEX for pcpttrf DOUBLE COMPLEX for pzpttrf. Array, dimension (<i>laf</i>). Auxiliary Fillin space. Fillin is created during the factorization routine p?pttrf and this is stored in <i>af</i> . Note that if a linear system is to be solved using p?pttrs after the factorization routine, <i>af</i> must not be altered.

<code>work(1)</code>	On exit, <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) INTEGER. If <code>info=0</code>, the execution is successful. <code>info < 0</code>: If the <i>i</i>-th argument is an array and the <i>j</i>-th entry had an illegal value, then <code>info = -(i*100+j)</code>; if the <i>i</i>-th argument is a scalar and had an illegal value, then <code>info = -i</code>. <code>info > 0</code>: If <code>info = k ≤ NPROCS</code>, the submatrix stored on processor <code>info</code> and factored locally was not positive definite, and the factorization was not completed. If <code>info = k > NPROCS</code>, the submatrix stored on processor <code>info-NPROCS</code> representing interactions with other processors was not nonsingular, and the factorization was not completed.

p?dttrf

Computes the LU factorization of a diagonally dominant-like tridiagonal distributed matrix.

Syntax

```
call psdttrf(n, dl, d, du, ja, desca, af, laf, work, lwork, info)
call pddttrf(n, dl, d, du, ja, desca, af, laf, work, lwork, info)
call pcdttrf(n, dl, d, du, ja, desca, af, laf, work, lwork, info)
call pzdtttrf(n, dl, d, du, ja, desca, af, laf, work, lwork, info)
```

Description

This routine computes the LU factorization of an *n*-by-*n* real/complex diagonally dominant-like tridiagonal distributed matrix `A(1:n, ja:ja+n-1)` without pivoting for stability.

The resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form:

$$A(1:n, ja:ja+n-1) = P*L*U*P^T,$$

where P is a permutation matrix, and L and U are banded lower and upper triangular matrices, respectively.

Input Parameters

n (global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix $A(1:n, ja:ja+n-1)$ ($n \geq 0$).

dl, d, du (local)
 REAL for pspttrf
 DOUBLE PRECISION for pdpttrf
 COMPLEX for pcpttrf
 DOUBLE COMPLEX for pzpttrf.
 Pointers to the local arrays of dimension $(desca(nb\_))$ each.
 On entry, the array *dl* contains the local part of the global vector storing the subdiagonal elements of the matrix. Globally, *dl*(1) is not referenced, and *dl* must be aligned with *d*.
 On entry, the array *d* contains the local part of the global vector storing the diagonal elements of the matrix.
 On entry, the array *du* contains the local part of the global vector storing the super-diagonal elements of the matrix. *du*(*n*) is not referenced, and *du* must be aligned with *d*.

ja (global) INTEGER. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) INTEGER array, dimension $(dlen\_)$. The array descriptor for the distributed matrix *A*. If $desca(dtype\_)$ = 501, then $dlen\_ \geq 7$;
 else if $desca(dtype\_)$ = 1, then $dlen\_ \geq 9$.

laf (local) INTEGER. The dimension of the array *af*.
 Must be $laf \geq 2*(NB+2)$.
 If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

work (local) Same type as *d*. Workspace array of dimension *lwork*.

lwork (local or global) INTEGER. The size of the *work* array, must be at least $lwork \geq 8 * NPCOL$.

Output Parameters

dl, d, du On exit, overwritten by the information containing the factors of the matrix.

af (local)
 REAL for psdtttrf
 DOUBLE PRECISION for pddtttrf
 COMPLEX for pcdtttrf
 DOUBLE COMPLEX for pzdtttrf.
 Array, dimension (*laf*).
 Auxiliary Fillin space. Fillin is created during the factorization routine p?dtttrf and this is stored in *af*.
 Note that if a linear system is to be solved using p?dtttrs after the factorization routine, *af* must not be altered.

work(1) On exit, *work(1)* contains the minimum value of *lwork* required for optimum performance.

info (global) INTEGER.
 If *info*=0, the execution is successful.
info < 0:
 If the *i*-th argument is an array and the *j*-th entry had an illegal value, then $info = -(i*100+j)$; if the *i*-th argument is a scalar and had an illegal value, then $info = -i$.
info > 0:
 If $info = k \leq NPCOCS$, the submatrix stored on processor *info* and factored locally was not diagonally dominant-like, and the factorization was not completed. If $info = k > NPCOCS$, the submatrix stored on processor *info*-NPCOCS representing interactions with other processors was not nonsingular, and the factorization was not completed.

Routines for Solving Systems of Linear Equations

This section describes the ScaLAPACK routines for solving systems of linear equations. Before calling most of these routines, you need to factorize the matrix of your system of equations (see [Routines for Matrix Factorization](#) in this chapter). However, the factorization is not necessary if your system of equations has a triangular matrix.

p?getrs

Solves a system of distributed linear equations with a general square matrix, using the LU factorization computed by p?getrf.

Syntax

```
call psgetrs(trans, n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
call pdgetrs(trans, n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
call pcgetrs(trans, n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
call pzgetrs(trans, n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
```

Description

This routine solves a system of distributed linear equations with a general n -by- n distributed matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ using the LU factorization computed by p?getrf.

The system has one of the following forms specified by *trans*:

$\text{sub}(A) * X = \text{sub}(B)$ (no transpose),

$\text{sub}(A)^T * X = \text{sub}(B)$ (transpose),

$\text{sub}(A)^H * X = \text{sub}(B)$ (conjugate transpose),

where $\text{sub}(B) = B(ib:ib+n-1, jb:jb+nrhs-1)$.

Before calling this routine, you must call p?getrf to compute the LU factorization of $\text{sub}(A)$.

Input Parameters

trans (global) CHARACTER*1. Must be 'N' or 'T' or 'C'.
Indicates the form of the equations:
If *trans* = 'N', then $\text{sub}(A) * X = \text{sub}(B)$ is solved for X .

	If $trans = 'T'$, then $sub(A)^T * X = sub(B)$ is solved for X. If $trans = 'C'$, then $sub(A)^H * X = sub(B)$ is solved for X.
n	(global) INTEGER. The number of linear equations; the order of the submatrix $sub(A)$ ($n \geq 0$).
$nrhs$	(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix $sub(B)$ ($nrhs \geq 0$).
a, b	(global) REAL for psgetrs DOUBLE PRECISION for pdgetrs COMPLEX for pcgetrs DOUBLE COMPLEX for pzgetrs. Pointers into the local memory to arrays of local dimension $a(lld\_a, LOCc(ja+n-1))$ and $b(lld\_b, LOCc(jb+nrhs-1))$, respectively. On entry, the array a contains the local pieces of the factors L and U from the factorization $sub(A) = P * L * U$; the unit diagonal elements of L are not stored. On entry, the array b contains the right hand sides $sub(B)$.
ia, ja	(global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of the submatrix $sub(A)$, respectively.
$desca$	(global and local) INTEGER array, dimension ($dlen\_$). The array descriptor for the distributed matrix A .
$ipiv$	(local) INTEGER array. The dimension of $ipiv$ is $(LOCr(m\_a) + mb\_a)$. This array contains the pivoting information: local row i of the matrix was interchanged with the global row $ipiv(i)$. This array is tied to the distributed matrix A.
ib, jb	(global) INTEGER. The row and column indices in the global array B indicating the first row and the first column of the submatrix $sub(B)$, respectively.

descb (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

Output Parameters

b On exit, overwritten by the solution distributed matrix *x*.
info INTEGER. If *info*=0, the execution is successful. *info* < 0:
 If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = $-(i*100+j)$; if the *i*-th argument is a scalar and had an illegal value, then *info* = $-i$.

p?gbtrs

Solves a system of distributed linear equations with a general band matrix, using the LU factorization computed by p?gbtrf.

Syntax

```
call psgbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, af,
laf, work, lwork, info)
```

```
call pdgbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, af,
laf, work, lwork, info)
```

```
call pcgbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, af,
laf, work, lwork, info)
```

```
call pzgbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, af,
laf, work, lwork, info)
```

Description

This routine solves a system of distributed linear equations with a general band distributed matrix $\text{sub}(A) = A(1:n, ja:ja+n-1)$ using the LU factorization computed by p?gbtrf.

The system has one of the following forms specified by *trans*:

$\text{sub}(A) * X = \text{sub}(B)$ (no transpose),

$\text{sub}(A)^T * X = \text{sub}(B)$ (transpose),

$\text{sub}(A)^H * X = \text{sub}(B)$ (conjugate transpose),

where $\text{sub}(B) = B(ib:ib+n-1, 1:nrhs)$.

Before calling this routine, you must call [p?gbtrf](#) to compute the *LU* factorization of $\text{sub}(A)$.

Input Parameters

<i>trans</i>	(global) CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', then $\text{sub}(A) * X = \text{sub}(B)$ is solved for <i>X</i>. If <i>trans</i> = 'T', then $\text{sub}(A)^T * X = \text{sub}(B)$ is solved for <i>X</i>. If <i>trans</i> = 'C', then $\text{sub}(A)^H * X = \text{sub}(B)$ is solved for <i>X</i>.
<i>n</i>	(global) INTEGER. The number of linear equations; the order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).
<i>bwl</i>	(global) INTEGER. The number of sub-diagonals within the band of <i>A</i> ($0 \leq bwl \leq n-1$).
<i>bwu</i>	(global) INTEGER. The number of super-diagonals within the band of <i>A</i> ($0 \leq bwu \leq n-1$).
<i>nrhs</i>	(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix $\text{sub}(B)$ ($nrhs \geq 0$).
<i>a, b</i>	(global) REAL for psgbtrs DOUBLE PRECISION for pdgbtrs COMPLEX for pcgbtrs DOUBLE COMPLEX for pzgbtrs. Pointers into the local memory to arrays of local dimension $a(lld\_a, LOC_c(ja+n-1))$ and $b(lld\_b, LOC_c(nrhs))$, respectively. The array <i>a</i> contains details of the <i>LU</i> factorization of the distributed band matrix <i>A</i>. On entry, the array <i>b</i> contains the local pieces of the right hand sides $B(ib:ib+n-1, 1:nrhs)$.

<i>ja</i>	(global) INTEGER. The index in the global array <i>a</i> that points to the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> . If <i>desca</i> (<i>dtype_</i>) = 501, then <i>dlen_</i> ≥ 7; else if <i>desca</i> (<i>dtype_</i>) = 1, then <i>dlen_</i> ≥ 9.
<i>ib</i>	(global) INTEGER. The index in the global array <i>a</i> that points to the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>descb</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> . If <i>desca</i> (<i>dtype_</i>) = 501, then <i>dlen_</i> ≥ 7; else if <i>desca</i> (<i>dtype_</i>) = 1, then <i>dlen_</i> ≥ 9.
<i>laf</i>	(local) INTEGER. The dimension of the array <i>af</i> . Must be $laf \geq NB \cdot (bwl + bwu) + 6 \cdot (bwl + bwu) \cdot (bwl + 2 \cdot bwu)$. If <i>laf</i> is not large enough, an error code will be returned and the minimum acceptable size will be returned in <i>af</i> (1).
<i>work</i>	(local) Same type as <i>a</i> . Workspace array of dimension <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER. The size of the <i>work</i> array, must be at least $lwork \geq nrhs \cdot (NB + 2 \cdot bwl + 4 \cdot bwu)$.

Output Parameters

<i>ipiv</i>	(local) INTEGER array. The dimension of <i>ipiv</i> must be ≥ <i>desca</i> (NB). Contains pivot indices for local factorizations. Note that you should not alter the contents of this array between factorization and solve.
<i>b</i>	On exit, overwritten by the local pieces of the solution distributed matrix <i>x</i> .
<i>af</i>	(local) REAL for psgbtrs DOUBLE PRECISION for pdgbtrs

COMPLEX for p_{cg}btrs
DOUBLE COMPLEX for p_zgbtrs.
Array, dimension (*laf*).
Auxiliary Fillin space. Fillin is created during the factorization routine p_?gbtrf and this is stored in *af*.
Note that if a linear system is to be solved using p_?gbtrs after the factorization routine, *af* must not be altered after the factorization.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info INTEGER. If *info*=0, the execution is successful.
info < 0:
If the *i*-th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?potrs

Solves a system of linear equations with a Cholesky-factored symmetric/Hermitian distributed positive-definite matrix.

Syntax

```
call pspotrs(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pdpotrs(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pcpotrs(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pzpotsrs(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
```

Description

The routine p_?potrs solves for *x* a system of distributed linear equations in the form:

$$\text{sub}(A) * X = \text{sub}(B) ,$$

where $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ is an *n*-by-*n* real symmetric or complex Hermitian positive definite distributed matrix, and $\text{sub}(B)$ denotes the distributed matrix $B(ib:ib+n-1, jb:jb+nrhs-1)$.

This routine uses Cholesky factorization

$$\text{sub}(A) = U^H * U, \text{ or } \text{sub}(A) = L * L^H$$

computed by `p?potrf`.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', upper triangle of sub(<i>A</i>) is stored; If <i>uplo</i> = 'L', lower triangle of sub(<i>A</i>) is stored.
<i>n</i>	(global) INTEGER. The order of the distributed submatrix sub(<i>A</i>) ($n \geq 0$).
<i>nrhs</i>	(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix sub(<i>B</i>) ($nrhs \geq 0$).
<i>a, b</i>	(local) REAL for pspotrs DOUBLE PRECISION for pdpotrs COMPLEX for pcpotrs DOUBLE COMPLEX for pzpotrs. Pointers into the local memory to arrays of local dimension <i>a</i> (<i>lld_a</i> , <i>LOCc</i> (<i>ja</i> + <i>n</i> -1)) and <i>b</i> (<i>lld_b</i> , <i>LOCc</i> (<i>jb</i> + <i>nrhs</i> -1)), respectively. The array <i>a</i> contains the factors <i>L</i> or <i>U</i> from the Cholesky factorization $\text{sub}(A) = L * L^H$ or $\text{sub}(A) = U^H * U$, as computed by <code>p?potrf</code> . On entry, the array <i>b</i> contains the local pieces of the right hand sides sub(<i>B</i>).
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>A</i> indicating the first row and the first column of the submatrix sub(<i>A</i>), respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the global array <i>B</i> indicating the first row and the first column of the submatrix sub(<i>B</i>), respectively.

descb (global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

Output Parameters

b Overwritten by the local pieces of the solution matrix *X*.

info `INTEGER`. If *info*=0, the execution is successful.
info < 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?pbtrs

Solves a system of linear equations with a Cholesky-factored symmetric/Hermitian positive-definite band matrix.

Syntax

```
call pspbtrs( uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf, work,
             lwork, info)

call pdpbtrs( uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf, work,
             lwork, info)

call pcpbtrs( uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf, work,
             lwork, info)

call pzpbttrs( uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf, work,
              lwork, info)
```

Description

The routine `p?pbtrs` solves for *X* a system of distributed linear equations in the form:

$$\text{sub}(A) * X = \text{sub}(B) ,$$

where $\text{sub}(A) = A(1:n, ja:ja+n-1)$ is an *n*-by-*n* real symmetric or complex Hermitian positive definite distributed band matrix, and $\text{sub}(B)$ denotes the distributed matrix $B(ib:ib+n-1, 1:nrhs)$.

This routine uses Cholesky factorization

$\text{sub}(A) = P*U^H*U*P^T$, or $\text{sub}(A) = P*L*L^H*P^T$

computed by `p?pbtrf`.

Input Parameters

uplo (global) CHARACTER*1. Must be 'U' or 'L'.
 If *uplo* = 'U', upper triangle of $\text{sub}(A)$ is stored;
 If *uplo* = 'L', lower triangle of $\text{sub}(A)$ is stored.

n (global) INTEGER. The order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).

bw (global) INTEGER. The number of superdiagonals of the distributed matrix if *uplo* = 'U', or the number of subdiagonals if *uplo* = 'L' ($bw \geq 0$).

nrhs (global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix $\text{sub}(B)$ ($nrhs \geq 0$).

a, b (local)
 REAL for `pspbtrs`
 DOUBLE PRECISION for `pdpbtrs`
 COMPLEX for `pcpbtrs`
 DOUBLE COMPLEX for `pzpbtrs`.
 Pointers into the local memory to arrays of local dimension $a(lld\_a, LOCC(ja+n-1))$ and $b(lld\_b, LOCC(nrhs-1))$, respectively.
 The array *a* contains the permuted triangular factor *U* or *L* from the Cholesky factorization $\text{sub}(A) = P*U^H*U*P^T$, or $\text{sub}(A) = P*L*L^H*P^T$ of the band matrix *A*, as returned by `p?pbtrf`.
 On entry, the array *b* contains the local pieces of the *n*-by-*nrhs* right hand side distributed matrix $\text{sub}(B)$.

ja (global) INTEGER. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
 If `desca(dtype_) = 501`, then $dlen_ \geq 7$;

	else if <code>desca(dtype_) = 1</code> , then $dlen_ \geq 9$.
<code>ib</code>	(global) INTEGER. The row index in the global array <i>B</i> indicating the first row of the submatrix <code>sub(B)</code> .
<code>descb</code>	(global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix <i>B</i> . If <code>descb(dtype_) = 502</code> , then $dlen_ \geq 7$; else if <code>descb(dtype_) = 1</code> , then $dlen_ \geq 9$.
<code>af, work</code>	(local) Arrays, same type as <i>a</i> . The array <i>af</i> is of dimension (laf). It contains auxiliary Fillin space. Fillin is created during the factorization routine <code>p?dbtrf</code> and this is stored in <i>af</i> . The array <i>work</i> is a workspace array of dimension $lwork$.
<code>laf</code>	(local) INTEGER. The dimension of the array <i>af</i> . Must be $laf \geq nrhs*bw$. If laf is not large enough, an error code will be returned and the minimum acceptable size will be returned in <code>af(1)</code> .
<code>lwork</code>	(local or global) INTEGER. The size of the array <i>work</i> , must be at least $lwork \geq bw^2$.

Output Parameters

<code>b</code>	On exit, if <code>info=0</code> , this array contains the local pieces of the <i>n</i> -by- <i>nrhs</i> solution distributed matrix <i>X</i> .
<code>work(1)</code>	On exit, <code>work(1)</code> contains the minimum value of $lwork$ required for optimum performance.
<code>info</code>	INTEGER. If <code>info=0</code> , the execution is successful. <code>info < 0</code> : If the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <code>info = -(i*100+j)</code> ; if the <i>i</i> -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

p?pttrs

Solves a system of linear equations with a symmetric (Hermitian) positive-definite tridiagonal distributed matrix using the factorization computed by p?pttrf.

Syntax

```
call pspttrs(n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work, lwork,
info)

call pdpttrs(n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work, lwork,
info)

call pcpttrs(uplo, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work,
lwork, info)

call pzpttrs(uplo, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work,
lwork, info)
```

Description

The routine p?pttrs solves for X a system of distributed linear equations in the form:

$$\text{sub}(A) * X = \text{sub}(B) ,$$

where $\text{sub}(A) = A(1:n, ja:ja+n-1)$ is an n -by- n real symmetric or complex Hermitian positive definite tridiagonal distributed matrix, and $\text{sub}(B)$ denotes the distributed matrix $B(ib:ib+n-1, 1:nrhs)$.

This routine uses the factorization

$$\text{sub}(A) = P * L * D * L^H * P^T, \text{ or } \text{sub}(A) = P * U^H * D * U * P^T$$

computed by p?pttrf.

Input Parameters

<i>uplo</i>	(global, used in complex flavors only) CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', upper triangle of $\text{sub}(A)$ is stored; If <i>uplo</i> = 'L', lower triangle of $\text{sub}(A)$ is stored.
<i>n</i>	(global) INTEGER. The order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).

<i>nrhs</i>	(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix sub(<i>B</i>) (<i>nrhs</i> ≥0).
<i>d, e</i>	(local) REAL for pspttrs DOUBLE PRECISION for pdpttrs COMPLEX for pcpttrs DOUBLE COMPLEX for pzpttrs. Pointers into the local memory to arrays of dimension (<i>desca</i> (<i>nb_</i>)) each. These arrays contain details of the factorization as returned by p?pttrf
<i>ja</i>	(global) INTEGER. The index in the global array <i>A</i> that points to the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> . If <i>desca</i> (<i>dtype_</i>) = 501 or 502, then <i>dlen_</i> ≥ 7; else if <i>desca</i> (<i>dtype_</i>) = 1, then <i>dlen_</i> ≥ 9.
<i>b</i>	(local) Same type as <i>d, e</i> . Pointer into the local memory to an array of local dimension <i>b</i> (<i>lld_b</i> , <i>LOCc</i> (<i>nrhs</i>)). On entry, the array <i>b</i> contains the local pieces of the <i>n</i> -by- <i>nrhs</i> right hand side distributed matrix sub(<i>B</i>).
<i>ib</i>	(global) INTEGER. The row index in the global array <i>B</i> that points to the first row of the matrix to be operated on (which may be either all of <i>B</i> or a submatrix of <i>B</i>).
<i>descb</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>B</i> . If <i>descb</i> (<i>dtype_</i>) = 502, then <i>dlen_</i> ≥ 7; else if <i>descb</i> (<i>dtype_</i>) = 1, then <i>dlen_</i> ≥ 9.
<i>af, work</i>	(local) REAL for pspttrs DOUBLE PRECISION for pdpttrs COMPLEX for pcpttrs DOUBLE COMPLEX for pzpttrs.

Arrays of dimension (*laf*) and (*lwork*), respectively. The array *af* contains auxiliary Fillin space. Fillin is created during the factorization routine `p?pttrf` and this is stored in *af*.

The array *work* is a workspace array.

laf (local) INTEGER. The dimension of the array *af*.

Must be $laf \geq NB+2$.

If *laf* is not large enough, an error code is returned and the minimum acceptable size will be returned in *af*(1).

lwork (local or global) INTEGER. The size of the array *work*, must be at least

$lwork \geq (10+2*\min(100,nrhs))*NPCOL+4*nrhs$.

Output Parameters

b On exit, this array contains the local pieces of the solution distributed matrix *X*.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info INTEGER. If *info*=0, the execution is successful.
info < 0:
 if the *i*-th argument is an array and the *j*-th entry had an illegal value, then $info = -(i*100+j)$;
 if the *i*-th argument is a scalar and had an illegal value, then $info = -i$.

p?dttrs

Solves a system of linear equations with a diagonally dominant-like tridiagonal distributed matrix using the factorization computed by p?dttrf.

Syntax

```
call psdttrs(trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af, laf,
work, lwork, info)

call pddttrs(trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af, laf,
work, lwork, info)

call pcdttrs(trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af, laf,
work, lwork, info)

call pzdttrs(trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af, laf,
work, lwork, info)
```

Description

The routine p?dttrs solves for X one of the systems of equations:

$$\text{sub}(A) * X = \text{sub}(B),$$

$$(\text{sub}(A))^T * X = \text{sub}(B), \text{ or}$$

$$(\text{sub}(A))^H * X = \text{sub}(B),$$

where $\text{sub}(A) = (1:n, ja:ja+n-1)$; is a diagonally dominant-like tridiagonal distributed matrix, and $\text{sub}(B)$ denotes the distributed matrix $B(ib:ib+n-1, 1:nrhs)$.

This routine uses the LU factorization computed by p?dttrf.

Input Parameters

trans (global) CHARACTER*1. Must be 'N' or 'T' or 'C'.
Indicates the form of the equations:
If *trans* = 'N', then $\text{sub}(A) * X = \text{sub}(B)$ is solved for X .
If *trans* = 'T', then $(\text{sub}(A))^T * X = \text{sub}(B)$ is solved for X .
If *trans* = 'C', then $(\text{sub}(A))^H * X = \text{sub}(B)$ is solved for X .

<i>n</i>	(global) INTEGER. The order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).
<i>nrhs</i>	(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix $\text{sub}(B)$ ($nrhs \geq 0$).
<i>dl, d, du</i>	(local) REAL for psdttrs DOUBLE PRECISION for pddttrs COMPLEX for pcdttrs DOUBLE COMPLEX for pzdttrs. Pointers to the local arrays of dimension $(\text{desca}(nb\_))$ each. On entry, these arrays contain details of the factorization. Globally, $dl(1)$ and $du(n)$ are not referenced; dl and du must be aligned with d .
<i>ja</i>	(global) INTEGER. The index in the global array A that points to the start of the matrix to be operated on (which may be either all of A or a submatrix of A).
<i>desca</i>	(global and local) INTEGER array, dimension $(dlen\_)$. The array descriptor for the distributed matrix A . If $\text{desca}(dtype_) = 501$ or 502 , then $dlen_ \geq 7$; else if $\text{desca}(dtype_) = 1$, then $dlen_ \geq 9$.
<i>b</i>	(local) Same type as d . Pointer into the local memory to an array of local dimension $b(lld\_b, LOCC(nrhs))$. On entry, the array b contains the local pieces of the n -by- $nrhs$ right hand side distributed matrix $\text{sub}(B)$.
<i>ib</i>	(global) INTEGER. The row index in the global array B that points to the first row of the matrix to be operated on (which may be either all of B or a submatrix of B).
<i>descb</i>	(global and local) INTEGER array, dimension $(dlen\_)$. The array descriptor for the distributed matrix B . If $\text{descb}(dtype_) = 502$, then $dlen_ \geq 7$; else if $\text{descb}(dtype_) = 1$, then $dlen_ \geq 9$.

<i>af, work</i>	(local) REAL for psdtttrs DOUBLE PRECISION for pddtttrs COMPLEX for pcdtttrs DOUBLE COMPLEX for pzdtttrs. Arrays of dimension (<i>laf</i>) and (<i>lwork</i>), respectively. The array <i>af</i> contains auxiliary Fillin space. Fillin is created during the factorization routine p?dttrf and this is stored in <i>af</i>. If a linear system is to be solved using p?dttrs after the factorization routine, <i>af</i> must not be altered. The array <i>work</i> is a workspace array.
<i>laf</i>	(local) INTEGER. The dimension of the array <i>af</i>. Must be $laf \geq NB * (bwl + bwu) + 6 * (bwl + bwu) * (bwl + 2 * bwu)$. If <i>laf</i> is not large enough, an error code will be returned and the minimum acceptable size will be returned in <i>af</i>(1).
<i>lwork</i>	(local or global) INTEGER. The size of the array <i>work</i>, must be at least $lwork \geq 10 * NPCOL + 4 * nrhs$.

Output Parameters

<i>b</i>	On exit, this array contains the local pieces of the solution distributed matrix <i>x</i> .
<i>work</i> (1)	On exit, <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	INTEGER. If <i>info</i>=0, the execution is successful. <i>info</i> < 0: if the <i>i</i>th argument is an array and the <i>j</i>-th entry had an illegal value, then <i>info</i> = -(<i>i</i>*100+<i>j</i>); if the <i>i</i>-th argument is a scalar and had an illegal value, then <i>info</i> = -<i>i</i>.

p?dbtrs

Solves a system of linear equations with a diagonally dominant-like banded distributed matrix using the factorization computed by p?dbtrf.

Syntax

```
call psdbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af, laf,
work, lwork, info)

call pddbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af, laf,
work, lwork, info)

call pcdbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af, laf,
work, lwork, info)

call pzdbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af, laf,
work, lwork, info)
```

Description

The routine p?dbtrs solves for X one of the systems of equations:

$$\text{sub}(A) * X = \text{sub}(B),$$

$$(\text{sub}(A))^T * X = \text{sub}(B), \text{ or}$$

$$(\text{sub}(A))^H * X = \text{sub}(B),$$

where $\text{sub}(A) = A(1:n, ja:ja+n-1)$ is a diagonally dominant-like banded distributed matrix, and $\text{sub}(B)$ denotes the distributed matrix $B(ib:ib+n-1, 1:nrhs)$.

This routine uses the LU factorization computed by p?dbtrf.

Input Parameters

trans (global) CHARACTER*1. Must be 'N' or 'T' or 'C'.
 Indicates the form of the equations:
 If *trans* = 'N', then $\text{sub}(A) * X = \text{sub}(B)$ is solved for X .
 If *trans* = 'T', then $(\text{sub}(A))^T * X = \text{sub}(B)$ is solved for X .
 If *trans* = 'C', then $(\text{sub}(A))^H * X = \text{sub}(B)$ is solved for X .

<i>n</i>	(global) INTEGER. The order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).
<i>bwl</i>	(global) INTEGER. The number of subdiagonals within the band of A ($0 \leq bwl \leq n-1$).
<i>bwu</i>	(global) INTEGER. The number of superdiagonals within the band of A ($0 \leq bwu \leq n-1$).
<i>nrhs</i>	(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix $\text{sub}(B)$ ($nrhs \geq 0$).
<i>a, b</i>	(local) REAL for psdbtrs DOUBLE PRECISION for pddbtrs COMPLEX for pcdbtrs DOUBLE COMPLEX for pzdbtrs. Pointers into the local memory to arrays of local dimension $a(lld\_a, LOCC(ja+n-1))$ and $b(lld\_b, LOCC(nrhs))$, respectively. On entry, the array <i>a</i> contains details of the <i>LU</i> factorization of the band matrix A , as computed by p?dbtrf. On entry, the array <i>b</i> contains the local pieces of the right hand side distributed matrix $\text{sub}(B)$.
<i>ja</i>	(global) INTEGER. The index in the global array A that points to the start of the matrix to be operated on (which may be either all of A or a submatrix of A).
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix A . If $desca(dtype\_)$ = 501, then $dlen\_ \geq 7$; else if $desca(dtype\_)$ = 1, then $dlen\_ \geq 9$.
<i>ib</i>	(global) INTEGER. The row index in the global array B that points to the first row of the matrix to be operated on (which may be either all of B or a submatrix of B).

<i>descb</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>B</i> . If <i>descb(dtype_)</i> = 502, then <i>dlen_</i> ≥ 7; else if <i>descb(dtype_)</i> = 1, then <i>dlen_</i> ≥ 9.
<i>af, work</i>	(local) REAL for psdbtrs DOUBLE PRECISION for pddbtrs COMPLEX for pcdbtrs DOUBLE COMPLEX for pzdbtrs. Arrays of dimension (<i>laf</i>) and (<i>lwork</i>), respectively The array <i>af</i> contains auxiliary Fillin space. Fillin is created during the factorization routine <i>p?dbtrf</i> and this is stored in <i>af</i> . The array <i>work</i> is a workspace array.
<i>laf</i>	(local) INTEGER. The dimension of the array <i>af</i> . Must be $laf \geq NB * (bwl + bwu) + 6 * (\max(bwl, bwu))^2$. If <i>laf</i> is not large enough, an error code will be returned and the minimum acceptable size will be returned in <i>af</i> (1).
<i>lwork</i>	(local or global) INTEGER. The size of the array <i>work</i> , must be at least $lwork \geq (\max(bwl, bwu))^2$.

Output Parameters

<i>b</i>	On exit, this array contains the local pieces of the solution distributed matrix <i>X</i> .
<i>work</i> (1)	On exit, <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. <i>info</i> < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <i>info</i> = $-(i*100+j)$; if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = $-i$.

p?trtrs

Solves a system of linear equations with a triangular distributed matrix.

Syntax

```
call pstrtrs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb,
info)

call pdtrtrs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb,
info)

call pctrtrs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb,
info)

call pztrtrs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb,
info)
```

Description

This routine solves for x one of the following systems of linear equations:

$$\text{sub}(A) * X = \text{sub}(B),$$

$$(\text{sub}(A))^T * X = \text{sub}(B), \text{ or}$$

$$(\text{sub}(A))^H * X = \text{sub}(B),$$

where $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ is a triangular distributed matrix of order n , and $\text{sub}(B)$ denotes the distributed matrix $B(ib:ib+n-1, jb:jb+nrhs-1)$.

A check is made to verify that $\text{sub}(A)$ is nonsingular.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Must be 'U' or 'L'. Indicates whether $\text{sub}(A)$ is upper or lower triangular: If <i>uplo</i> = 'U', then $\text{sub}(A)$ is upper triangular. If <i>uplo</i> = 'L', then $\text{sub}(A)$ is lower triangular.
<i>trans</i>	(global) CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', then $\text{sub}(A) * X = \text{sub}(B)$ is solved for x . If <i>trans</i> = 'T', then $\text{sub}(A)^T * X = \text{sub}(B)$ is solved for x . If <i>trans</i> = 'C', then $\text{sub}(A)^H * X = \text{sub}(B)$ is solved for x .

If $trans = 'C'$, then $sub(A)^H X = sub(B)$ is solved for X .

diag (global) CHARACTER*1. Must be 'N' or 'U'.
 If $diag = 'N'$, then $sub(A)$ is not a unit triangular matrix.
 If $diag = 'U'$, then $sub(A)$ is unit triangular.

n (global) INTEGER. The order of the distributed submatrix $sub(A)$ ($n \geq 0$).

nrhs (global) INTEGER. The number of right-hand sides; i.e., the number of columns of the distributed matrix $sub(B)$ ($nrhs \geq 0$).

a, b (local)
 REAL for pstrtrs
 DOUBLE PRECISION for pdtrtrs
 COMPLEX for pctrtrs
 DOUBLE COMPLEX for pztrtrs.
 Pointers into the local memory to arrays of local dimension $a(lld\_a, LOCC(ja+n-1))$ and $b(lld\_b, LOCC(jb+nrhs-1))$, respectively.
 The array a contains the local pieces of the distributed triangular matrix $sub(A)$.
 If $uplo = 'U'$, the leading n -by- n upper triangular part of $sub(A)$ contains the upper triangular matrix, and the strictly lower triangular part of $sub(A)$ is not referenced.
 If $uplo = 'L'$, the leading n -by- n lower triangular part of $sub(A)$ contains the lower triangular matrix, and the strictly upper triangular part of $sub(A)$ is not referenced.
 If $diag = 'U'$, the diagonal elements of $sub(A)$ are also not referenced and are assumed to be 1.
 On entry, the array b contains the local pieces of the right hand side distributed matrix $sub(B)$.

ia, ja (global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of the submatrix $sub(A)$, respectively.

desca (global and local) INTEGER array, dimension ($dlen\_$). The array descriptor for the distributed matrix A .

<i>ib, jb</i>	(global) <code>INTEGER</code> . The row and column indices in the global array <i>B</i> indicating the first row and the first column of the submatrix <code>sub(B)</code> , respectively.
<i>descb</i>	(global and local) <code>INTEGER</code> array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>B</i> .

Output Parameters

<i>b</i>	On exit, if <i>info</i> =0, <code>sub(B)</code> is overwritten by the solution matrix <i>X</i> .
<i>info</i>	<code>INTEGER</code> . If <i>info</i> =0, the execution is successful. <i>info</i> < 0: if the <i>i</i> th argument is an array and the <i>j</i> th entry had an illegal value, then <i>info</i> = -(<i>i</i> *100+ <i>j</i>); if the <i>i</i> th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> ; <i>info</i> > 0: if <i>info</i> = <i>i</i> , the <i>i</i> -th diagonal element of <code>sub(A)</code> is zero, indicating that the submatrix is singular and the solutions <i>X</i> have not been computed.

Routines for Estimating the Condition Number

This section describes the ScaLAPACK routines for estimating the condition number of a matrix. The condition number is used for analyzing the errors in the solution of a system of linear equations. Since the condition number may be arbitrarily large when the matrix is nearly singular, the routines actually compute the *reciprocal* condition number.

p?gecon

Estimates the reciprocal of the condition number of a general distributed matrix in either the 1-norm or the infinity-norm.

Syntax

```
call psgecon( norm, n, a, ia, ja, desca, anorm, rcond, work, lwork, iwork,
             liwork, info)

call pdgecon( norm, n, a, ia, ja, desca, anorm, rcond, work, lwork, iwork,
             liwork, info)

call pcgecon( norm, n, a, ia, ja, desca, anorm, rcond, work, lwork, rwork,
             lrwork, info)

call pzgecon( norm, n, a, ia, ja, desca, anorm, rcond, work, lwork, rwork,
             lrwork, info)
```

Description

This routine estimates the reciprocal of the condition number of a general distributed real/complex matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ in either the 1-norm or infinity-norm, using the *LU* factorization computed by [p?getrf](#).

An estimate is obtained for $\|(\text{sub}(A))^{-1}\|$, and the reciprocal of the condition number is computed as

$$rcond = \frac{1}{\|\text{sub}(A)\| \times \|(\text{sub}(A))^{-1}\|}$$

Input Parameters

norm (global) CHARACTER*1. Must be '1' or 'O' or 'I'.
 Specifies whether the 1-norm condition number or the infinity-norm condition number is required.
 If *norm* = '1' or 'O', then the 1-norm is used;
 If *norm* = 'I', then the infinity-norm is used.

<i>n</i>	(global) INTEGER. The order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).
<i>a</i>	(local) REAL for psgecon DOUBLE PRECISION for pdgecon COMPLEX for pcgecon DOUBLE COMPLEX for pzgecon. Pointer into the local memory to an array of dimension $a(lld\_a, LOCC(ja+n-1))$. The array <i>a</i> contains the local pieces of the factors <i>L</i> and <i>U</i> from the factorization $\text{sub}(A) = P * L * U$; the unit diagonal elements of <i>L</i> are not stored.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>A</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>anorm</i>	(global) REAL for single precision flavors, DOUBLE PRECISION for double precision flavors. If <i>norm</i> = '1' or 'O', the 1-norm of the original distributed matrix $\text{sub}(A)$; If <i>norm</i> = 'I', the infinity-norm of the original distributed matrix $\text{sub}(A)$.
<i>work</i>	(local) REAL for psgecon DOUBLE PRECISION for pdgecon COMPLEX for pcgecon DOUBLE COMPLEX for pzgecon. The array <i>work</i> of dimension (<i>lwork</i>) is a workspace array.
<i>lwork</i>	(local or global) INTEGER. The dimension of the array <i>work</i> . For real flavors: <i>lwork</i> must be at least

$$lwork \geq 2*LOCr(n+mod(ia-1,mb\_a))+$$

$$2*LOCc(n+mod(ja-1,nb\_a))+max(2, max(nb\_a*max(1,$$

$$ceil(NPROW-1, NPCOL)),$$

$$LOCc(n+mod(ja-1,nb\_a))+nb\_a*max(1, ceil(NPCOL-1,$$

$$NPROW))).$$

For complex flavors:

lwork must be at least

$$lwork \geq 2*LOCr(n+mod(ia-1,mb\_a))+max(2,$$

$$max(nb\_a*ceil(NPROW-1, NPCOL),$$

$$LOCc(n+mod(ja-1,nb\_a))+ nb\_a*ceil(NPCOL-1,$$

$$NPROW))).$$

LOCr and *LOCc* values can be computed using the ScaLAPACK tool function `numroc`; *NPROW* and *NPCOL* can be determined by calling the subroutine `blacs_gridinfo`.

iwork (local) INTEGER. Workspace array, DIMENSION (*liwork*). Used in real flavors only.

liwork (local or global) INTEGER. The dimension of the array *iwork*; used in real flavors only. Must be at least

$$liwork \geq LOCr(n+mod(ia-1,mb\_a)).$$

rwork (local) REAL for `pcgecon`
DOUBLE PRECISION for `pzgecon`
Workspace array, DIMENSION (*lrwork*). Used in complex flavors only.

lrwork (local or global) INTEGER. The dimension of the array *rwork*; used in complex flavors only. Must be at least

$$lrwork \geq 2*LOCc(n+mod(ja-1,nb\_a)).$$

Output Parameters

rcond (global) REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
The reciprocal of the condition number of the distributed matrix `sub(A)`. See Description.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

<code>iwork(1)</code>	On exit, <code>iwork(1)</code> contains the minimum value of <code>liwork</code> required for optimum performance (for real flavors).
<code>rwork(1)</code>	On exit, <code>rwork(1)</code> contains the minimum value of <code>lrwork</code> required for optimum performance (for complex flavors).
<code>info</code>	(global) INTEGER. If <code>info=0</code> , the execution is successful. <code>info < 0</code> : If the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <code>info = -(i*100+j)</code> ; if the <i>i</i> -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

p?pocon

Estimates the reciprocal of the condition number (in the 1 - norm) of a symmetric / Hermitian positive-definite distributed matrix.

Syntax

```
call pspcocon( uplo, n, a, ia, ja, desca, anorm, rcond, work, lwork, iwork,
               liwork, info)

call pdpcocon( uplo, n, a, ia, ja, desca, anorm, rcond, work, lwork, iwork,
               liwork, info)

call pcpcocn( uplo, n, a, ia, ja, desca, anorm, rcond, work, lwork, rwork,
               lrwork, info)

call pzpcocn( uplo, n, a, ia, ja, desca, anorm, rcond, work, lwork, rwork,
               lrwork, info)
```

Description

This routine estimates the reciprocal of the condition number (in the 1 - norm) of a real symmetric or complex Hermitian positive definite distributed matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$, using the Cholesky factorization $\text{sub}(A) = U^H * U$ or $\text{sub}(A) = L * L^H$ computed by `p?potrf`.

An estimate is obtained for $||(\text{sub}(A))^{-1}||$, and the reciprocal of the condition number is computed as

$$rcond = \frac{1}{\|sub(A)\| \times \|(sub(A))^{-1}\|}$$

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Must be 'U' or 'L'. Specifies whether the factor stored in sub(A) is upper or lower triangular. If <i>uplo</i> = 'U', sub(A) stores the upper triangular factor U of the Cholesky factorization $sub(A) = U^H * U$. If <i>uplo</i> = 'L', sub(A) stores the lower triangular factor L of the Cholesky factorization $sub(A) = L * L^H$.
<i>n</i>	(global) INTEGER. The order of the distributed submatrix sub(A) ($n \geq 0$).
<i>a</i>	(local) REAL for pspocon DOUBLE PRECISION for pdpocon COMPLEX for pcpocon DOUBLE COMPLEX for pzpocon. Pointer into the local memory to an array of dimension $a(lld\_a, LOCC(ja+n-1))$. The array <i>a</i> contains the local pieces of the factors L or U from the Cholesky factorization $sub(A) = U^H * U$, or $sub(A) = L * L^H$, as computed by p?potrf.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>A</i> indicating the first row and the first column of the submatrix sub(A), respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen</i>). The array descriptor for the distributed matrix <i>A</i>.
<i>anorm</i>	(global) REAL for single precision flavors, DOUBLE PRECISION for double precision flavors. The 1-norm of the symmetric/Hermitian distributed matrix sub(A).
<i>work</i>	(local) REAL for pspocon

	DOUBLE PRECISION for pdpocon COMPLEX for pcpocon DOUBLE COMPLEX for pzpocon. The array <i>work</i> of dimension (<i>lwork</i>) is a workspace array.
<i>lwork</i>	(local or global) INTEGER. The dimension of the array <i>work</i> . For real flavors: <i>lwork</i> must be at least $lwork \geq 2*LOCr(n+mod(ia-1,mb\_a)) + 2*LOCc(n+mod(ja-1,nb\_a)) + \max(2, \max(nb\_a*ceil(NPROW-1, NPCOL), LOCc(n+mod(ja-1,nb\_a)) + nb\_a*ceil(NPCOL-1, NPROW)))$ For complex flavors: <i>lwork</i> must be at least $lwork \geq 2*LOCr(n+mod(ia-1,mb\_a)) + \max(2, \max(nb\_a*\max(1,ceil(NPROW-1, NPCOL)), LOCc(n+mod(ja-1,nb\_a)) + nb\_a*\max(1,ceil(NPCOL-1, NPROW))))$
<i>iwork</i>	(local) INTEGER. Workspace array, DIMENSION (<i>liwork</i>). Used in real flavors only.
<i>liwork</i>	(local or global) INTEGER. The dimension of the array <i>iwork</i> ; used in real flavors only. Must be at least $liwork \geq LOCr(n+mod(ia-1,mb\_a))$.
<i>rwork</i>	(local) REAL for pcpocon DOUBLE PRECISION for pzpocon Workspace array, DIMENSION (<i>lrwork</i>). Used in complex flavors only.
<i>lrwork</i>	(local or global) INTEGER. The dimension of the array <i>rwork</i> ; used in complex flavors only. Must be at least $lrwork \geq 2*LOCc(n+mod(ja-1,nb\_a))$.

Output Parameters

<i>rcond</i>	(global) REAL for single precision flavors. DOUBLE PRECISION for double precision flavors.

	The reciprocal of the condition number of the distributed matrix $\text{sub}(A)$.
<code>work(1)</code>	On exit, <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>iwork(1)</code>	On exit, <code>iwork(1)</code> contains the minimum value of <code>liwork</code> required for optimum performance (for real flavors).
<code>rwork(1)</code>	On exit, <code>rwork(1)</code> contains the minimum value of <code>lrwork</code> required for optimum performance (for complex flavors).
<code>info</code>	(global) INTEGER. If <code>info=0</code> , the execution is successful. <code>info < 0</code> : If the <i>i</i> th argument is an array and the <i>j</i> th entry had an illegal value, then <code>info = -(i*100+j)</code> ; if the <i>i</i> th argument is a scalar and had an illegal value, then <code>info = -i</code> .

p?trcon

Estimates the reciprocal of the condition number of a triangular distributed matrix in either 1-norm or infinity-norm.

Syntax

```
call pstrcon( norm, uplo, diag, n, a, ia, ja, desca, rcond, work, lwork,
iwork, liwork, info)

call pdtrcon( norm, uplo, diag, n, a, ia, ja, desca, rcond, work, lwork,
iwork, liwork, info)

call pctrcon( norm, uplo, diag, n, a, ia, ja, desca, rcond, work, lwork,
rwork, lrwork, info)

call pztrcon( norm, uplo, diag, n, a, ia, ja, desca, rcond, work, lwork,
rwork, lrwork, info)
```

Description

This routine estimates the reciprocal of the condition number of a triangular distributed matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$, in either the 1-norm or the infinity-norm.

The norm of $\text{sub}(A)$ is computed and an estimate is obtained for $\|(\text{sub}(A))^{-1}\|$, then the reciprocal of the condition number is computed as

$$rcond = \frac{1}{\|sub(A)\| \times \|(sub(A))^{-1}\|}$$

Input Parameters

<i>norm</i>	(global) CHARACTER*1. Must be '1' or 'O' or 'I'. Specifies whether the 1-norm condition number or the infinity-norm condition number is required. If <i>norm</i> = '1' or 'O', then the 1-norm is used; If <i>norm</i> = 'I', then the infinity-norm is used.
<i>uplo</i>	(global) CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', sub(<i>A</i>) is upper triangular. If <i>uplo</i> = 'L', sub(<i>A</i>) is lower triangular.
<i>diag</i>	(global) CHARACTER*1. Must be 'N' or 'U'. If <i>diag</i> = 'N', sub(<i>A</i>) is non-unit triangular. If <i>diag</i> = 'U', sub(<i>A</i>) is unit triangular.
<i>n</i>	(global) INTEGER. The order of the distributed submatrix sub(<i>A</i>), (<i>n</i>≥0).
<i>a</i>	(local) REAL for pstrcon DOUBLE PRECISION for pdtrcon COMPLEX for petrcon DOUBLE COMPLEX for pztrcon. Pointer into the local memory to an array of dimension <i>a</i>(<i>lld_a</i>, <i>LOCc</i>(<i>ja</i>+<i>n</i>-1)). The array <i>a</i> contains the local pieces of the triangular distributed matrix sub(<i>A</i>). If <i>uplo</i> = 'U', the leading <i>n</i>-by-<i>n</i> upper triangular part of this distributed matrix contains the upper triangular matrix, and its strictly lower triangular part is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i>-by-<i>n</i> lower triangular part of this distributed matrix contains the lower triangular matrix, and its strictly upper triangular part is not referenced. If <i>diag</i> = 'U', the diagonal elements of sub(<i>A</i>) are also not referenced and are assumed to be 1.

<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>A</i> indicating the first row and the first column of the submatrix <i>sub(A)</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>work</i>	(local) REAL for pstrcon DOUBLE PRECISION for pdtrcon COMPLEX for pctrcon DOUBLE COMPLEX for pztrcon. The array <i>work</i> of dimension (<i>lwork</i>) is a workspace array.
<i>lwork</i>	(local or global) INTEGER. The dimension of the array <i>work</i> . For real flavors: <i>lwork</i> must be at least $lwork \geq 2*LOCr(n+\text{mod}(ia-1,mb\_a))+$ $LOCc(n+\text{mod}(ja-1,nb\_a))+\max(2,$ $\max(nb\_a*\max(1,\text{ceil}(NPROW-1,NPCOL)),$ $LOCc(n+\text{mod}(ja-1,nb\_a))+nb\_a*\max(1,\text{ceil}(NPCOL-1,$ $NPROW))).$ For complex flavors: <i>lwork</i> must be at least $lwork \geq 2*LOCr(n+\text{mod}(ia-1,mb\_a))+\max(2,$ $\max(nb\_a*\text{ceil}(NPROW-1,NPCOL),$ $LOCc(n+\text{mod}(ja-1,nb\_a))+nb\_a*\text{ceil}(NPCOL-1,$ $NPROW))).$
<i>iwork</i>	(local) INTEGER. Workspace array, DIMENSION (<i>liwork</i>). Used in real flavors only.
<i>liwork</i>	(local or global) INTEGER. The dimension of the array <i>iwork</i> ; used in real flavors only. Must be at least $liwork \geq LOCr(n+\text{mod}(ia-1,mb\_a)).$
<i>rwork</i>	(local) REAL for pcpccon DOUBLE PRECISION for pzpccon Workspace array, DIMENSION (<i>lrwork</i>). Used in complex flavors only.
<i>lrwork</i>	(local or global) INTEGER. The dimension of the array <i>rwork</i> ; used in complex flavors only. Must be at least

$$lrwork \geq LOCC(n + \text{mod}(ja-1, nb\_a)).$$

Output Parameters

<i>rcond</i>	(global) REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. The reciprocal of the condition number of the distributed matrix sub(A).
<i>work</i> (1)	On exit, <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>iwork</i> (1)	On exit, <i>iwork</i> (1) contains the minimum value of <i>liwork</i> required for optimum performance (for real flavors).
<i>rwork</i> (1)	On exit, <i>rwork</i> (1) contains the minimum value of <i>lrwork</i> required for optimum performance (for complex flavors).
<i>info</i>	(global) INTEGER. If <i>info</i> =0, the execution is successful. <i>info</i> < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <i>info</i> = -(<i>i</i> *100+ <i>j</i>); if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

Refining the Solution and Estimating Its Error

This section describes the ScaLAPACK routines for refining the computed solution of a system of linear equations and estimating the solution error. You can call these routines after factorizing the matrix of the system of equations and computing the solution (see [Routines for Matrix Factorization](#) and [Solving Systems of Linear Equations](#)).

p?gerfs

Improves the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution.

Syntax

```
call psgerfs(trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, ipiv,
b, ib, jb, descb, x, ix, jx, descx, ferr, berr, work, lwork, iwork, liwork,
info)
```

```
call pdgerfs(trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, ipiv,
b, ib, jb, descb, x, ix, jx, descx, ferr, berr, work, lwork, iwork, liwork,
info)
```

```
call pcgerfs(trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, ipiv,
b, ib, jb, descb, x, ix, jx, descx, ferr, berr, work, lwork, rwork, lrwork,
info)
```

```
call pzgerfs(trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, ipiv,
b, ib, jb, descb, x, ix, jx, descx, ferr, berr, work, lwork, rwork, lrwork,
info)
```

Description

This routine improves the computed solution to one of the systems of linear equations

$\text{sub}(A) * \text{sub}(X) = \text{sub}(B),$

$\text{sub}(A)^T * \text{sub}(X) = \text{sub}(B),$ or

$\text{sub}(A)^H * \text{sub}(X) = \text{sub}(B)$ and provides error bounds and backward error estimates for the solution.

Here $\text{sub}(A) = A(\text{ia}:\text{ia}+\text{n}-1, \text{ja}:\text{ja}+\text{n}-1)$, $\text{sub}(B) = B(\text{ib}:\text{ib}+\text{n}-1, \text{jb}:\text{jb}+\text{nrhs}-1)$, and $\text{sub}(X) = X(\text{ix}:\text{ix}+\text{n}-1, \text{jx}:\text{jx}+\text{nrhs}-1)$.

Input Parameters

trans (global) CHARACTER*1. Must be 'N' or 'T' or 'C'.
 Specifies the form of the system of equations:
 If *trans* = 'N', the system has the form $\text{sub}(A) * \text{sub}(X)$
 = $\text{sub}(B)$ (No transpose);

	If $trans = 'T'$, the system has the form $sub(A)^T * sub(X) = sub(B)$ (Transpose); If $trans = 'C'$, the system has the form $sub(A)^H * sub(X) = sub(B)$ (Conjugate transpose).
n	(global) INTEGER. The order of the distributed submatrix $sub(A)$ ($n \geq 0$).
$nrhs$	(global) INTEGER. The number of right-hand sides, i.e., the number of columns of the matrices $sub(B)$ and $sub(X)$ ($nrhs \geq 0$).
a, af, b, x	(local) REAL for psgerfs DOUBLE PRECISION for pdgerfs COMPLEX for pcgerfs DOUBLE COMPLEX for pzgerfs. Pointers into the local memory to arrays of local dimension $a(lld\_a, LOCC(ja+n-1))$, $af(lld\_af, LOCC(jaf+n-1))$, $b(lld\_b, LOCC(jb+nrhs-1))$, and $x(lld\_x, LOCC(jx+nrhs-1))$, respectively. The array a contains the local pieces of the distributed matrix $sub(A)$. The array af contains the local pieces of the distributed factors of the matrix $sub(A) = P * L * U$ as computed by p?getrf. The array b contains the local pieces of the distributed matrix of right hand sides $sub(B)$. On entry, the array x contains the local pieces of the distributed solution matrix $sub(X)$.
ia, ja	(global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of the submatrix $sub(A)$, respectively.
$desca$	(global and local) INTEGER array, dimension ($dlen\_$). The array descriptor for the distributed matrix A .
iaf, jaf	(global) INTEGER. The row and column indices in the global array AF indicating the first row and the first column of the submatrix $sub(AF)$, respectively.

<i>descaf</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>AF</i> .
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the global array <i>B</i> indicating the first row and the first column of the submatrix <i>sub(B)</i> , respectively.
<i>descb</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>B</i> .
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the global array <i>X</i> indicating the first row and the first column of the submatrix <i>sub(X)</i> , respectively.
<i>descx</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>X</i> .
<i>ipiv</i>	(local) INTEGER. Array, dimension $LOCr(m_{af} + m_{b_{af}})$. This array contains pivoting information as computed by p?getrf . If <i>ipiv</i> (<i>i</i>)= <i>j</i> , then the local row <i>i</i> was swapped with the global row <i>j</i> . This array is tied to the distributed matrix <i>A</i> .
<i>work</i>	(local) REAL for <i>psgerfs</i> DOUBLE PRECISION for <i>pdgerfs</i> COMPLEX for <i>pcgerfs</i> DOUBLE COMPLEX for <i>pzgerfs</i> . The array <i>work</i> of dimension (<i>lwork</i>) is a workspace array.
<i>lwork</i>	(local or global) INTEGER. The dimension of the array <i>work</i> . For real flavors: <i>lwork</i> must be at least $lwork \geq 3 * LOCr(n + \text{mod}(ia-1, mb_a))$ For complex flavors: <i>lwork</i> must be at least $lwork \geq 2 * LOCr(n + \text{mod}(ia-1, mb_a))$
<i>iwork</i>	(local) INTEGER. Workspace array, DIMENSION (<i>liwork</i>). Used in real flavors only.
<i>liwork</i>	(local or global) INTEGER. The dimension of the array <i>iwork</i> ; used in real flavors only. Must be at least

liwork $\geq LOCr(n+\text{mod}(ib-1,mb\_b))$.

rwork (local) REAL for `pcgerfs`
 DOUBLE PRECISION for `pzgerfs`
 Workspace array, DIMENSION (*lrwork*). Used in complex flavors only.

lrwork (local or global) INTEGER. The dimension of the array *rwork*; used in complex flavors only. Must be at least $liwork \geq LOCr(n+\text{mod}(ib-1,mb\_b))$.

Output Parameters

x On exit, contains the improved solution vectors.

ferr, *berr* REAL for single precision flavors.
 DOUBLE PRECISION for double precision flavors.
 Arrays, dimension $LOCc(jb+nrhs-1)$ each.
 The array *ferr* contains the estimated forward error bound for each solution vector of `sub(x)`.
 If `XTRUE` is the true solution corresponding to `sub(x)`, *ferr* is an estimated upper bound for the magnitude of the largest element in $(\text{sub}(X) - XTRUE)$ divided by the magnitude of the largest element in `sub(x)`. The estimate is as reliable as the estimate for `rcond`, and is almost always a slight overestimate of the true error.
 This array is tied to the distributed matrix *x*.
 The array *berr* contains the component-wise relative backward error of each solution vector (that is, the smallest relative change in any entry of `sub(A)` or `sub(B)` that makes `sub(x)` an exact solution). This array is tied to the distributed matrix *x*.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

iwork(1) On exit, *iwork*(1) contains the minimum value of *liwork* required for optimum performance (for real flavors).

rwork(1) On exit, *rwork*(1) contains the minimum value of *lrwork* required for optimum performance (for complex flavors).

info (global) INTEGER. If *info*=0, the execution is successful.

info < 0:

If the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = $-(i*100+j)$; if the *i*-th argument is a scalar and had an illegal value, then *info* = $-i$.

p?porfs

Improves the computed solution to a system of linear equations with symmetric/Hermitian positive definite distributed matrix and provides error bounds and backward error estimates for the solution.

Syntax

```
call psporfs(uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, b, ib,
             jb, descb, x, ix, jx, descx, ferr, berr, work, lwork, iwork, liwork, info)
call pdporfs(uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, b, ib,
             jb, descb, x, ix, jx, descx, ferr, berr, work, lwork, iwork, liwork, info)
call pcporfs(uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, b, ib,
             jb, descb, x, ix, jx, descx, ferr, berr, work, lwork, rwork, lrwork, info)
call pzporfs(uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, b, ib,
             jb, descb, x, ix, jx, descx, ferr, berr, work, lwork, rwork, lrwork, info)
```

Description

The routine p?porfs improves the computed solution to the system of linear equations

$$\text{sub}(A) * \text{sub}(X) = \text{sub}(B),$$

where $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$ is a real symmetric or complex Hermitian positive definite distributed matrix and

$$\text{sub}(B) = B(\text{ib}:\text{ib}+n-1, \text{jb}:\text{jb}+\text{nrhs}-1),$$

$$\text{sub}(X) = X(\text{ix}:\text{ix}+n-1, \text{jx}:\text{jx}+\text{nrhs}-1)$$

are right-hand side and solution submatrices, respectively. This routine also provides error bounds and backward error estimates for the solution.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Must be 'U' or 'L'. Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix $\text{sub}(A)$ is stored. If <i>uplo</i> = 'U', $\text{sub}(A)$ is upper triangular. If <i>uplo</i> = 'L', $\text{sub}(A)$ is lower triangular.
<i>n</i>	(global) INTEGER. The order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
<i>nrhs</i>	(global) INTEGER. The number of right-hand sides, i.e., the number of columns of the matrices $\text{sub}(B)$ and $\text{sub}(X)$ ($nrhs \geq 0$).
<i>a, af, b, x</i>	(local) REAL for psporfs DOUBLE PRECISION for pdporfs COMPLEX for pcporfs DOUBLE COMPLEX for pzporfs. Pointers into the local memory to arrays of local dimension $a(lld\_a, LOCC(ja+n-1))$, $af(lld\_af, LOCC(ja+n-1))$, $b(lld\_b, LOCC(jb+nrhs-1))$, and $x(lld\_x, LOCC(jx+nrhs-1))$, respectively. The array <i>a</i> contains the local pieces of the n-by-n symmetric/Hermitian distributed matrix $\text{sub}(A)$. If <i>uplo</i> = 'U', the leading n-by-n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If <i>uplo</i> = 'L', the leading n-by-n lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the distributed matrix, and its strictly upper triangular part is not referenced. The array <i>af</i> contains the factors L or U from the Cholesky factorization $\text{sub}(A) = L * L^H$ or $\text{sub}(A) = U^H * U$, as computed by p?potrf. On entry, the array <i>b</i> contains the local pieces of the distributed matrix of right hand sides $\text{sub}(B)$. On entry, the array <i>x</i> contains the local pieces of the solution vectors $\text{sub}(X)$.

<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>A</i> indicating the first row and the first column of the submatrix <i>sub(A)</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>iaf, jaf</i>	(global) INTEGER. The row and column indices in the global array <i>AF</i> indicating the first row and the first column of the submatrix <i>sub(AF)</i> , respectively.
<i>descaf</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>AF</i> .
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the global array <i>B</i> indicating the first row and the first column of the submatrix <i>sub(B)</i> , respectively.
<i>descb</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>B</i> .
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the global array <i>X</i> indicating the first row and the first column of the submatrix <i>sub(X)</i> , respectively.
<i>descx</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>X</i> .
<i>work</i>	(local) REAL for psporfs DOUBLE PRECISION for pdporfs COMPLEX for pcporfs DOUBLE COMPLEX for pzporfs. The array <i>work</i> of dimension (<i>lwork</i>) is a workspace array.
<i>lwork</i>	(local) INTEGER. The dimension of the array <i>work</i> . For real flavors: <i>lwork</i> must be at least $lwork \geq 3*LOCr(n+\text{mod}(ia-1,mb\_a))$ For complex flavors: <i>lwork</i> must be at least $lwork \geq 2*LOCr(n+\text{mod}(ia-1,mb\_a))$
<i>iwork</i>	(local) INTEGER. Workspace array, DIMENSION (<i>liwork</i>). Used in real flavors only.

<i>liwork</i>	(local or global) INTEGER. The dimension of the array <i>iwork</i> ; used in real flavors only. Must be at least $liwork \geq LOCr(n+mod(ib-1,mb\_b))$.
<i>rwork</i>	(local) REAL for pcporfs DOUBLE PRECISION for pzpofrs Workspace array, DIMENSION (<i>lrwork</i>). Used in complex flavors only.
<i>lrwork</i>	(local or global) INTEGER. The dimension of the array <i>rwork</i> ; used in complex flavors only. Must be at least $lrwork \geq LOCr(n+mod(ib-1,mb\_b))$.

Output Parameters

<i>x</i>	On exit, contains the improved solution vectors.
<i>ferr</i> , <i>berr</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. Arrays, dimension $LOCc(jb+nrhs-1)$ each. The array <i>ferr</i> contains the estimated forward error bound for each solution vector of $sub(X)$. If <i>XTRUE</i> is the true solution corresponding to $sub(X)$, <i>ferr</i> is an estimated upper bound for the magnitude of the largest element in $(sub(X) - XTRUE)$ divided by the magnitude of the largest element in $sub(X)$. The estimate is as reliable as the estimate for <i>rcond</i> , and is almost always a slight overestimate of the true error. This array is tied to the distributed matrix <i>x</i> . The array <i>berr</i> contains the component-wise relative backward error of each solution vector (that is, the smallest relative change in any entry of $sub(A)$ or $sub(B)$ that makes $sub(x)$ an exact solution). This array is tied to the distributed matrix <i>x</i> .
<i>work</i> (1)	On exit, <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>iwork</i> (1)	On exit, <i>iwork</i> (1) contains the minimum value of <i>liwork</i> required for optimum performance (for real flavors).
<i>rwork</i> (1)	On exit, <i>rwork</i> (1) contains the minimum value of <i>lrwork</i> required for optimum performance (for complex flavors).

info (global) INTEGER. If *info*=0, the execution is successful.
info < 0:
 If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = $-(i*100+j)$; if the *i*-th argument is a scalar and had an illegal value, then *info* = $-i$.

p?trrfs

Provides error bounds and backward error estimates for the solution to a system of linear equations with a distributed triangular coefficient matrix.

Syntax

```
call pstrrfs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb,
x, ix, jx, descx, ferr, berr, work, lwork, iwork, liwork, info)

call pdtrrfs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb,
x, ix, jx, descx, ferr, berr, work, lwork, iwork, liwork, info)

call pctrfs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb,
x, ix, jx, descx, ferr, berr, work, lwork, rwork, lrwork, info)

call pztrrfs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb,
x, ix, jx, descx, ferr, berr, work, lwork, rwork, lrwork, info)
```

Description

The routine p?trrfs provides error bounds and backward error estimates for the solution to one of the systems of linear equations

$\text{sub}(A) * \text{sub}(X) = \text{sub}(B),$

$\text{sub}(A)^T * \text{sub}(X) = \text{sub}(B),$ or

$\text{sub}(A)^H * \text{sub}(X) = \text{sub}(B),$

where $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ is a triangular matrix,

$\text{sub}(B) = B(ib:ib+n-1, jb:jb+nrhs-1),$ and

$\text{sub}(X) = X(ix:ix+n-1, jx:jx+nrhs-1).$

The solution matrix x must be computed by `p?trtrs` or some other means before entering this routine. The routine `p?trrfs` does not do iterative refinement because doing so cannot improve the backward error.

Input Parameters

<code>uplo</code>	(global) CHARACTER*1. Must be 'U' or 'L'. If <code>uplo</code> = 'U', <code>sub(A)</code> is upper triangular. If <code>uplo</code> = 'L', <code>sub(A)</code> is lower triangular.
<code>trans</code>	(global) CHARACTER*1. Must be 'N' or 'T' or 'C'. Specifies the form of the system of equations: If <code>trans</code> = 'N', the system has the form <code>sub(A) * sub(X)</code> = <code>sub(B)</code> (No transpose); If <code>trans</code> = 'T', the system has the form <code>sub((A)^T * sub(X)</code> = <code>sub(B)</code> (Transpose); If <code>trans</code> = 'C', the system has the form <code>sub(A)^H * sub(X)</code> = <code>sub(B)</code> (Conjugate transpose).
<code>diag</code>	CHARACTER*1. Must be 'N' or 'U'. If <code>diag</code> = 'N', then <code>sub(A)</code> is non-unit triangular. If <code>diag</code> = 'U', then <code>sub(A)</code> is unit triangular.
<code>n</code>	(global) INTEGER. The order of the distributed matrix <code>sub(A)</code> ($n \geq 0$).
<code>nrhs</code>	(global) INTEGER. The number of right-hand sides, that is, the number of columns of the matrices <code>sub(B)</code> and <code>sub(X)</code> ($nrhs \geq 0$).
<code>a, b, x</code>	(local) REAL for <code>pstrrfs</code> DOUBLE PRECISION for <code>pdtrrfs</code> COMPLEX for <code>pctrfs</code> DOUBLE COMPLEX for <code>pztrrfs</code> . Pointers into the local memory to arrays of local dimension <code>a(lld_a, LOCC(ja+n-1))</code> , <code>b(lld_b, LOCC(jb+nrhs-1))</code> , and <code>x(lld_x, LOCC(jx+nrhs-1))</code> , respectively. The array <code>a</code> contains the local pieces of the original triangular distributed matrix <code>sub(A)</code> .

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *sub(A)* contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *sub(A)* contains the lower triangular part of the distributed matrix, and its strictly upper triangular part is not referenced.

If *diag* = 'U', the diagonal elements of *sub(A)* are also not referenced and are assumed to be 1.

On entry, the array *b* contains the local pieces of the distributed matrix of right hand sides *sub(B)*.

On entry, the array *x* contains the local pieces of the solution vectors *sub(X)*.

ia, ja (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix *sub(A)*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

ib, jb (global) INTEGER. The row and column indices in the global array *B* indicating the first row and the first column of the submatrix *sub(B)*, respectively.

descb (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

ix, jx (global) INTEGER. The row and column indices in the global array *X* indicating the first row and the first column of the submatrix *sub(X)*, respectively.

descx (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *X*.

work (local)
 REAL for *pstrrfs*
 DOUBLE PRECISION for *pdtrrfs*
 COMPLEX for *pctrfs*
 DOUBLE COMPLEX for *pztrfs*.
 The array *work* of dimension (*lwork*) is a workspace array.

lwork (local) INTEGER. The dimension of the array *work*.

For real flavors:

	$lwork$ must be at least $lwork \geq 3*LOCr(n+\text{mod}(ia-1,mb\_a))$ For complex flavors: $lwork$ must be at least $lwork \geq 2*LOCr(n+\text{mod}(ia-1,mb\_a))$
$iwork$	(local) INTEGER. Workspace array, DIMENSION ($liwork$). Used in real flavors only.
$liwork$	(local or global) INTEGER. The dimension of the array $iwork$; used in real flavors only. Must be at least $liwork \geq LOCr(n+\text{mod}(ib-1,mb\_b))$.
$rwork$	(local) REAL for pctrdfs DOUBLE PRECISION for pztrdfs Workspace array, DIMENSION ($lrwork$). Used in complex flavors only.
$lrwork$	(local or global) INTEGER. The dimension of the array $rwork$; used in complex flavors only. Must be at least $lrwork \geq LOCr(n+\text{mod}(ib-1,mb\_b))$.

Output Parameters

$ferr, berr$	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. Arrays, dimension $LOCc(jb+nrhs-1)$ each. The array $ferr$ contains the estimated forward error bound for each solution vector of $\text{sub}(x)$. If $XTRUE$ is the true solution corresponding to $\text{sub}(x)$, $ferr$ is an estimated upper bound for the magnitude of the largest element in $(\text{sub}(x) - XTRUE)$ divided by the magnitude of the largest element in $\text{sub}(x)$. The estimate is as reliable as the estimate for $rcond$, and is almost always a slight overestimate of the true error. This array is tied to the distributed matrix x .

	The array <i>berr</i> contains the component-wise relative backward error of each solution vector (that is, the smallest relative change in any entry of $\text{sub}(A)$ or $\text{sub}(B)$ that makes $\text{sub}(X)$ an exact solution). This array is tied to the distributed matrix <i>X</i> .
<i>work</i> (1)	On exit, <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>iwork</i> (1)	On exit, <i>iwork</i> (1) contains the minimum value of <i>liwork</i> required for optimum performance (for real flavors).
<i>rwork</i> (1)	On exit, <i>rwork</i> (1) contains the minimum value of <i>lrwork</i> required for optimum performance (for complex flavors).
<i>info</i>	(global) INTEGER. If <i>info</i> =0, the execution is successful. <i>info</i> < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <i>info</i> = $-(i*100+j)$; if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = $-i$.

Routines for Matrix Inversion

This sections describes ScaLAPACK routines that compute the inverse of a matrix based on the previously obtained factorization. Note that it is not recommended to solve a system of equations $Ax = b$ by first computing A^{-1} and then forming the matrix-vector product $x = A^{-1}b$. Call a solver routine instead (see [Solving Systems of Linear Equations](#)); this is more efficient and more accurate.

p?getri

Computes the inverse of a LU-factored distributed matrix.

Syntax

```
call psgetri(n, a, ia, ja, desca, ipiv, work, lwork, iwork, liwork, info)
call pdgetri(n, a, ia, ja, desca, ipiv, work, lwork, iwork, liwork, info)
call pcgetri(n, a, ia, ja, desca, ipiv, work, lwork, iwork, liwork, info)
call pzgetri(n, a, ia, ja, desca, ipiv, work, lwork, iwork, liwork, info)
```

Description

This routine computes the inverse of a general distributed matrix $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$ using the LU factorization computed by `p?getrf`. This method inverts U and then computes the inverse of $\text{sub}(A)$ by solving the system

$$\text{inv}(\text{sub}(A)) * L = \text{inv}(U)$$

for $\text{inv}(\text{sub}(A))$.

Input Parameters

<i>n</i>	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).
<i>a</i>	(local) REAL for <code>psgetri</code> DOUBLE PRECISION for <code>pdgetri</code> COMPLEX for <code>pcgetri</code> DOUBLE COMPLEX for <code>pzgetri</code> . Pointer into the local memory to an array of local dimension $a(\text{lld\_a}, \text{LOCc}(\text{ja}+n-1))$. On entry, the array <i>a</i> contains the local pieces of the L and U obtained by the factorization $\text{sub}(A) = P * L * U$ computed by <code>p?getrf</code> .
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>A</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>work</i>	(local) REAL for <code>psgetri</code> DOUBLE PRECISION for <code>pdgetri</code> COMPLEX for <code>pcgetri</code> DOUBLE COMPLEX for <code>pzgetri</code> . The array <i>work</i> of dimension (<i>lwork</i>) is a workspace array.
<i>lwork</i>	(local) INTEGER. The dimension of the array <i>work</i> . <i>lwork</i> must be at least $\text{lwork} \geq \text{LOCr}(n + \text{mod}(\text{ia}-1, \text{mb\_a})) * \text{nb\_a}$.

The array *work* is used to keep at most an entire column block of sub(*A*).

iwork (local) INTEGER. Workspace array used for physically transposing the pivots, DIMENSION (*liwork*).

liwork (local or global) INTEGER. The dimension of the array *iwork*. The minimal value *liwork* of is determined by the following code:

```
if NPROW == NPCOL then
    liwork = LOCC(n_a + mod(ja-1,nb_a)) + nb_a
else
    liwork = LOCC(n_a + mod(ja-1,nb_a)) +
    max(ceil(ceil(LOCr(m_a)/mb_a)/(lcm/NPROW)),nb_a)
end if
```

where *lcm* is the least common multiple of process rows and columns (NPROW and NPCOL).

Output Parameters

ipiv (local) INTEGER.
Array, dimension (*LOCr(m_a) + mb_a*).
This array contains the pivoting information.
If *ipiv*(*i*)=*j*, then the local row *i* was swapped with the global row *j*.
This array is tied to the distributed matrix *A*.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

iwork(1) On exit, *iwork*(1) contains the minimum value of *liwork* required for optimum performance.

info (global) INTEGER. If *info*=0, the execution is successful.
info < 0:

If the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

$info > 0$:

If $info = i$, $U(i,i)$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, and division by zero will occur if it is used to solve a system of equations.

p?potri

Computes the inverse of a symmetric/Hermitian positive definite distributed matrix.

Syntax

```
call pspotri(uplo, n, a, ia, ja, desca, info)
call pdpotri(uplo, n, a, ia, ja, desca, info)
call pcpotri(uplo, n, a, ia, ja, desca, info)
call pzpotri(uplo, n, a, ia, ja, desca, info)
```

Description

This routine computes the inverse of a real symmetric or complex Hermitian positive definite distributed matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ using the Cholesky factorization $\text{sub}(A) = U^H * U$ or $\text{sub}(A) = L * L^H$ computed by p?potrf.

Input Parameters

$uplo$	(global) CHARACTER*1. Must be 'U' or 'L'. Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix $\text{sub}(A)$ is stored. If $uplo = 'U'$, upper triangle of $\text{sub}(A)$ is stored. If $uplo = 'L'$, lower triangle of $\text{sub}(A)$ is stored.
n	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).

<i>a</i>	(local) REAL for pspotri DOUBLE PRECISION for pdpotri COMPLEX for pcpotri DOUBLE COMPLEX for pzpotri. Pointer into the local memory to an array of local dimension $a(lld\_a, LOCC(ja+n-1))$. On entry, the array <i>a</i> contains the local pieces of the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $\text{sub}(A) = U^H * U$, or $\text{sub}(A) = L * L^H$, as computed by p?potrf.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>A</i> indicating the first row and the first column of the submatrix sub(<i>A</i>), respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen</i>). The array descriptor for the distributed matrix <i>A</i>.

Output Parameters

<i>a</i>	On exit, overwritten by the local pieces of the upper or lower triangle of the (symmetric/Hermitian) inverse of sub(<i>A</i>).
<i>info</i>	(global) INTEGER. If <i>info</i>=0, the execution is successful. <i>info</i> < 0: If the <i>i</i>-th argument is an array and the <i>j</i>-th entry had an illegal value, then <i>info</i> = $-(i*100+j)$; if the <i>i</i>-th argument is a scalar and had an illegal value, then <i>info</i> = $-i$. <i>info</i> > 0: If <i>info</i> = <i>i</i>, the (<i>i</i>, <i>i</i>) element of the factor <i>U</i> or <i>L</i> is zero, and the inverse could not be computed.

p?trtri

Computes the inverse of a triangular distributed matrix.

Syntax

```
call pstrtri(uplo, diag, n, a, ia, ja, desca, info)
call pdtrtri(uplo, diag, n, a, ia, ja, desca, info)
call pctrtri(uplo, diag, n, a, ia, ja, desca, info)
call pztrtri(uplo, diag, n, a, ia, ja, desca, info)
```

Description

This routine computes the inverse of a real or complex upper or lower triangular distributed matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Must be 'U' or 'L'. Specifies whether the distributed matrix $\text{sub}(A)$ is upper or lower triangular. If <i>uplo</i> = 'U', $\text{sub}(A)$ is upper triangular. If <i>uplo</i> = 'L', $\text{sub}(A)$ is lower triangular.
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. Specifies whether or not the distributed matrix $\text{sub}(A)$ is unit triangular. If <i>diag</i> = 'N', then $\text{sub}(A)$ is non-unit triangular. If <i>diag</i> = 'U', then $\text{sub}(A)$ is unit triangular.
<i>n</i>	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).
<i>a</i>	(local) REAL for pstrtri DOUBLE PRECISION for pdtrtri COMPLEX for pctrtri DOUBLE COMPLEX for pztrtri.

Pointer into the local memory to an array of local dimension
 $a(lld\_a, LOCC(ja+n-1))$.

The array a contains the local pieces of the triangular distributed matrix $\text{sub}(A)$.

If $uplo = 'U'$, the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular matrix to be inverted, and the strictly lower triangular part of $\text{sub}(A)$ is not referenced.

If $uplo = 'L'$, the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular matrix, and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.

ia, ja

(global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.

$desca$

(global and local) INTEGER array, dimension $(dlen\_)$. The array descriptor for the distributed matrix A .

Output Parameters

a

On exit, overwritten by the (triangular) inverse of the original matrix.

$info$

(global) INTEGER. If $info=0$, the execution is successful.

$info < 0$:

If the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

$info > 0$:

If $info = k$, $A(ia+k-1, ja+k-1)$ is exactly zero. The triangular matrix $\text{sub}(A)$ is singular and its inverse can not be computed.

Routines for Matrix Equilibration

ScaLAPACK routines described in this section are used to compute scaling factors needed to equilibrate a matrix. Note that these routines do not actually scale the matrices.

p?geequ

Computes row and column scaling factors intended to equilibrate a general rectangular distributed matrix and reduce its condition number.

Syntax

```
call psgeequ(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, info)
call pdgeequ(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, info)
call pcgeequ(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, info)
call pzgeequ(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, info)
```

Description

This routine computes row and column scalings intended to equilibrate an m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ and reduce its condition number. The output array r returns the row scale factors and the array c the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix B with elements $b_{ij}=r(i)*a_{ij}*c(j)$ have absolute value 1.

$r(i)$ and $c(j)$ are restricted to be between $SMLNUM$ = smallest safe number and $BIGNUM$ = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of $\text{sub}(A)$ but works well in practice.

The auxiliary function p?laqge uses scaling factors computed by p?geequ to scale a general rectangular matrix.

Input Parameters

m	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$ ($m \geq 0$).
n	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).
a	(local) REAL for psgeequ DOUBLE PRECISION for pdgeequ

COMPLEX for pcgeequ
 DOUBLE COMPLEX for pzgeequ .
 Pointer into the local memory to an array of local dimension
 $a(lld\_a, LOCc(ja+n-1))$.
 The array a contains the local pieces of the m -by- n
 distributed matrix whose equilibration factors are to be
 computed.

ia, ja (global) INTEGER. The row and column indices in the global
 array A indicating the first row and the first column of the
 submatrix $\text{sub}(A)$, respectively.

$desca$ (global and local) INTEGER array, dimension $(dlen\_)$. The
 array descriptor for the distributed matrix A .

Output Parameters

r, c (local) REAL for single precision flavors;
 DOUBLE PRECISION for double precision flavors.
 Arrays, dimension $LOCr(m\_a)$ and $LOCc(n\_a)$, respectively.
 If $info = 0$, or $info > ia+m-1$, the array r ($ia:ia+m-1$)
 contains the row scale factors for $\text{sub}(A)$. r is aligned with
 the distributed matrix A , and replicated across every process
 column. r is tied to the distributed matrix A .
 If $info = 0$, the array c ($ja:ja+n-1$) contains the column
 scale factors for $\text{sub}(A)$. c is aligned with the distributed
 matrix A , and replicated down every process row. c is tied
 to the distributed matrix A .

$rowcnd, colcnd$ (global) REAL for single precision flavors;
 DOUBLE PRECISION for double precision flavors.
 If $info = 0$ or $info > ia+m-1$, $rowcnd$ contains the ratio
 of the smallest $r(i)$ to the largest $r(i)$ ($ia \leq i \leq$
 $ia+m-1$). If $rowcnd \geq 0.1$ and $amax$ is neither too large
 nor too small, it is not worth scaling by r ($ia:ia+m-1$).
 If $info = 0$, $colcnd$ contains the ratio of the smallest $c(j)$
 to the largest $c(j)$ ($ja \leq j \leq ja+n-1$).

If $colcnd \geq 0.1$, it is not worth scaling by $c(ja:ja+n-1)$.

$amax$ (global) REAL for single precision flavors;

DOUBLE PRECISION for double precision flavors.
 Absolute value of the largest matrix element. If *amax* is very close to overflow or very close to underflow, the matrix should be scaled.

info (global) INTEGER. If *info*=0, the execution is successful.
info < 0:
 If the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = $-(i*100+j)$; if the *i*-th argument is a scalar and had an illegal value, then *info* = $-i$.
info > 0:
 If *info* = *i* and
 $i \leq m$, the *i*th row of the distributed matrix *sub(A)* is exactly zero;
 $i > m$, the (*i-m*)th column of the distributed matrix *sub(A)* is exactly zero.

p?poequ

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite distributed matrix and reduce its condition number.

Syntax

```
call pspoequ(n, a, ia, ja, desca, sr, sc, scond, amax, info)
call pdpoequ(n, a, ia, ja, desca, sr, sc, scond, amax, info)
call pcpcpoequ(n, a, ia, ja, desca, sr, sc, scond, amax, info)
call pzpcpoequ(n, a, ia, ja, desca, sr, sc, scond, amax, info)
```

Description

This routine computes row and column scalings intended to equilibrate a real symmetric or complex Hermitian positive definite distributed matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ and reduce its condition number (with respect to the two-norm). The output arrays *sr* and *sc* return the row and column scale factors

$$s(i) = \frac{1}{\sqrt{a_{i,i}}}$$

These factors are chosen so that the scaled distributed matrix B with elements $b_{ij}=s(i)*a_{ij}*s(j)$ has ones on the diagonal.

This choice of sr and sc puts the condition number of B within a factor n of the smallest possible condition number over all possible diagonal scalings.

The auxiliary function `p?laqsy` uses scaling factors computed by `p?geequ` to scale a general rectangular matrix.

Input Parameters

n	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).
a	(local) REAL for <code>pspoequ</code> DOUBLE PRECISION for <code>pdpoequ</code> COMPLEX for <code>pcpoequ</code> DOUBLE COMPLEX for <code>pzpoequ</code> . Pointer into the local memory to an array of local dimension $a(\text{lld\_a}, \text{LOCc}(ja+n-1))$. The array a contains the n -by- n symmetric/Hermitian positive definite distributed matrix $\text{sub}(A)$ whose scaling factors are to be computed. Only the diagonal elements of $\text{sub}(A)$ are referenced.
ia, ja	(global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
$desca$	(global and local) INTEGER array, dimension (dlen_) . The array descriptor for the distributed matrix A .

Output Parameters

sr, sc	(local) REAL for single precision flavors;

DOUBLE PRECISION for double precision flavors.
 Arrays, dimension $LOCr(m_a)$ and $LOCc(n_a)$, respectively.
 If $info = 0$, the array $sr(ia:ia+n-1)$ contains the row scale factors for $sub(A)$. sr is aligned with the distributed matrix A , and replicated across every process column. sr is tied to the distributed matrix A .
 If $info = 0$, the array $sc(ja:ja+n-1)$ contains the column scale factors for $sub(A)$. sc is aligned with the distributed matrix A , and replicated down every process row. sc is tied to the distributed matrix A .

scond (global)
 REAL for single precision flavors;
 DOUBLE PRECISION for double precision flavors.
 If $info = 0$, *scond* contains the ratio of the smallest $sr(i)$ (or $sc(j)$) to the largest $sr(i)$ (or $sc(j)$), with
 $ia \leq i \leq ia+n-1$ and $ja \leq j \leq ja+n-1$.
 If $scond \geq 0.1$ and *amax* is neither too large nor too small, it is not worth scaling by sr (or sc).

amax (global)
 REAL for single precision flavors;
 DOUBLE PRECISION for double precision flavors.
 Absolute value of the largest matrix element. If *amax* is very close to overflow or very close to underflow, the matrix should be scaled.

info (global) INTEGER.
 If $info=0$, the execution is successful.
 $info < 0$:
 If the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.
 $info > 0$:
 If $info = k$, the k -th diagonal entry of $sub(A)$ is nonpositive.

Orthogonal Factorizations

This section describes the ScaLAPACK routines for the QR (RQ) and LQ (QL) factorization of matrices. Routines for the RZ factorization as well as for generalized QR and RQ factorizations are also included. For the mathematical definition of the factorizations, see the respective LAPACK sections or refer to [SLUG].

Table 6-3 lists ScaLAPACK routines that perform orthogonal factorization of matrices.

Table 6-3 Computational Routines for Orthogonal Factorizations

Matrix type, factorization	Factorize without pivoting	Factorize with pivoting	Generate matrix Q	Apply matrix Q
general matrices, QR factorization	p?geqrf	p?geqpf	p?orgqr p?ungqr	p?omqrp?unmqr
general matrices, RQ factorization	p?gerqf		p?orgqp?ungrq	p?omrqp?unmrq
general matrices, LQ factorization	p?gelqf		p?orglp?unglq	p?omlp?unmlq
general matrices, QL factorization	p?geqlf		p?orgqlp?ungql	p?omqlp?unmql
trapezoidal matrices, RZ factorization	p?tzzrf			p?omrzp?unmrz
pair of matrices, generalized QR factorization	p?ggqrf			
pair of matrices, generalized RQ factorization	p?ggrqf			

[p?geqrf](#)

Computes the QR factorization of a general m -by- n matrix.

Syntax

```
call psgeqrf( m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgeqrf( m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgeqrf( m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgeqrf( m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine forms the QR factorization of a general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ as

$$A = Q^* R$$

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows in the distributed submatrix $\text{sub}(A)$; ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed submatrix $\text{sub}(A)$; ($n \geq 0$).
<i>a</i>	(local) REAL for psgeqrf DOUBLE PRECISION for pdgeqrf COMPLEX for pcgeqrf DOUBLE COMPLEX for pzgeqrf. Pointer into the local memory to an array of local dimension ($lld\_a, LOCC(ja+n-1)$). Contains the local pieces of the distributed matrix $\text{sub}(A)$ to be factored.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i>
<i>work</i>	(local). REAL for psgeqrf DOUBLE PRECISION for pdgeqrf. COMPLEX for pcgeqrf. DOUBLE COMPLEX for pzgeqrf Workspace array of dimension <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, dimension of <i>work</i> , must be at least $lwork \geq nb\_a * (mp0+nq0+nb\_a)$, where $iroff = \text{mod}(ia-1, mb\_a)$, $icoff = \text{mod}(ja-1, nb\_a)$, $iarow = \text{indxg2p}(ia, mb\_a, MYROW, rsrc\_a, NPROW)$,

`iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),`
`mp0 = numroc(m+iroff, mb_a, MYROW, iarow, NPROW),`
`nq0 = numroc(n+icoff, nb_a, MYCOL, iacol, NPCOL),`
 and `numroc, indxg2p` are ScaLAPACK tool functions; `MYROW,`
`MYCOL, NPROW` and `NPCOL` can be determined by calling the
 subroutine `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace
 query is assumed; the routine only calculates the minimum
 and optimal size for all work arrays. Each of these values
 is returned in the first entry of the corresponding work array,
 and no error message is issued by `pxerbla`.

Output Parameters

<code>a</code>	The elements on and above the diagonal of <code>sub(A)</code> contain the $\min(m,n)$ -by- n upper trapezoidal matrix R (R is upper triangular if $m \geq n$); the elements below the diagonal, with the array <code>tau</code> , represent the orthogonal/unitary matrix Q as a product of elementary reflectors (see <i>Application Notes</i> below).
<code>tau</code>	(local) REAL for <code>psgeqrf</code> DOUBLE PRECISION for <code>pdgeqrf</code> COMPLEX for <code>pcgeqrf</code> DOUBLE COMPLEX for <code>pzgeqrf</code> . Array, DIMENSION <code>LOCc(ja+min(m,n)-1)</code> . Contains the scalar factor <code>tau</code> of elementary reflectors. <code>tau</code> is tied to the distributed matrix <code>A</code> .
<code>work(1)</code>	On exit, <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) INTEGER. = 0, the execution is successful. < 0, if the i -th argument is an array and the j -entry had an illegal value, then <code>info = - (i* 100+j)</code> , if the i -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(ja) * H(ja+1) * \dots * H(ja+k-1),$$

where $k = \min(m, n)$.

Each $H(i)$ has the form

$$H(j) = I - \tau * v * v'$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:m)$ is stored on exit in $A(ia+i:ia+m-1, ja+i-1)$, and τ in $\tau(ja+i-1)$.

p?geqpf

Computes the QR factorization of a general m-by-n matrix with pivoting.

Syntax

```
call psgeqpf( m, n, a, ia, ja, desca, ipiv, tau, work, lwork, info)
call pdgeqpf( m, n, a, ia, ja, desca, ipiv, tau, work, lwork, info)
call pcgeqpf( m, n, a, ia, ja, desca, ipiv, tau, work, lwork, info)
call pzgeqpf( m, n, a, ia, ja, desca, ipiv, tau, work, lwork, info)
```

Description

The routine forms the QR factorization with column pivoting of a general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ as

$$\text{sub}(A) * P = Q * R$$

Input Parameters

m	(global) INTEGER. The number of rows in the submatrix $\text{sub}(A)$ ($m \geq 0$).
n	(global) INTEGER. The number of columns in the submatrix $\text{sub}(A)$ ($n \geq 0$).
a	(local) REAL for psgeqpf

DOUBLE PRECISION for pdgeqpf
 COMPLEX for pcgeqpf
 DOUBLE COMPLEX for pzgeqpf.
 Pointer into the local memory to an array of local dimension
 (*lld_a*, *LOCc(ja+n-1)*).
 Contains the local pieces of the distributed matrix sub(*A*)
 to be factored.

ia, ja (global) INTEGER. The row and column indices in the global
 array *a* indicating the first row and the first column of the
 submatrix *A(ia:ia+m-1, ja:ja+n-1)*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The
 array descriptor for the distributed matrix *A*.

work (local).
 REAL for psgeqpf
 DOUBLE PRECISION for pdgeqpf.
 COMPLEX for pcgeqpf.
 DOUBLE COMPLEX for pzgeqpf
 Workspace array of dimension *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at
 least
For real flavors:
 $lwork \geq \max(3, mp0+nq0) + LOCc(ja+n-1) + nq0.$
For complex flavors:
 $lwork \geq \max(3, mp0+nq0) .$
 Here
 $iroff = \text{mod}(ia-1, mb\_a), icoff = \text{mod}(ja-1, nb\_a),$
 $iarow = \text{indxg2p}(ia, mb\_a, MYROW, rsrc\_a, NPROW),$
 $iacol = \text{indxg2p}(ja, nb\_a, MYCOL, csrc\_a, NPCOL),$
 $mp0 = \text{numroc}(m+iroff, mb\_a, MYROW, iarow, NPROW),$
 $nq0 = \text{numroc}(n+icoff, nb\_a, MYCOL, iacol, NPCOL),$
 $LOCc(ja+n-1) = \text{numroc}(ja+n-1, nb\_a,$
 $MYCOL, csrc\_a, NPCOL),$ and *numroc*, *indxg2p* are
 ScaLAPACK tool functions; *MYROW*, *MYCOL*, *NPROW* and *NPCOL*
 can be determined by calling the subroutine
blacs_gridinfo.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

<i>a</i>	The elements on and above the diagonal of $\text{sub}(A)$ contain the $\min(m, n)$ -by- n upper trapezoidal matrix R (R is upper triangular if $m \geq n$); the elements below the diagonal, with the array <i>tau</i> , represent the orthogonal/unitary matrix Q as a product of elementary reflectors (see <i>Application Notes</i> below).
<i>ipiv</i>	(local) INTEGER. Array, DIMENSION $LOCc(ja+n-1)$. $ipiv(i) = k$, the local i -th column of $\text{sub}(A)*P$ was the global k -th column of $\text{sub}(A)$. <i>ipiv</i> is tied to the distributed matrix A .
<i>tau</i>	(local) REAL for psgeqpf DOUBLE PRECISION for pdgeqpf COMPLEX for pcgeqpf DOUBLE COMPLEX for pzgeqpf. Array, DIMENSION $LOCc(ja+\min(m, n)-1)$. Contains the scalar factor <i>tau</i> of elementary reflectors. <i>tau</i> is tied to the distributed matrix A .
<i>work(1)</i>	On exit, <i>work(1)</i> contains the minimum value of $lwork$ required for optimum performance.
<i>info</i>	(global) INTEGER. = 0, the execution is successful. < 0, if the i -th argument is an array and the j -entry had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(1) * H(2) * \dots * H(n),$$

Each $H(i)$ has the form

$$H = I - \tau v v'$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:m)$ is stored on exit in $A(ia+i-1:ia+m-1, ja+i-1)$.

The matrix P is represented in $jpvt$ as follows: if $jpvt(j) = i$ then the j -th column of P is the i -th canonical unit vector.

p?orgqr

Generates the orthogonal matrix Q of the QR factorization formed by p?geqrf.

Syntax

```
call psorgqr(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorgqr(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine generates the whole or part of m -by- n real distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal columns, which is defined as the first n columns of a product of k elementary reflectors of order m

$$Q = H(1) * H(2) * \dots * H(k)$$

as returned by p?geqrf.

Input Parameters

m	(global) INTEGER. The number of rows in the submatrix $\text{sub}(Q)$ ($m \geq 0$).
n	(global) INTEGER. The number of columns in the submatrix $\text{sub}(Q)$ ($m \geq n \geq 0$).
k	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q ($n \geq k \geq 0$).
a	(local) REAL for psorgqr

DOUBLE PRECISION for pdorgqr
 Pointer into the local memory to an array of local dimension
 (*lld_a*, *LOCc(ja+n-1)*). The *j*-th column must contain
 the vector which defines the elementary reflector $H(j)$,
 $ja \leq j \leq ja + k - 1$, as returned by [p?geqrf](#) in the *k* columns
 of its distributed matrix argument $A(ia:*, ja:ja+k-1)$.

ia, ja (global) INTEGER. The row and column indices in the global
 array *a* indicating the first row and the first column of the
 submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The
 array descriptor for the distributed matrix *A*.

tau (local)
 REAL for psorgqr
 DOUBLE PRECISION for pdorgqr
 Array, DIMENSION *LOCc(ja+k-1)*.
 Contains the scalar factor *tau(j)* of elementary reflectors
 $H(j)$ as returned by [p?geqrf](#). *tau* is tied to the distributed
 matrix *A*.

work (local)
 REAL for psorgqr
 DOUBLE PRECISION for pdorgqr
 Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of *work*.
 Must be at least $lwork \geq nb\_a * (nqa0 + mpa0 + nb\_a)$,
 where
 $iroffa = \text{mod}(ia-1, mb\_a)$, $icoffa = \text{mod}(ja-1,$
 $nb\_a)$,
 $iarow = \text{indxg2p}(ia, mb\_a, MYROW, rsrc\_a, NPROW)$,
 $iacol = \text{indxg2p}(ja, nb\_a, MYCOL, csrc\_a, NPCOL)$,
 $mpa0 = \text{numroc}(m+iroffa, mb\_a, MYROW, iarow,$
 $NPROW)$,
 $nqa0 = \text{numroc}(n+icoffa, nb\_a, MYCOL, iacol,$
 $NPCOL)$;
indxg2p and *numroc* are ScaLAPACK tool functions; MYROW,
 MYCOL, NPROW and NPCOL can be determined by calling the
 subroutine *blacs_gridinfo*.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>a</code>	Contains the local pieces of the m -by- n distributed matrix Q .
<code>work(1)</code>	On exit, <code>work(1)</code> contains the minimum value of $lwork$ required for optimum performance.
<code>info</code>	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -entry had an illegal value, then $info = -(i * 100 + j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

p?ungqr

Generates the complex unitary matrix Q of the QR factorization formed by `p?geqrf`.

Syntax

```
call pcungqr( m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzungqr( m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine generates the whole or part of m -by- n complex distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal columns, which is defined as the first n columns of a product of k elementary reflectors of order m

$$Q = H(1) * H(2) * \dots * H(k)$$

as returned by `p?geqrf`.

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows in the submatrix <code>sub(Q)</code> ; ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the submatrix <code>sub(Q)</code> ($m \geq n \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix <i>Q</i> ($n \geq k \geq 0$).
<i>a</i>	(local) COMPLEX for <code>pcungqr</code> DOUBLE COMPLEX for <code>pzungqr</code> Pointer into the local memory to an array of dimension (<code>lld_a</code> , <code>LOCc(ja+n-1)</code>). The <i>j</i> -th column must contain the vector which defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by <code>p?geqrf</code> in the <i>k</i> columns of its distributed matrix argument <code>A(ia:*, ja:ja+k-1)</code> .
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<code>dlen_</code>). The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) COMPLEX for <code>pcungqr</code> DOUBLE COMPLEX for <code>pzungqr</code> Array, DIMENSION <code>LOCc(ja+k-1)</code> . Contains the scalar factor <i>tau</i> (<i>j</i>) of elementary reflectors $H(j)$ as returned by <code>p?geqrf</code> . <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>work</i>	(local) COMPLEX for <code>pcungqr</code> DOUBLE COMPLEX for <code>pzungqr</code> Workspace array of dimension of <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, dimension of <i>work</i> , must be at least $lwork \geq nb\_a * (nqa0 + mpa0 + nb\_a)$, where $iroffa = \text{mod}(ia-1, mb\_a)$,

```

icoffa = mod(ja-1, nb_a),
iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
mpa0 = numroc(m+iroffa, mb_a, MYROW, iarow, NPROW),
nqa0 = numroc(n+icoffa, nb_a, MYCOL, iacol, NPCOL)
indxg2p and numroc are ScaLAPACK tool functions; MYROW,
MYCOL, NPROW and NPCOL can be determined by calling the
subroutine blacs_gridinfo.

```

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `p?xerbla`.

Output Parameters

a	Contains the local pieces of the m -by- n distributed matrix Q .
$work(1)$	On exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance.
$info$	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -entry had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

p?ormqr

Multiplies a general matrix by the orthogonal matrix Q of the QR factorization formed by p?geqrf.

Syntax

```

call psormqr( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)

call pdormqr( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)

```

Description

The routine overwrites the general real m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
$trans = 'T':$	$Q^T * \text{sub}(C)$	$\text{sub}(C) * Q^T$

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors

$$Q = H(1) H(2) \dots H(k)$$

as returned by `p?geqrf`. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

<i>side</i>	(global) CHARACTER ='L': Q or Q^T is applied from the left. ='R': Q or Q^T is applied from the right.
<i>trans</i>	(global) CHARACTER ='N', no transpose, Q is applied. ='T', transpose, Q^T is applied.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(C)$ ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(C)$ ($n \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: If $side = 'L'$, $m \geq k \geq 0$ If $side = 'R'$, $n \geq k \geq 0$.
<i>a</i>	(local) REAL for <code>psormqr</code> DOUBLE PRECISION for <code>pdormqr</code> . Pointer into the local memory to an array of dimension $(lld\_a, LOCC(ja+k-1))$. The j -th column must contain the vector which defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by <code>p?geqrf</code> in the k columns of

	its distributed matrix argument $A(ia:*, ja:ja+k-1)$. $A(ia:*, ja:ja+k-1)$ is modified by the routine but restored on exit.
	If $side = 'L'$, $lld\_a \geq \max(1, LOCr(ia+m-1))$
	If $side = 'R'$, $lld\_a \geq \max(1, LOCr(ia+n-1))$
ia, ja	(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix A , respectively.
$desca$	(global and local) INTEGER array, dimension $(dlen\_)$. The array descriptor for the distributed matrix A .
tau	(local) REAL for psormqr DOUBLE PRECISION for pdormqr Array, DIMENSION $LOCc(ja+k-1)$. Contains the scalar factor $tau(j)$ of elementary reflectors $H(j)$ as returned by p?geqrf. tau is tied to the distributed matrix A .
c	(local) REAL for psormqr DOUBLE PRECISION for pdormqr Pointer into the local memory to an array of local dimension $(lld\_c, LOCc(jc+n-1))$. Contains the local pieces of the distributed matrix $sub(C)$ to be factored.
ic, jc	(global) INTEGER. The row and column indices in the global array c indicating the first row and the first column of the submatrix C , respectively.
$descc$	(global and local) INTEGER array, dimension $(dlen\_)$. The array descriptor for the distributed matrix C .
$work$	(local) REAL for psormqr DOUBLE PRECISION for pdormqr. Workspace array of dimension of $lwork$.
$lwork$	(local or global) INTEGER, dimension of $work$, must be at least: if $side = 'L'$,

```

lwork ≥ max((nb_a*(nb_a-1))/2, (nqc0+mpc0)*nb_a
+ nb_a*nb_a
else if side = 'R',
lwork ≥ max((nb_a*(nb_a-1))/2,
(nqc0+max(npa0+numroc(numroc(n+icoffc, nb_a, 0,
0, NPCOL), nb_a, 0, 0, lcmq), mpc0))*nb_a) +
nb_a*nb_a
end if
where
lcmq = lcm/NPCOL with lcm = ilcm(NPROW, NPCOL),
iroffa = mod(ia-1, mb_a),
icoffa = mod(ja-1, nb_a),
iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
npa0= numroc(n+iroffa, mb_a, MYROW, iarow,
NPROW),
iroffc = mod(ic-1, mb_c),
icoffc = mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0= numroc(m+iroffc, mb_c, MYROW, icrow,
NPROW),
nqc0= numroc(n+icoffc, nb_c, MYCOL, iccol,
NPCOL),
ilcm, indxg2p and numroc are ScaLAPACK tool functions;
MYROW, MYCOL, NPROW and NPCOL can be determined by
calling the subroutine blacs_gridinfo.

```

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by *pzerbla*.

Output Parameters

<i>c</i>	Overwritten by the product $Q \cdot \text{sub}(C)$, or $Q^T \cdot \text{sub}(C)$, or $\text{sub}(C) \cdot Q^T$, or $\text{sub}(C) \cdot Q$.
<i>work</i> (1)	On exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.

info (global) INTEGER.
 = 0: the execution is successful.
 < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?unmqr

Multiplies a complex matrix by the unitary matrix Q of the QR factorization formed by p?geqrf.

Syntax

```
call pcunmqr( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info )

call pzunmqr( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info )
```

Description

The routine overwrites the general complex m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
$trans = 'T':$	$Q^H * \text{sub}(C)$	$\text{sub}(C) * Q^H$

where Q is a complex unitary distributed matrix defined as the product of k elementary reflectors

$Q = H(1) H(2) \dots H(k)$ as returned by p?geqrf. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

side (global) CHARACTER
 = 'L': Q or Q^H is applied from the left.
 = 'R': Q or Q^H is applied from the right.

trans (global) CHARACTER
 = 'N', no transpose, Q is applied.

	= 'C', conjugate transpose, Q^H is applied.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix sub(<i>c</i>) ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix sub(<i>c</i>) ($n \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix <i>Q</i> . Constraints: If <i>side</i> = 'L', $m \geq k \geq 0$ If <i>side</i> = 'R', $n \geq k \geq 0$.
<i>a</i>	(local) COMPLEX for pcunmqr DOUBLE COMPLEX for pzunmqr. Pointer into the local memory to an array of dimension (<i>lld_a</i> , <i>LOCc</i> (<i>ja+k-1</i>)). The <i>j</i> -th column must contain the vector which defines the elementary reflector $H(j)$, $ja \leq j \leq ja + k - 1$, as returned by p?geqrf in the <i>k</i> columns of its distributed matrix argument $A(ia:*, ja:ja+k-1)$. $A(ia:*, ja:ja+k-1)$ is modified by the routine but restored on exit. If <i>side</i> = 'L', $lld\_a \geq \max(1, LOCr(ia+m-1))$ If <i>side</i> = 'R', $lld\_a \geq \max(1, LOCr(ia+n-1))$
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) COMPLEX for pcunmqr DOUBLE COMPLEX for pzunmqr Array, DIMENSION <i>LOCc</i> (<i>ja+k-1</i>). Contains the scalar factor <i>tau</i> (<i>j</i>) of elementary reflectors $H(j)$ as returned by p?geqrf . <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>c</i>	(local)

COMPLEX for pcunmqr
DOUBLE COMPLEX for pzunmqr.
Pointer into the local memory to an array of local dimension
(*lld_c*, *LOCc(jc+n-1)*).
Contains the local pieces of the distributed matrix sub(*c*)
to be factored.

ic, jc (global) INTEGER. The row and column indices in the global
array *c* indicating the first row and the first column of the
submatrix *c*, respectively.

desc (global and local) INTEGER array, dimension (*dlen_*). The
array descriptor for the distributed matrix *c*.

work (local)
COMPLEX for pcunmqr
DOUBLE COMPLEX for pzunmqr.
Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at
least:
If *side* = 'L',
 $lwork \geq \max((nb\_a * (nb\_a - 1)) / 2, (nqc0 + mpc0) * nb\_a) + nb\_a * nb\_a$
else if *side* = 'R',
 $lwork \geq \max((nb\_a * (nb\_a - 1)) / 2, (nqc0 + \max(npa0 + \text{numroc}(\text{numroc}(n + ioffset, nb\_a, 0, 0, NPCOL), nb\_a, 0, 0, lcmq), mpc0)) * nb\_a) + nb\_a * nb\_a$
end if
where
 $lcmq = lcm / NPCOL$ with $lcm = ilcm(NPROW, NPCOL)$,
 $iroffa = \text{mod}(ia - 1, mb\_a)$,
 $icoffa = \text{mod}(ja - 1, nb\_a)$,
 $iarow = \text{indxg2p}(ia, mb\_a, MYROW, rsrc\_a, NPROW)$,
 $npa0 = \text{numroc}(n + iroffa, mb\_a, MYROW, iarow, NPROW)$,
 $iroffc = \text{mod}(ic - 1, mb\_c)$,
 $icoffc = \text{mod}(jc - 1, nb\_c)$,
 $icrow = \text{indxg2p}(ic, mb\_c, MYROW, rsrc\_c, NPROW)$,
 $iccol = \text{indxg2p}(jc, nb\_c, MYCOL, csrc\_c, NPCOL)$,

```

mpc0 = numroc(m+iroffc, mb_c, MYROW, icrow,
NPROW),
nqc0 = numroc(n+icoffc, nb_c, MYCOL, iccol,
NPCOL),
ilcm, indxcg2p and numroc are ScaLAPACK tool functions;
MYROW, MYCOL, NPROW and NPCOL can be determined by
calling the subroutine blacs_gridinfo.
If lwork = -1, then lwork is global input and a workspace
query is assumed; the routine only calculates the minimum
and optimal size for all work arrays. Each of these values
is returned in the first entry of the corresponding work array,
and no error message is issued by p?erbla.

```

Output Parameters

<i>c</i>	Overwritten by the product $Q^* \text{sub}(C)$, or $Q^H \text{sub}(C)$, or $\text{sub}(C) * Q^H$, or $\text{sub}(C) * Q$.
<i>work(1)</i>	On exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = - (<i>i</i> * 100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

p?gelqf

Computes the LQ factorization of a general rectangular matrix.

Syntax

```

call psgelqf( m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgelqf( m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgelqf( m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgelqf( m, n, a, ia, ja, desca, tau, work, lwork, info)

```

Description

The routine computes the LQ factorization of a real/complex distributed m -by- n matrix $\text{sub}(A) = A(\text{ia}:\text{ia}+m-1, \text{ia}:\text{ia}+n-1) = L^*Q$.

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows in the submatrix $\text{sub}(Q)$ ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the submatrix $\text{sub}(Q)$ ($n \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q ($n \geq k \geq 0$).
<i>a</i>	(local) REAL for psgelqf DOUBLE PRECISION for pdgelqf COMPLEX for pcgelqf DOUBLE COMPLEX for pzgelqf Pointer into the local memory to an array of local dimension $(\text{lld\_a}, \text{LOCc}(\text{ja}+n-1))$. Contains the local pieces of the distributed matrix $\text{sub}(A)$ to be factored.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix $A((\text{ia}:\text{ia}+m-1, \text{ia}:\text{ia}+n-1))$, respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (dlen_) . The array descriptor for the distributed matrix <i>A</i> .
<i>work</i>	(local) REAL for psgelqf DOUBLE PRECISION for pdgelqf COMPLEX for pcgelqf DOUBLE COMPLEX for pzgelqf Workspace array of dimension of <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, dimension of <i>work</i> , must be at least $\text{lwork} \geq \text{mb\_a} * (\text{mp0} + \text{nq0} + \text{mb\_a})$, where $\text{iroff} = \text{mod}(\text{ia}-1, \text{mb\_a})$,

```
icoff = mod(ja-1, nb_a),
iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
mp0 = numroc(m+iroff, mb_a, MYROW, iarow, NPROW),
nq0 = numroc(n+icoff, nb_a, MYCOL, iacol, NPCOL)
indxg2p and numroc are ScaLAPACK tool functions; MYROW,
MYCOL, NPROW and NPCOL can be determined by calling the
subroutine blacs_gridinfo.
```

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

<i>a</i>	The elements on and below the diagonal of sub(<i>A</i>) contain the <i>m</i> by min(<i>m</i> , <i>n</i>) lower trapezoidal matrix <i>L</i> (<i>L</i> is lower trapezoidal if <i>m</i> ≤ <i>n</i>); the elements above the diagonal, with the array <i>tau</i> , represent the orthogonal/unitary matrix <i>Q</i> as a product of elementary reflectors (see <i>Application Notes</i> below).
<i>tau</i>	(local) REAL for psgelqf DOUBLE PRECISION for pdgelqf COMPLEX for pcgelqf DOUBLE COMPLEX for pzgelqf Array, DIMENSION <i>LOCr</i> (<i>ia</i> +min(<i>m</i> , <i>n</i>)-1)). Contains the scalar factors of elementary reflectors. <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>work</i> (1)	On exit, <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = - (<i>i</i> * 100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(ia+k-1) * H(ia+k-2) * \dots * H(ia),$$

where $k = \min(m, n)$

Each $H(i)$ has the form

$$H(i) = I - \tau v v'$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:n)$ is stored on exit in $A(ia+i-1:ia+i-1, ja+n-1)$, and τ in $\tau(ia+i-1)$.

p?orglq

Generates the real orthogonal matrix Q of the LQ factorization formed by p?gelqf.

Syntax

```
call psorglq( m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorglq( m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine generates the whole or part of m -by- n real distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the first m rows of a product of k elementary reflectors of order n

$$Q = H(k) * \dots * H(2) * H(1)$$

as returned by p?gelqf.

Input Parameters

m	(global) INTEGER. The number of rows in the submatrix $\text{sub}(Q)$; ($m \geq 0$).
n	(global) INTEGER. The number of columns in the submatrix $\text{sub}(Q)$ ($n \geq m \geq 0$).

<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q ($m \geq k \geq 0$).
<i>a</i>	(local) REAL for psorglq DOUBLE PRECISION for pdorglq Pointer into the local memory to an array of local dimension (<i>lld_a</i> , <i>LOCc</i> (<i>ja+n-1</i>)). On entry, the <i>i</i> -th row must contain the vector which defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by p?gelqf in the <i>k</i> rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix $A((ia:ia+m-1, ja:ja+n-1)$, respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>work</i>	(local) REAL for psorglq DOUBLE PRECISION for pdorglq Workspace array of dimension of <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, dimension of <i>work</i> , must be at least $lwork \geq mb\_a * (mpa0 + nqa0 + mb\_a)$, where $iroffa = \text{mod}(ia-1, mb\_a)$, $icoffa = \text{mod}(ja-1, nb\_a)$, $iarow = \text{indxg2p}(ia, mb\_a, MYROW, rsrc\_a, NPROW)$, $iacol = \text{indxg2p}(ja, nb\_a, MYCOL, csrc\_a, NPCOL)$, $mpa0 = \text{numroc}(m + iroffa, mb\_a, MYROW, iarow, NPROW)$, $nqa0 = \text{numroc}(n + icoffa, nb\_a, MYCOL, iacol, NPCOL)$ <i>indxg2p</i> and <i>numroc</i> are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine <i>blacs_gridinfo</i> . If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla .

Output Parameters

<i>a</i>	Contains the local pieces of the m -by- n distributed matrix Q to be factored.
<i>tau</i>	(local) REAL for psorglq DOUBLE PRECISION for pdorglq Array, DIMENSION $LOCr(ia+k-1)$. Contains the scalar factors <i>tau</i> of elementary reflectors $H(i)$. <i>tau</i> is tied to the distributed matrix A .
<i>work</i> (1)	On exit, <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -entry had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

p?unglq

Generates the unitary matrix Q of the LQ factorization formed by p?gelqf.

Syntax

```
call pcunglq( m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzunglq( m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine generates the whole or part of m -by- n complex distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the first m rows of a product of k elementary reflectors of order n

$Q = (H(k))^H \dots (H(2))^H (H(1))^H$ as returned by p?gelqf.

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows in the submatrix <code>sub(Q)</code> ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the submatrix <code>sub(Q)</code> ($n \geq m \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix <i>Q</i> ($m \geq k \geq 0$).
<i>a</i>	(local) COMPLEX for <code>pcunglq</code> DOUBLE COMPLEX for <code>pzunglq</code> Pointer into the local memory to an array of local dimension (<code>lld_a</code> , <code>LOCc(ja+n-1)</code>). On entry, the <i>i</i> -th row must contain the vector which defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by <code>p?gelqf</code> in the <i>k</i> rows of its distributed matrix argument <code>A(ia:ia+k-1, ja:*)</code> .
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <code>A(ia:ia+m-1, ja:ja+n-1)</code> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<code>dlen_</code>). The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) COMPLEX for <code>pcunglq</code> DOUBLE COMPLEX for <code>pzunglq</code> Array, DIMENSION <code>LOCr(ia+k-1)</code> . Contains the scalar factors <i>tau</i> of elementary reflectors $H(i)$. <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>work</i>	(local) COMPLEX for <code>pcunglq</code> DOUBLE COMPLEX for <code>pzunglq</code> Workspace array of dimension of <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, dimension of <i>work</i> , must be at least $lwork \geq mb\_a * (mpa0 + nqa0 + mb\_a)$, where $iroffa = \text{mod}(ia-1, mb\_a)$, $icoffa = \text{mod}(ja-1, nb\_a)$,

```

iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
mpa0 = numroc(m+iroffa, mb_a, MYROW, iarow,
NPROW),
nqa0 = numroc(n+icoffa, nb_a, MYCOL, iacol,
NPCOL)

```

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `p?erbla`.

Output Parameters

<code>a</code>	Contains the local pieces of the m -by- n distributed matrix Q to be factored.
<code>work(1)</code>	On exit, <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -entry had an illegal value, then <code>info</code> = $-(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

p?ormlq

Multiplies a general matrix by the orthogonal matrix Q of the LQ factorization formed by p?gelqf.

Syntax

```

call psormlq( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, work,
lwork, info)

call pdormlq( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, work,
lwork, info)

```

Description

The routine overwrites the general real m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
$trans = 'T':$	$Q^T * \text{sub}(C)$	$\text{sub}(C) * Q^T$

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors

$$Q = H(k) \dots H(2) H(1)$$

as returned by `p?gelqf`. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

$side$	(global) CHARACTER $= 'L'$: Q or Q^T is applied from the left. $= 'R'$: Q or Q^T is applied from the right.
$trans$	(global) CHARACTER $= 'N'$, no transpose, Q is applied. $= 'T'$, transpose, Q^T is applied.
m	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(C)$ ($m \geq 0$).
n	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(C)$ ($n \geq 0$).
k	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: If $side = 'L'$, $m \geq k \geq 0$ If $side = 'R'$, $n \geq k \geq 0$.
a	(local) REAL for <code>psormlq</code> DOUBLE PRECISION for <code>pdormlq</code> . Pointer into the local memory to an array of dimension $(lld\_a, LOCC(ja+m-1))$, if $side = 'L'$ and $(lld\_a, LOCC(ja+n-1))$, if $side = 'R'$. The i -th row must contain

	the vector which defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by <code>p?gelqf</code> in the k rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$. $A(ia:ia+k-1, ja:*)$ is modified by the routine but restored on exit.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) REAL for <code>psormlq</code> DOUBLE PRECISION for <code>pdormlq</code> Array, DIMENSION $LOCc(ja+k-1)$. Contains the scalar factor $\tau(i)$ of elementary reflectors $H(i)$ as returned by <code>p?gelqf</code>. <i>tau</i> is tied to the distributed matrix <i>A</i>.
<i>c</i>	(local) REAL for <code>psormlq</code> DOUBLE PRECISION for <code>pdormlq</code> Pointer into the local memory to an array of local dimension ($lld\_c, LOCc(jc+n-1)$). Contains the local pieces of the distributed matrix sub(<i>c</i>) to be factored.
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the global array <i>c</i> indicating the first row and the first column of the submatrix <i>C</i> , respectively.
<i>descc</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>C</i> .
<i>work</i>	(local) REAL for <code>psormlq</code> DOUBLE PRECISION for <code>pdormlq</code>. Workspace array of dimension of <i>lwork</i>.
<i>lwork</i>	(local or global) INTEGER, dimension of the array <i>work</i>; must be at least: If <i>side</i> = 'L',

```

lwork ≥ max((mb_a*(mb_a-1))/2, (mpc0+max mqa0)+
numroc(numroc(m + iroffc, mb_a, 0, 0, NPROW),
mb_a, 0, 0, lcmp), nqc0))* mb_a) + mb_a*mb_a
else if side = 'R',
lwork ≥ max((mb_a*(mb_a-1))/2, (mpc0+nqc0)*mb_a
+ mb_a*mb_a
end if
where
lcmp = lcm/NPROW with lcm = ilcm (NPROW, NPCOL),
iroffa = mod(ia-1, mb_a),
icoffa = mod(ja-1, nb_a),
iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
mqa0 = numroc(m+icoffa, nb_a, MYCOL, iacol,
NPCOL),
iroffc = mod(ic-1, mb_c),
icoffc = mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(m+iroffc, mb_c, MYROW, icrow,
NPROW),
nqc0 = numroc(n+icoffc, nb_c, MYCOL, iccol,
NPCOL),
ilcm, indxg2p and numroc are ScaLAPACK tool functions;
MYROW, MYCOL, NPROW and NPCOL can be determined by
calling the subroutine blacs_gridinfo.

```

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by *pxerbla*.

Output Parameters

<i>c</i>	Overwritten by the product $Q \cdot \text{sub}(c)$, or $Q' \cdot \text{sub}(c)$, or $\text{sub}(c) \cdot Q'$, or $\text{sub}(c) \cdot Q$
<i>work</i> (1)	On exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER.

= 0: the execution is successful.
 < 0: if the i -th argument is an array and the j -entry had an illegal value, then $info = - (i * 100 + j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

p?unmlq

Multiplies a general matrix by the unitary matrix Q of the LQ factorization formed by p?gelqf.

Syntax

```
call pcunmlq( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info )

call pzunmlq( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info )
```

Description

The routine overwrites the general complex m -by- n distributed matrix $sub(C) = C$ ($ic:ic+m-1, jc:jc+n-1$) with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * sub(C)$	$sub(C) * Q$
$trans = 'T':$	$Q^H * sub(C)$	$sub(C) * Q^H$

where Q is a complex unitary distributed matrix defined as the product of k elementary reflectors

$$Q = H(k)' \dots H(2)' H(1)'$$

as returned by p?gelqf. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

side (global) CHARACTER
 = 'L': Q or Q^H is applied from the left.
 = 'R': Q or Q^H is applied from the right.

trans (global) CHARACTER
 = 'N', no transpose, Q is applied.
 = 'C', conjugate transpose, Q^H is applied.

<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix <code>sub(c)</code> ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix <code>sub(c)</code> ($n \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix <i>Q</i> . Constraints: If <i>side</i> = 'L', $m \geq k \geq 0$ If <i>side</i> = 'R', $n \geq k \geq 0$.
<i>a</i>	(local) COMPLEX for pcunmlq DOUBLE COMPLEX for pzunmlq. Pointer into the local memory to an array of dimension (<i>lld_a</i> , <i>LOCc(ja+m-1)</i>), if <i>side</i> = 'L', and (<i>lld_a</i> , <i>LOCc(ja+n-1)</i>), if <i>side</i> = 'R', where $lld\_a \geq \max(1, LOCr(ia+k-1))$. The <i>i</i> -th column must contain the vector which defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by <code>p?gelqf</code> in the <i>k</i> rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$. $A(ia:ia+k-1, ja:*)$ is modified by the routine but restored on exit.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) COMPLEX for pcunmlq DOUBLE COMPLEX for pzunmlq Array, DIMENSION <i>LOCc(ia+k-1)</i> . Contains the scalar factor <i>tau(i)</i> of elementary reflectors $H(i)$ as returned by <code>p?gelqf</code> . <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>c</i>	(local) COMPLEX for pcunmlq DOUBLE COMPLEX for pzunmlq.

	Pointer into the local memory to an array of local dimension $(lld\_c, LOCc(jc+n-1))$. Contains the local pieces of the distributed matrix sub(c) to be factored.
ic, jc	(global) INTEGER. The row and column indices in the global array c indicating the first row and the first column of the submatrix c, respectively.
$descc$	(global and local) INTEGER array, dimension $(dlen\_)$. The array descriptor for the distributed matrix c.
$work$	(local) COMPLEX for pcunmlq DOUBLE COMPLEX for pzunmlq. Workspace array of dimension of $lwork$.
$lwork$	(local or global) INTEGER, dimension of the array $work$; must be at least: If $side = 'L'$, $lwork \geq \max((mb\_a*(mb\_a-1))/2, (mpc0 + \max mqa0) +$ $\text{numroc}(\text{numroc}(m + iroffc, mb\_a, 0, 0, NPROW),$ $mb\_a, 0, 0, lcm), nqc0))*mb\_a) + mb\_a*mb\_a$ else if $side = 'R'$, $lwork \geq \max((mb\_a*(mb\_a-1))/2, (mpc0 +$ $nqc0)*mb\_a + mb\_a*mb\_a$ end if where $lcm = lcm/NPROW \text{ with } lcm = ilcm(NPROW, NPCOL),$ $iroffa = \text{mod}(ia-1, mb\_a),$ $icoffa = \text{mod}(ja-1, nb\_a),$ $iacol = \text{indxg2p}(ja, nb\_a, MYCOL, csrc\_a, NPCOL),$ $mqa0 = \text{numroc}(m + icoffa, nb\_a, MYCOL, iacol,$ $NPCOL),$ $iroffc = \text{mod}(ic-1, mb\_c),$ $icoffc = \text{mod}(jc-1, nb\_c),$ $icrow = \text{indxg2p}(ic, mb\_c, MYROW, rsrc\_c, NPROW),$ $iccol = \text{indxg2p}(jc, nb\_c, MYCOL, csrc\_c, NPCOL),$ $mpc0 = \text{numroc}(m + iroffc, mb\_c, MYROW, icrow,$ $NPROW),$

`nqc0 = numroc(n+icoffc, nb_c, MYCOL, iccol, NPCOL),`
`ilcm, indxg2p` and `numroc` are ScaLAPACK tool functions;
`MYROW, MYCOL, NPROW` and `NPCOL` can be determined by
calling the subroutine `blacs_gridinfo`.
If `lwork = -1`, then `lwork` is global input and a workspace
query is assumed; the routine only calculates the minimum
and optimal size for all work arrays. Each of these values
is returned in the first entry of the corresponding work array,
and no error message is issued by `pxerbla`.

Output Parameters

<code>c</code>	Overwritten by the product $Q \cdot \text{sub}(c)$, or $Q' \cdot \text{sub}(c)$, or $\text{sub}(c) \cdot Q'$, or $\text{sub}(c) \cdot Q$
<code>work(1)</code>	On exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <code>info = - (i* 100+j)</code> , if the <i>i</i> -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

p?geqlf

Computes the QL factorization of a general matrix.

Syntax

```
call psgeqlf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgeqlf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgeqlf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgeqlf(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine forms the QL factorization of a real/complex distributed m -by- n matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) = Q^*L$.

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows in the submatrix $\text{sub}(Q)$; ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the submatrix $\text{sub}(Q)$ ($n \geq 0$).
<i>a</i>	(local) REAL for psgeqlf DOUBLE PRECISION for pdgeqlf COMPLEX for pcgeqlf DOUBLE COMPLEX for pzgeqlf Pointer into the local memory to an array of local dimension $(lld\_a, LOCC(ja+n-1))$. Contains the local pieces of the distributed matrix $\text{sub}(A)$ to be factored.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix $A((ia:ia+m-1, ja:ja+n-1))$, respectively.
<i>desca</i>	(global and local) INTEGER array, dimension $(dlen\_)$. The array descriptor for the distributed matrix <i>A</i> .
<i>work</i>	(local) REAL for psgeqlf DOUBLE PRECISION for pdgeqlf COMPLEX for pcgeqlf DOUBLE COMPLEX for pzgeqlf Workspace array of dimension of <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, dimension of <i>work</i> , must be at least $lwork \geq nb\_a * (mp0 + nq0 + nb\_a)$, where $iroff = \text{mod}(ia-1, mb\_a),$ $icoff = \text{mod}(ja-1, nb\_a),$ $iarow = \text{indxg2p}(ia, mb\_a, MYROW, rsrc\_a, NPROW),$ $iacol = \text{indxg2p}(ja, nb\_a, MYCOL, csrc\_a, NPCOL),$ $mp0 = \text{numroc}(m+iroff, mb\_a, MYROW, iarow, NPROW),$

`nq0 = numroc(n+icoff, nb_a, MYCOL, iacol, NPCOL)`
`numroc` and `indxg2p` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>a</code>	On exit, if $m \geq n$, the lower triangle of the distributed submatrix $A(ia+m-n:ia+m-1, ja:ja+n-1)$ contains the n -by- n lower triangular matrix L ; if $m \leq n$, the elements on and below the $(n-m)$ -th superdiagonal contain the m -by- n lower trapezoidal matrix L ; the remaining elements, with the array <code>tau</code> , represent the orthogonal/unitary matrix Q as a product of elementary reflectors (see <i>Application Notes</i> below).
<code>tau</code>	(local) REAL for psgeqlf DOUBLE PRECISION for pdgeqlf COMPLEX for pcgeqlf DOUBLE COMPLEX for pzgeqlf Array, DIMENSION <code>LOCc(ja+n-1)</code> . Contains the scalar factors of elementary reflectors. <code>tau</code> is tied to the distributed matrix A .
<code>work(1)</code>	On exit, <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -entry had an illegal value, then <code>info</code> = $-(i * 100 + j)$, if the i -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(ja+k-1) * \dots * H(ja+1) * H(ja),$$

where $k = \min(m, n)$

Each $H(i)$ has the form

$$H(i) = I - \tau v v'$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(m-k+i+1:m) = 0$ and $v(m-k+i) = 1$; $v(m-k+i-1)$ is stored on exit in $A(ia+ia+m-k+i-2, ja+n-k+i-1)$, and τ in $\tau(ja+n-k+i-1)$.

p?orgql

Generates the orthogonal matrix Q of the QL factorization formed by p?geqlf.

Syntax

```
call psorgql( m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorgql( m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine generates the whole or part of m -by- n real distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the first m rows of a product of k elementary reflectors of order n

$$Q = H(k) * \dots * H(2) * H(1)$$

as returned by p?geqlf.

Input Parameters

m	(global) INTEGER. The number of rows in the submatrix $\text{sub}(Q)$; ($m \geq 0$).
n	(global) INTEGER. The number of columns in the submatrix $\text{sub}(Q)$ ($m \geq n \geq 0$).

<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q ($n \geq k \geq 0$).
<i>a</i>	(local) REAL for psorgql DOUBLE PRECISION for pdorgql Pointer into the local memory to an array of local dimension (<i>lld_a</i> , <i>LOCc</i> (<i>ja+n-1</i>)). On entry, the <i>j</i> -th column must contain the vector which defines the elementary reflector $H(j)$, $ja+n-k \leq j \leq ja+n-1$, as returned by p?geqlf in the <i>k</i> columns of its distributed matrix argument $A(ia:ja+n-k:ja+n-1)$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix $A((ia:ia+m-1, ja:ja+n-1)$, respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) REAL for psorgql DOUBLE PRECISION for pdorgql Array, DIMENSION <i>LOCc</i> (<i>ja+n-1</i>). Contains the scalar factors $\tau(j)$ of elementary reflectors $H(j)$. <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>work</i>	(local) REAL for psorgql DOUBLE PRECISION for pdorgql Workspace array of dimension of <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, dimension of <i>work</i> , must be at least $lwork \geq nb\_a * (nqa0 + mpa0 + nb\_a)$, where $iroffa = \text{mod}(ia-1, mb\_a),$ $icoffa = \text{mod}(ja-1, nb\_a),$ $iarow = \text{indxg2p}(ia, mb\_a, MYROW, rsrc\_a, NPROW),$ $iacol = \text{indxg2p}(ja, nb\_a, MYCOL, csrc\_a, NPCOL),$ $mpa0 = \text{numroc}(m + iroffa, mb\_a, MYROW, iarow, NPROW),$ $nqa0 = \text{numroc}(n + icoffa, nb\_a, MYCOL, iacol, NPCOL)$

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `p?xerbla`.

Output Parameters

<code>a</code>	Contains the local pieces of the m -by- n distributed matrix Q to be factored.
<code>work(1)</code>	On exit, <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -entry had an illegal value, then <code>info</code> = $-(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

p?ungql

Generates the unitary matrix Q of the Q^L factorization formed by `p?geqlf`.

Syntax

```
call pcungql( m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzungql( m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine generates the whole or part of m -by- n complex distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the first n columns of a product of k elementary reflectors of order m

$Q = (H(k))^H \dots (H(2))^H (H(1))^H$ as returned by `p?geqlf`.

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows in the submatrix $\text{sub}(Q)$ ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the submatrix $\text{sub}(Q)$ ($m \geq n \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q ($n \geq k \geq 0$).
<i>a</i>	(local) COMPLEX for pcungql DOUBLE COMPLEX for pzungql Pointer into the local memory to an array of local dimension (lld_a , $LOCc(ja+n-1)$). On entry, the j -th column must contain the vector which defines the elementary reflector $H(j)$, $ja+n-k \leq j \leq ja+n-1$, as returned by p?geqlf in the k columns of its distributed matrix argument $A(ia:*, ja+n-k: ja+n-1)$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.
<i>desca</i>	(global and local) INTEGER array, dimension ($dlen\_$). The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) COMPLEX for pcungql DOUBLE COMPLEX for pzungql Array, DIMENSION $LOCr(ia+n-1)$. Contains the scalar factors $\tau(j)$ of elementary reflectors $H(j)$. τ is tied to the distributed matrix <i>A</i> .
<i>work</i>	(local) COMPLEX for pcungql DOUBLE COMPLEX for pzungql Workspace array of dimension of <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, dimension of <i>work</i> , must be at least $lwork \geq nb\_a * (nqa0 + mpa0 + nb\_a)$, where $iroffa = \text{mod}(ia-1, mb\_a)$, $icoffa = \text{mod}(ja-1, nb\_a)$,


```

iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
mpa0 = numroc(m+iroffa, mb_a, MYROW, iarow,
NPROW),
nqa0 = numroc(n+icoffa, nb_a, MYCOL, iacol, NPCOL)
indxg2p and numroc are ScaLAPACK tool functions; MYROW,
MYCOL, NPROW and NPCOL can be determined by calling the
subroutine blacs_gridinfo.

```

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by p_xerbla.

Output Parameters

a	Contains the local pieces of the m -by- n distributed matrix Q to be factored.
$work(1)$	On exit, $work(1)$ contains the minimum value of $lwork$ required for optimum performance.
$info$	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -entry had an illegal value, then $info = - (i * 100 + j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

p_?ormql

Multiplies a general matrix by the orthogonal matrix Q of the QL factorization formed by p_?geqlf.

Syntax

```

call psormql( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)

call pdormql( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)

```

Description

The routine overwrites the general real m -by- n distributed matrix $\text{sub}(C) = C$ ($ic:ic+m-1, jc:jc+n-1$) with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
$trans = 'T':$	$Q^T * \text{sub}(C)$	$\text{sub}(C) * Q^T$

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors

$$Q = H(k) \dots H(2) H(1)$$

as returned by `p?geqlf`. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

<i>side</i>	(global) CHARACTER = 'L': Q or Q^T is applied from the left. = 'R': Q or Q^T is applied from the right.
<i>trans</i>	(global) CHARACTER = 'N', no transpose, Q is applied. = 'T', transpose, Q^T is applied.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(C)$ ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(C)$ ($n \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: If $side = 'L'$, $m \geq k \geq 0$ If $side = 'R'$, $n \geq k \geq 0$.
<i>a</i>	(local) REAL for <code>psormql</code> DOUBLE PRECISION for <code>pdormql</code> . Pointer into the local memory to an array of dimension ($lld\_a, LOCC(ja+k-1)$). The j -th column must contain the vector which defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by <code>p?gelqf</code> in the k columns of

	its distributed matrix argument $A(ia:*, ja:ja+k-1)$. $A(ia:*, ja:ja+k-1)$ is modified by the routine but restored on exit.
	If $side = 'L'$, $lld\_a \geq \max(1, LOCr(ia+m-1))$,
	If $side = 'R'$, $lld\_a \geq \max(1, LOCr(ia+n-1))$.
ia, ja	(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix A , respectively.
$desca$	(global and local) INTEGER array, dimension $(dlen\_)$. The array descriptor for the distributed matrix A .
tau	(local) REAL for psormql DOUBLE PRECISION for pdormql. Array, DIMENSION $LOCc(ja+n-1)$). Contains the scalar factor $tau(j)$ of elementary reflectors $H(j)$ as returned by p?geqlf. tau is tied to the distributed matrix A .
c	(local) REAL for psormql DOUBLE PRECISION for pdormql. Pointer into the local memory to an array of local dimension $(lld\_c, LOCc(jc+n-1))$. Contains the local pieces of the distributed matrix sub(c) to be factored.
ic, jc	(global) INTEGER. The row and column indices in the global array c indicating the first row and the first column of the submatrix C , respectively.
$descc$	(global and local) INTEGER array, dimension $(dlen\_)$. The array descriptor for the distributed matrix C .
$work$	(local) REAL for psormql DOUBLE PRECISION for pdormql. Workspace array of dimension of $lwork$.
$lwork$	(local or global) INTEGER, dimension of $work$, must be at least: If $side = 'L'$,

```

lwork ≥ max((nb_a*(nb_a-1))/2, (nqc0+mpc0)*nb_a
+ nb_a*nb_a
else if side = 'R',
lwork ≥ max((nb_a*(nb_a-1))/2, (nqc0+max npa0)+
numroc(numroc(n+icoffc, nb_a, 0, 0, NPCOL),
nb_a, 0, 0, lcmq), mpc0))*nb_a) + nb_a*nb_a
end if
where
lcmp = lcm/NPCOL with lcm = ilcm (NPROW, NPCOL),
iroffa = mod(ia-1, mb_a),
icoffa = mod(ja-1, nb_a),
iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
npa0 = numroc(n + iroffa, mb_a, MYROW, iarow,
NPROW),
iroffc = mod(ic-1, mb_c),
icoffc = mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(m+iroffc, mb_c, MYROW, icrow,
NPROW),
nqc0 = numroc(n+icoffc, nb_c, MYCOL, iccol,
NPCOL),
ilcm, indxg2p and numroc are ScaLAPACK tool functions;
MYROW, MYCOL, NPROW and NPCOL can be determined by
calling the subroutine blacs_gridinfo.

```

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by *pserbla*.

Output Parameters

<i>c</i>	Overwritten by the product $Q^* \text{sub}(c)$, or $Q^* \text{sub}(c)$, or $\text{sub}(c)^* Q'$, or $\text{sub}(c)^* Q$
<i>work</i> (1)	On exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER.

= 0: the execution is successful.
 < 0: if the i -th argument is an array and the j -entry had an illegal value, then $info = - (i * 100 + j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

p?unmql

Multiplies a general matrix by the unitary matrix Q of the QL factorization formed by p?geqlf.

Syntax

```
call pcunmql( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)

call pzunmql( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

Description

The routine overwrites the general complex m -by- n distributed matrix $\text{sub}(C) = C$ ($ic:ic+m-1, jc:jc+n-1$) with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
$trans = 'C':$	$Q^H * \text{sub}(C)$	$\text{sub}(C) * Q^H$

where Q is a complex unitary distributed matrix defined as the product of k elementary reflectors

$$Q = H(k)' \dots H(2)' H(1)'$$

as returned by p?geqlf. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

side (global) CHARACTER
 = 'L': Q or Q^H is applied from the left.
 = 'R': Q or Q^H is applied from the right.

trans (global) CHARACTER
 = 'N', no transpose, Q is applied.
 = 'C', conjugate transpose, Q^H is applied.

<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix <code>sub(c)</code> ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix <code>sub(c)</code> ($n \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix <i>Q</i> . Constraints: If <i>side</i> = 'L', $m \geq k \geq 0$ If <i>side</i> = 'R', $n \geq k \geq 0$.
<i>a</i>	(local) COMPLEX for <code>pcunmql</code> DOUBLE COMPLEX for <code>pzunmql</code> . Pointer into the local memory to an array of dimension (<i>lld_a</i> , <i>LOCc(ja+k-1)</i>). The <i>j</i> -th column must contain the vector which defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by <code>p?geqlf</code> in the <i>k</i> columns of its distributed matrix argument $A(ia:*, ja:ja+k-1)$. $A(ia:*, ja:ja+k-1)$ is modified by the routine but restored on exit. If <i>side</i> = 'L', $lld\_a \geq \max(1, LOCr(ia+m-1))$, If <i>side</i> = 'R', $lld\_a \geq \max(1, LOCr(ia+n-1))$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) COMPLEX for <code>pcunmql</code> DOUBLE COMPLEX for <code>pzunmql</code> Array, DIMENSION <i>LOCc(ia+n-1)</i> . Contains the scalar factor <i>tau(j)</i> of elementary reflectors $H(j)$ as returned by <code>p?geqlf</code> . <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>c</i>	(local) COMPLEX for <code>pcunmql</code>

DOUBLE COMPLEX for pzunmql.
 Pointer into the local memory to an array of local dimension
 (*lld_c*, *LOCc(jc+n-1)*).
 Contains the local pieces of the distributed matrix sub(*c*)
 to be factored.

ic, jc (global) INTEGER. The row and column indices in the global
 array *c* indicating the first row and the first column of the
 submatrix *c*, respectively.

descc (global and local) INTEGER array, dimension (*dlen_*). The
 array descriptor for the distributed matrix *c*.

work (local)
 COMPLEX for pcunmql
 DOUBLE COMPLEX for pzunmql.
 Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at
 least:
 If *side* = 'L',

$$lwork \geq \max((nb\_a * (nb\_a - 1)) / 2, (nqc0 + mpc0) * nb\_a + nb\_a * nb\_a)$$
 else if *side* = 'R',

$$lwork \geq \max((nb\_a * (nb\_a - 1)) / 2, (nqc0 + \max npa0) + \text{numroc}(\text{numroc}(n + iroffa, nb\_a, 0, 0, NPCOL), nb\_a, 0, 0, lcmq), mpc0)) * nb\_a + nb\_a * nb\_a)$$
 end if
 where

$$lcmp = lcm / NPCOL \text{ with } lcm = ilcm(NPROW, NPCOL),$$

$$iroffa = \text{mod}(ia - 1, mb\_a),$$

$$icoffa = \text{mod}(ja - 1, nb\_a),$$

$$iarow = \text{indxg2p}(ia, mb\_a, MYROW, rsrc\_a, NPROW),$$

$$npa0 = \text{numroc}(n + iroffa, mb\_a, MYROW, iarow, NPROW),$$

$$iroffc = \text{mod}(ic - 1, mb\_c),$$

$$icoffc = \text{mod}(jc - 1, nb\_c),$$

$$icrow = \text{indxg2p}(ic, mb\_c, MYROW, rsrc\_c, NPROW),$$

$$iccol = \text{indxg2p}(jc, nb\_c, MYCOL, csrc\_c, NPCOL),$$

$$mpc0 = \text{numroc}(m + iroffc, mb\_c, MYROW, icrow, NPROW),$$

`nqc0 = numroc(n+icoffc, nb_c, MYCOL, iccol, NPCOL),`
`ilcm, indxcg2p` and `numroc` are ScaLAPACK tool functions;
`MYROW, MYCOL, NPROW` and `NPCOL` can be determined by
calling the subroutine `blacs_gridinfo`.
If `lwork = -1`, then `lwork` is global input and a workspace
query is assumed; the routine only calculates the minimum
and optimal size for all work arrays. Each of these values
is returned in the first entry of the corresponding work array,
and no error message is issued by `pserbla`.

Output Parameters

<code>c</code>	Overwritten by the product $Q^* \text{sub}(c)$, or $Q' \text{sub}(c)$, or $\text{sub}(c)^* Q$, or $\text{sub}(c)^* Q$
<code>work(1)</code>	On exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <code>info = - (i* 100+j)</code> , if the <i>i</i> -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

p?gerqf

Computes the RQ factorization of a general rectangular matrix.

Syntax

```
call psgerqf( m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgerqf( m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgerqf( m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgerqf( m, n, a, ia, ja, desca, tau, work, lwork, info)
```


Description

The routine forms the QR factorization of a general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ as

$$A = R^*Q$$

Input Parameters

m	(global) INTEGER. The number of rows in the distributed submatrix $\text{sub}(A)$; ($m \geq 0$).
n	(global) INTEGER. The number of columns in the distributed submatrix $\text{sub}(A)$; ($n \geq 0$).
a	(local) REAL for <code>psgeqrf</code> DOUBLE PRECISION for <code>pdgeqrf</code> COMPLEX for <code>pcgeqrf</code> DOUBLE COMPLEX for <code>pzgeqrf</code> . Pointer into the local memory to an array of local dimension $(lld\_a, LOCC(ja+n-1))$. Contains the local pieces of the distributed matrix $\text{sub}(A)$ to be factored.
ia, ja	(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.
$desca$	(global and local) INTEGER array, dimension $(dlen\_)$. The array descriptor for the distributed matrix A
$work$	(local). REAL for <code>psgeqrf</code> DOUBLE PRECISION for <code>pdgeqrf</code> . COMPLEX for <code>pcgeqrf</code> . DOUBLE COMPLEX for <code>pzgeqrf</code> Workspace array of dimension $lwork$.
$lwork$	(local or global) INTEGER, dimension of $work$, must be at least $lwork \geq mb\_a * (mp0 + nq0 + mb\_a)$, where $iroff = \text{mod}(ia-1, mb\_a)$, $icoff = \text{mod}(ja-1, nb\_a)$,

`iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),`
`iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),`
`mp0 = numroc(m+iroff, mb_a, MYROW, iarow, NPROW),`
`nq0 = numroc(n+icoff, nb_a, MYCOL, iacol, NPCOL)`
and numroc, indxg2p are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

`a`

On exit, if $m \leq n$, the upper triangle of $A(ia:ia+m-1, ja:ja+n-1)$ contains the m -by- m upper triangular matrix R ; if $m \geq n$, the elements on and above the $(m - n)$ -th subdiagonal contain the m -by- n upper trapezoidal matrix R ; the remaining elements, with the array `tau`, represent the orthogonal/unitary matrix Q as a product of elementary reflectors (see *Application Notes* below).

`tau`

(local)
 REAL for `psgeqrf`
 DOUBLE PRECISION for `pdgeqrf`
 COMPLEX for `pcgeqrf`
 DOUBLE COMPLEX for `pzgeqrf`.
 Array, DIMENSION `LOCr(ia+m-1)`.
 Contains the scalar factor `tau` of elementary reflectors. `tau` is tied to the distributed matrix A .

`work(1)`

On exit, `work(1)` contains the minimum value of `lwork` required for optimum performance.

`info`

(global) INTEGER.
 = 0, the execution is successful.

< 0 , if the i -th argument is an array and the j -entry had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(ia) * H(ia+1) * \dots * H(ia+k-1),$$

where $k = \min(m, n)$.

Each $H(i)$ has the form

$$H(i) = I - \tau v v'$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(n-k+i+1:n) = 0$ and $v(n-k+i) = 1$; $v(1:n-k+i-1)/\text{conjg}(v(1:n-k+i-1))$ is stored on exit in $A(ia+m-k+i-1, ja:ja+n-k+i-2)$, and τ in $\tau(ia+m-k+i-1)$.

p?orgrq

Generates the orthogonal matrix Q of the RQ factorization formed by p?gerqf.

Syntax

```
call psorgrq( m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorgrq( m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine generates the whole or part of m -by- n real distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal columns, which is defined as the last m rows of a product of k elementary reflectors of order m

$$Q = H(1) * H(2) * \dots * H(k)$$

as returned by p?gerqf.

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows in the submatrix <code>sub(Q)</code> ; ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the submatrix <code>sub(Q)</code> ($n \geq m \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix <i>Q</i> ($m \geq k \geq 0$).
<i>a</i>	(local) REAL for <code>psorgqr</code> DOUBLE PRECISION for <code>pdorgqr</code> Pointer into the local memory to an array of local dimension (<code>lld_a</code> , <code>LOCc(ja+n-1)</code>). The <i>i</i> -th column must contain the vector which defines the elementary reflector $H(i)$, $ja \leq j \leq ja+k-1$, as returned by <code>p?gerqf</code> in the <i>k</i> columns of its distributed matrix argument $A(ia:*, ja:ja+k-1)$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<code>dlen_</code>). The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) REAL for <code>psorgqr</code> DOUBLE PRECISION for <code>pdorgqr</code> Array, DIMENSION <code>LOCc(ja+k-1)</code> . Contains the scalar factor $\tau(i)$ of elementary reflectors $H(i)$ as returned by <code>p?gerqf</code> . <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>work</i>	(local) REAL for <code>psorgqr</code> DOUBLE PRECISION for <code>pdorgqr</code> Workspace array of dimension of <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, dimension of <i>work</i> , must be at least $lwork \geq mb\_a * (mpa0 + nqa0 + mb\_a)$, where $iroffa = \text{mod}(ia-1, mb\_a)$,

```

icoffa = mod(ja-1, nb_a),
iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
mpa0 = numroc(m+iroffa, mb_a, MYROW, iarow,
NPROW),
nqa0 = numroc(n+icoffa, nb_a, MYCOL, iacol,
NPCOL)
indxg2p and numroc are ScaLAPACK tool functions; MYROW,
MYCOL, NPROW and NPCOL can be determined by calling the
subroutine blacs_gridinfo.

```

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

<i>a</i>	Contains the local pieces of the <i>m</i> -by- <i>n</i> distributed matrix <i>Q</i> .
<i>work</i> (1)	On exit, <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = -(<i>i</i> *100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

p?ungrq

Generates the unitary matrix *Q* of the *RQ* factorization formed by [p?gerqf](#).

Syntax

```

call pcungrq( m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzungrq( m, n, k, a, ia, ja, desca, tau, work, lwork, info)

```

Description

The routine generates the m -by- n complex distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the last m rows of a product of k elementary reflectors of order n

$$Q = (H(1))^H \star (H(2))^H \star \dots \star (H(k))^H \text{ as returned by } \text{p?gerqf}.$$

Input Parameters

m	(global) INTEGER. The number of rows in the submatrix $\text{sub}(Q)$; ($m \geq 0$).
n	(global) INTEGER. The number of columns in the submatrix $\text{sub}(Q)$ ($n \geq m \geq 0$).
k	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q ($m \geq k \geq 0$).
a	(local) COMPLEX for pcungrq DOUBLE COMPLEX for pzungrqc Pointer into the local memory to an array of dimension $(lld\_a, LOCC(ja+n-1))$. The i -th row must contain the vector which defines the elementary reflector $H(i)$, $ia+m-k \leq i \leq ia+m-1$, as returned by p?gerqf in the k rows of its distributed matrix argument $A(ia+m-k:ia+m-1, ja:*)$.
ia, ja	(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix A , respectively.
$desca$	(global and local) INTEGER array, dimension $(dlen\_)$. The array descriptor for the distributed matrix A .
τ	(local) COMPLEX for pcungrq DOUBLE COMPLEX for pzungrqc Array, DIMENSION $LOCr(ia+m-1)$. Contains the scalar factor $\tau(i)$ of elementary reflectors $H(i)$ as returned by p?gerqf . τ is tied to the distributed matrix A .

work (local)
 COMPLEX for pcungrq
 DOUBLE COMPLEX for pzungrq
 Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least $lwork \geq mb\_a*(mpa0 + nqa0 + mb\_a)$, where

```

iroffa = mod(ia-1, mb_a),
icoffa = mod(ja-1, nb_a),
iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
mpa0 = numroc(m+iroffa, mb_a, MYROW, iarow, NPROW),
nqa0 = numroc(n+icoffa, nb_a, MYCOL, iacol, NPCOL)

```

indxg2p and *numroc* are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine *blacs_gridinfo*.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by *pxerbla*.

Output Parameters

a Contains the local pieces of the m -by- n distributed matrix Q .

work(1) On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info (global) INTEGER.
 = 0: the execution is successful.
 < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = $-(i*100+j)$, if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?ormrq

Multiplies a general matrix by the orthogonal matrix Q of the RQ factorization formed by p?gerqf.

Syntax

```
call psormrq( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

```
call pdormrq( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

Description

The routine overwrites the general real m -by- n distributed matrix $\text{sub}(C)$ = $C(ic:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
$trans = 'T':$	$Q^T * \text{sub}(C)$	$\text{sub}(C) * Q^T$

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors

$$Q = H(1) H(2) \dots H(k)$$

as returned by p?gerqf. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

$side$	(global) CHARACTER = 'L': Q or Q^T is applied from the left. = 'R': Q or Q^T is applied from the right.
$trans$	(global) CHARACTER = 'N', no transpose, Q is applied. = 'T', transpose, Q^T is applied.
m	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(C)$ ($m \geq 0$).
n	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(C)$ ($n \geq 0$).

<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q. Constraints: If <i>side</i> = 'L', $m \geq k \geq 0$ If <i>side</i> = 'R', $n \geq k \geq 0$.
<i>a</i>	(local) REAL for psormqr DOUBLE PRECISION for pdormqr. Pointer into the local memory to an array of dimension (<i>lld_a</i>, <i>LOCc</i>(<i>ja</i>+<i>m</i>-1)) if <i>side</i> = 'L', and (<i>lld_a</i>, <i>LOCc</i>(<i>ja</i>+<i>n</i>-1)) if <i>side</i> = 'R'. The <i>i</i>-th row must contain the vector which defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by p?gerqf in the <i>k</i> rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$. $A(ia:ia+k-1, ja:*)$ is modified by the routine but restored on exit.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i>, respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i>.
<i>tau</i>	(local) REAL for psormqr DOUBLE PRECISION for pdormqr Array, DIMENSION <i>LOCc</i>(<i>ja</i>+<i>k</i>-1)). Contains the scalar factor <i>tau</i>(<i>i</i>) of elementary reflectors $H(i)$ as returned by p?gerqf. <i>tau</i> is tied to the distributed matrix <i>A</i>.
<i>c</i>	(local) REAL for psormrq DOUBLE PRECISION for pdormrq Pointer into the local memory to an array of local dimension (<i>lld_c</i>, <i>LOCc</i>(<i>jc</i>+<i>n</i>-1)). Contains the local pieces of the distributed matrix sub(<i>c</i>) to be factored.

<i>ic, jc</i>	(global) INTEGER. The row and column indices in the global array <i>c</i> indicating the first row and the first column of the submatrix <i>c</i> , respectively.
<i>descc</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>c</i> .
<i>work</i>	(local) REAL for psormrq DOUBLE PRECISION for pdormrq. Workspace array of dimension of <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, dimension of <i>work</i> , must be at least: If <i>side</i> = 'L', $lwork \geq \max((mb\_a * (mb\_a - 1)) / 2, (mpc0 + \max(mqa0 + \text{numroc}(\text{numroc}(n + iroffc, mb\_a, 0, 0, NPROW), mb\_a, 0, 0, lcmp), nqc0)) * mb\_a) + mb\_a * mb\_a)$ else if <i>side</i> = 'R', $lwork \geq \max((mb\_a * (mb\_a - 1)) / 2, (mpc0 + nqc0) * mb\_a) + mb\_a * mb\_a$ end if where <i>lcmp</i> = <i>lcm</i> / <i>NPROW</i> with <i>lcm</i> = <i>ilcm</i> (<i>NPROW</i> , <i>NPCOL</i>), <i>iroffa</i> = <i>mod</i> (<i>ia</i> - 1, <i>mb_a</i>), <i>icoffa</i> = <i>mod</i> (<i>ja</i> - 1, <i>nb_a</i>), <i>iacol</i> = <i>indxg2p</i> (<i>ja</i> , <i>nb_a</i> , <i>MYCOL</i> , <i>csrc_a</i> , <i>NPCOL</i>), <i>mqa0</i> = <i>numroc</i> (<i>n</i> + <i>icoffa</i> , <i>nb_a</i> , <i>MYCOL</i> , <i>iacol</i> , <i>NPCOL</i>), <i>iroffc</i> = <i>mod</i> (<i>ic</i> - 1, <i>mb_c</i>), <i>icoffc</i> = <i>mod</i> (<i>jc</i> - 1, <i>nb_c</i>), <i>icrow</i> = <i>indxg2p</i> (<i>ic</i> , <i>mb_c</i> , <i>MYROW</i> , <i>rsrc_c</i> , <i>NPROW</i>), <i>iccol</i> = <i>indxg2p</i> (<i>jc</i> , <i>nb_c</i> , <i>MYCOL</i> , <i>csrc_c</i> , <i>NPCOL</i>), <i>mpc0</i> = <i>numroc</i> (<i>m</i> + <i>iroffc</i> , <i>mb_c</i> , <i>MYROW</i> , <i>icrow</i> , <i>NPROW</i>), <i>nqc0</i> = <i>numroc</i> (<i>n</i> + <i>icoffc</i> , <i>nb_c</i> , <i>MYCOL</i> , <i>iccol</i> , <i>NPCOL</i>), <i>ilcm</i> , <i>indxg2p</i> and <i>numroc</i> are ScaLAPACK tool functions; <i>MYROW</i> , <i>MYCOL</i> , <i>NPROW</i> and <i>NPCOL</i> can be determined by calling the subroutine <i>blacs_gridinfo</i> .

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

c	Overwritten by the product $Q^* \text{sub}(c)$, or $Q'^* \text{sub}(c)$, or $\text{sub}(c)^* Q'$, or $\text{sub}(c)^* Q$
$work(1)$	On exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance.
$info$	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -entry had an illegal value, then $info = -(i * 100 + j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

p?unmrq

Multiplies a general matrix by the unitary matrix Q of the RQ factorization formed by p?gerqf.

Syntax

```
call pcunmrq( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

```
call pzunmrq( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

Description

The routine overwrites the general complex m -by- n distributed matrix $\text{sub}(c) = C(ic:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q^* \text{sub}(c)$	$\text{sub}(c)^* Q$
$trans = 'C':$	$Q^H \text{sub}(c)$	$\text{sub}(c)^* Q^H$

where Q is a complex unitary distributed matrix defined as the product of k elementary reflectors

$$Q = H(1)' H(2)' \dots H(k)'$$

as returned by `p?gerqf`. Q is of order m if `side = 'L'` and of order n if `side = 'R'`.

Input Parameters

<code>side</code>	(global) CHARACTER = 'L': Q or Q^H is applied from the left. = 'R': Q or Q^H is applied from the right.
<code>trans</code>	(global) CHARACTER = 'N', no transpose, Q is applied. = 'C', conjugate transpose, Q^H is applied.
<code>m</code>	(global) INTEGER. The number of rows in the distributed matrix sub(C) ($m \geq 0$).
<code>n</code>	(global) INTEGER. The number of columns in the distributed matrix sub(C) ($n \geq 0$).
<code>k</code>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: If <code>side = 'L'</code> , $m \geq k \geq 0$ If <code>side = 'R'</code> , $n \geq k \geq 0$.
<code>a</code>	(local) COMPLEX for <code>pcunmrq</code> DOUBLE COMPLEX for <code>pzunmrq</code> . Pointer into the local memory to an array of dimension $(lld\_a, LOCC(ja+m-1))$ if <code>side = 'L'</code> , and $(lld\_a, LOCC(ja+n-1))$ if <code>side = 'R'</code> . The i -th row must contain the vector which defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by <code>p?gerqf</code> in the k rows of its distributed matrix argument $A(ia:ia+k-1, ja^*)$. $A(ia:ia+k-1, ja^*)$ is modified by the routine but restored on exit.
<code>ia, ja</code>	(global) INTEGER. The row and column indices in the global array <code>a</code> indicating the first row and the first column of the submatrix A , respectively.

<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) COMPLEX for pcunmrq DOUBLE COMPLEX for pzunmrq Array, DIMENSION <i>LOCc(ja+k-1)</i> . Contains the scalar factor <i>tau(i)</i> of elementary reflectors <i>H(i)</i> as returned by p?gerqf. <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>c</i>	(local) COMPLEX for pcunmrq DOUBLE COMPLEX for pzunmrq. Pointer into the local memory to an array of local dimension (<i>lld_c</i> , <i>LOCc(jc+n-1)</i>). Contains the local pieces of the distributed matrix sub(<i>c</i>) to be factored.
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the global array <i>c</i> indicating the first row and the first column of the submatrix <i>C</i> , respectively.
<i>descc</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>C</i> .
<i>work</i>	(local) COMPLEX for pcunmrq DOUBLE COMPLEX for pzunmrq. Workspace array of dimension of <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, dimension of <i>work</i> , must be at least: If <i>side</i> = 'L', $lwork \geq \max((mb\_a*(mb\_a-1))/2, (mpc0 + \max(mqa0+numroc(numroc(n+iroffc, mb\_a, 0, 0, NPROW), mb\_a, 0, 0, lcmp), nqc0))*mb\_a) + mb\_a*mb\_a)$ else if <i>side</i> = 'R', $lwork \geq \max((mb\_a*(mb\_a-1))/2, (mpc0 + nqc0)*mb\_a) + mb\_a*mb\_a$ end if

where

```
lcmp = lcm/NPROW with lcm = ilcm(NPROW, NPCOL),
iroffa = mod(ia-1, mb_a),
icoffa = mod(ja-1, nb_a),
iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
mqa0 = numroc(m+icoffa, nb_a, MYCOL, iacol,
NPCOL),
iroffc = mod(ic-1, mb_c),
icoffc = mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(m+iroffc, mb_c, MYROW, icrow,
NPROW),
nqc0 = numroc(n+icoffc, nb_c, MYCOL, iccol,
NPCOL),
```

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>c</code>	Overwritten by the product $Q^* \text{sub}(c)$ or $Q'^* \text{sub}(c)$, or $\text{sub}(c)^* Q'$, or $\text{sub}(c)^* Q$
<code>work(1)</code>	On exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <code>info = - (i* 100+j)</code> , if the <i>i</i> -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

p?tzzrf

Reduces the upper trapezoidal matrix A to upper triangular form.

Syntax

```
call pstzzrf( m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdtzzrf( m, n, a, ia, ja, desca, tau, work, lwork, info)
call pctzzrf( m, n, a, ia, ja, desca, tau, work, lwork, info)
call pztzzrf( m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Description

This routine reduces the m -by- n ($m \leq n$) real/complex upper trapezoidal matrix $\text{sub}(A) = (ia:ia+m-1, ja:ja+n-1)$ to upper triangular form by means of orthogonal/unitary transformations. The upper trapezoidal matrix A is factored as

$$A = \begin{pmatrix} R & 0 \end{pmatrix} * Z,$$

where Z is an n -by- n orthogonal/unitary matrix and R is an m -by- m upper triangular matrix.

Input Parameters

m	(global) INTEGER. The number of rows in the submatrix $\text{sub}(A)$; ($m \geq 0$).
n	(global) INTEGER. The number of columns in the submatrix $\text{sub}(A)$ ($n \geq 0$).
a	(local) REAL for pstzzrf DOUBLE PRECISION for pdtzzrf. COMPLEX for pctzzrf. DOUBLE COMPLEX for pztzzrf. Pointer into the local memory to an array of dimension $(lld\_a, LOCC(ja+n-1))$. Contains the local pieces of the m -by- n distributed matrix $\text{sub}(A)$ to be factored.
ia, ja	(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix A , respectively.

<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>work</i>	(local) REAL for <i>pstzrzf</i> DOUBLE PRECISION for <i>pdtzrzf</i> . COMPLEX for <i>pctzrzf</i> . DOUBLE COMPLEX for <i>pztzrzf</i> . Workspace array of dimension of <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, dimension of <i>work</i> , must be at least $lwork \geq mb\_a * (mp0 + nq0 + mb\_a)$, where $iroff = \text{mod}(ia-1, mb\_a)$, $icoff = \text{mod}(ja-1, nb\_a)$, $iarow = \text{indxg2p}(ia, mb\_a, MYROW, rsrc\_a, NPROW)$, $iacol = \text{indxg2p}(ja, nb\_a, MYCOL, csrc\_a, NPCOL)$, $mp0 = \text{numroc}(m + iroff, mb\_a, MYROW, iarow, NPROW)$, $nq0 = \text{numroc}(n + icoff, nb\_a, MYCOL, iacol, NPCOL)$ <i>indxg2p</i> and <i>numroc</i> are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine <i>blacs_gridinfo</i> . If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by <i>pxerbla</i> .

Output Parameters

<i>a</i>	On exit, the leading <i>m</i> -by- <i>m</i> upper triangular part of sub(<i>A</i>) contains the upper triangular matrix <i>R</i> , and elements <i>m</i> +1 to <i>n</i> of the first <i>m</i> rows of sub (<i>A</i>), with the array <i>tau</i> , represent the orthogonal/unitary matrix <i>Z</i> as a product of <i>m</i> elementary reflectors.
<i>work</i> (1)	On exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>tau</i>	(local)

REAL for pstzrzf
 DOUBLE PRECISION for pdtzrzf.
 COMPLEX for pctzrzf.
 DOUBLE COMPLEX for pztzrzf.
 Array, DIMENSION $LOCr(ia+m-1)$.
 Contains the scalar factor of elementary reflectors. τ is tied to the distributed matrix A .
info (global) INTEGER.
 = 0: the execution is successful.
 < 0: if the i -th argument is an array and the j -entry had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

The factorization is obtained by the Householder's method. The k -th transformation matrix, $Z(k)$, which is or whose conjugate transpose is used to introduce zeros into the $(m - k + 1)$ -th row of $\text{sub}(A)$, is given in the form

$$Z(k) = \begin{bmatrix} i & 0 \\ 0 & T(k) \end{bmatrix}$$

where

$$T(k) = i - \tau u(k) u(k)^*,$$

$$u(k) = \begin{bmatrix} 1 \\ 0 \\ z(k) \end{bmatrix}$$

τ is a scalar and $Z(k)$ is an $(n - m)$ element vector. τ and $Z(k)$ are chosen to annihilate the elements of the k -th row of $\text{sub}(A)$. The scalar τ is returned in the k -th element of τ and the vector $u(k)$ in the k -th row of $\text{sub}(A)$, such that the elements of $Z(k)$ are in $a(k, m + 1), \dots, a(k, n)$. The elements of R are returned in the upper triangular part of $\text{sub}(A)$. Z is given by

$$Z = Z(1) * Z(2) * \dots * Z(m).$$

p?ormrz

Multiplies a general matrix by the orthogonal matrix from a reduction to upper triangular form formed by p?tzzrf.

Syntax

```
call psormrz( side, trans, m, n, k, l, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)

call pdormrz( side, trans, m, n, k, l, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

Description

The routine overwrites the general real m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
$trans = 'T':$	$Q^T * \text{sub}(C)$	$\text{sub}(C) * Q^T$

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors

$$Q = H(1) H(2) \dots H(k)$$

as returned by p?tzzrf. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

$side$	(global) CHARACTER = 'L': Q or Q^T is applied from the left. = 'R': Q or Q^T is applied from the right.
$trans$	(global) CHARACTER = 'N', no transpose, Q is applied. = 'T', transpose, Q^T is applied.
m	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(C)$ ($m \geq 0$).

<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(C)$ ($n \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: If <i>side</i> = 'L', $m \geq k \geq 0$ If <i>side</i> = 'R', $n \geq k \geq 0$.
<i>l</i>	(global) The columns of the distributed submatrix $\text{sub}(A)$ containing the meaningful part of the Householder reflectors. If <i>side</i> = 'L', $m \geq l \geq 0$ If <i>side</i> = 'R', $n \geq l \geq 0$.
<i>a</i>	(local) REAL for psormrz DOUBLE PRECISION for pdormrz. Pointer into the local memory to an array of dimension (<i>lld_a</i> , <i>LOCc</i> (<i>ja</i> + <i>m</i> -1)) if <i>side</i> = 'L', and (<i>lld_a</i> , <i>LOCc</i> (<i>ja</i> + <i>n</i> -1)) if <i>side</i> = 'R', where <i>lld_a</i> $\geq \max(1, \text{LOCr}(\text{ia}+k-1))$. The <i>i</i> -th row must contain the vector which defines the elementary reflector $H(i)$, $\text{ia} \leq i \leq \text{ia}+k-1$, as returned by p?tzrpf in the <i>k</i> rows of its distributed matrix argument $A(\text{ia}:\text{ia}+k-1, \text{ja}:*)$. $A(\text{ia}:\text{ia}+k-1, \text{ja}:*)$ is modified by the routine but restored on exit.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) REAL for psormrz DOUBLE PRECISION for pdormrz Array, DIMENSION <i>LOCc</i> (<i>ia</i> + <i>k</i> -1). Contains the scalar factor <i>tau</i> (<i>i</i>) of elementary reflectors $H(i)$ as returned by p?tzrpf. <i>tau</i> is tied to the distributed matrix <i>A</i> .

<i>c</i>	(local) REAL for psormrz DOUBLE PRECISION for pdormrz Pointer into the local memory to an array of local dimension (<i>lld_c</i>, <i>LOCc(jc+n-1)</i>). Contains the local pieces of the distributed matrix sub(<i>c</i>) to be factored.
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the global array <i>c</i> indicating the first row and the first column of the submatrix <i>c</i>, respectively.
<i>desc</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>c</i>.
<i>work</i>	(local) REAL for psormrz DOUBLE PRECISION for pdormrz. Workspace array of dimension of <i>lwork</i>.
<i>lwork</i>	(local or global) INTEGER, dimension of <i>work</i>, must be at least: If <i>side</i> = 'L', $lwork \geq \max((mb\_a * (mb\_a - 1)) / 2, (mpc0 + \max(mqa0 + \text{numroc}(\text{numroc}(n + iroffa, mb\_a, 0, 0, NPROW), mb\_a, 0, 0, lcmp), nqc0)) * mb\_a) + mb\_a * mb\_a)$ else if <i>side</i> = 'R', $lwork \geq \max((mb\_a * (mb\_a - 1)) / 2, (mpc0 + nqc0) * mb\_a) + mb\_a * mb\_a$ end if where <i>lcmp</i> = <i>lcm</i> / NPROW with <i>lcm</i> = <i>ilcm</i> (NPROW, NPCOL), <i>iroffa</i> = mod(<i>ia</i>-1, <i>mb_a</i>), <i>icoffa</i> = mod(<i>ja</i>-1, <i>nb_a</i>), <i>iacol</i> = indxg2p(<i>ja</i>, <i>nb_a</i>, MYCOL, <i>csrc_a</i>, NPCOL), <i>mqa0</i> = numroc(<i>n</i>+<i>icoffa</i>, <i>nb_a</i>, MYCOL, <i>iacol</i>, NPCOL), <i>iroffc</i> = mod(<i>ic</i>-1, <i>mb_c</i>), <i>icoffc</i> = mod(<i>jc</i>-1, <i>nb_c</i>), <i>icrow</i> = indxg2p(<i>ic</i>, <i>mb_c</i>, MYROW, <i>rsrc_c</i>, NPROW), <i>iccol</i> = indxg2p(<i>jc</i>, <i>nb_c</i>, MYCOL, <i>csrc_c</i>, NPCOL),

```

mpc0 = numroc(m+iroffc, mb_c, MYROW, icrow,
NPROW),
nqc0 = numroc(n+icoffc, nb_c, MYCOL, iccol,
NPCOL),
ilcm, indxg2p and numroc are ScaLAPACK tool functions;
MYROW, MYCOL, NPROW and NPCOL can be determined by
calling the subroutine blacs_gridinfo.
If lwork = -1, then lwork is global input and a workspace
query is assumed; the routine only calculates the minimum
and optimal size for all work arrays. Each of these values
is returned in the first entry of the corresponding work array,
and no error message is issued by p?xerbla.

```

Output Parameters

<i>c</i>	Overwritten by the product $Q*\text{sub}(c)$, or $Q'*\text{sub}(c)$, or $\text{sub}(c)*Q'$, or $\text{sub}(c)*Q$
<i>work(1)</i>	On exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = - (<i>i</i> * 100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

p?unmrz

Multiplies a general matrix by the unitary transformation matrix from a reduction to upper triangular form determined by p?tzrzf.

Syntax

```

call pcunmrz( side, trans, m, n, k, l, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)

call pzunmrz( side, trans, m, n, k, l, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)

```

Description

The routine overwrites the general complex m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
$trans = 'C':$	$Q^H * \text{sub}(C)$	$\text{sub}(C) * Q^H$

where Q is a complex unitary distributed matrix defined as the product of k elementary reflectors

$$Q = H(1)' H(2)' \dots H(k)'$$

as returned by `pctzrzf/pztzrzf`. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

$side$	(global) CHARACTER $= 'L'$: Q or Q^H is applied from the left. $= 'R'$: Q or Q^H is applied from the right.
$trans$	(global) CHARACTER $= 'N'$, no transpose, Q is applied. $= 'C'$, conjugate transpose, Q^H is applied.
m	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(C)$, ($m \geq 0$).
n	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(C)$, ($n \geq 0$).
k	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: If $side = 'L'$, $m \geq k \geq 0$ If $side = 'R'$, $n \geq k \geq 0$.
l	(global) INTEGER. The columns of the distributed submatrix $\text{sub}(A)$ containing the meaningful part of the Householder reflectors. If $side = 'L'$, $m \geq l \geq 0$ If $side = 'R'$, $n \geq l \geq 0$.
a	(local)

	COMPLEX for pcunmrz DOUBLE COMPLEX for pzunmrz. Pointer into the local memory to an array of dimension $(lld\_a, LOCc(ja+m-1))$ if $side = 'L'$, and $(lld\_a, LOCc(ja+n-1))$ if $side = 'R'$, where $lld\_a \geq \max(1, LOCr(ja+k-1))$. The i-th row must contain the vector which defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by p?gerqf in the k rows of its distributed matrix argument $A(ia:ia+k-1, ja^*)$. $A(ia:ia+k-1, ja^*)$ is modified by the routine but restored on exit.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension $(dlen\_)$. The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) COMPLEX for pcunmrz DOUBLE COMPLEX for pzunmrz Array, DIMENSION $LOCc(ia+k-1)$. Contains the scalar factor $\tau(i)$ of elementary reflectors $H(i)$ as returned by p?gerqf. τ is tied to the distributed matrix <i>A</i>.
<i>c</i>	(local) COMPLEX for pcunmrz DOUBLE COMPLEX for pzunmrz. Pointer into the local memory to an array of local dimension $(lld\_c, LOCc(jc+n-1))$. Contains the local pieces of the distributed matrix sub(<i>c</i>) to be factored.
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the global array <i>c</i> indicating the first row and the first column of the submatrix <i>C</i> , respectively.
<i>descc</i>	(global and local) INTEGER array, dimension $(dlen\_)$. The array descriptor for the distributed matrix <i>C</i> .
<i>work</i>	(local) COMPLEX for pcunmrz

lwork

DOUBLE COMPLEX for pzunmrz.
 Workspace array of dimension *lwork*.
 (local or global) INTEGER, dimension of *work*, must be at least:
 If *side* = 'L',

$$lwork \geq \max((mb\_a * (mb\_a - 1)) / 2, (mpc0 + \max(mqa0 + \text{numroc}(\text{numroc}(n + iroffa, mb\_a, 0, 0, NPROW), mb\_a, 0, 0, lcmp), nqc0)) * mb\_a) + mb\_a * mb\_a)$$

 else if *side* = 'R',

$$lwork \geq \max((mb\_a * (mb\_a - 1)) / 2, (mpc0 + nqc0) * mb\_a + mb\_a * mb\_a)$$

 end if
 where

$$lcmp = lcm / NPROW \text{ with } lcm = ilcm(NPROW, NPCOL),$$

$$iroffa = \text{mod}(ia - 1, mb\_a),$$

$$icoffa = \text{mod}(ja - 1, nb\_a),$$

$$iacol = \text{indxg2p}(ja, nb\_a, MYCOL, csrc\_a, NPCOL),$$

$$mqa0 = \text{numroc}(m + icoffa, nb\_a, MYCOL, iacol, NPCOL),$$

$$iroffc = \text{mod}(ic - 1, mb\_c),$$

$$icoffc = \text{mod}(jc - 1, nb\_c),$$

$$icrow = \text{indxg2p}(ic, mb\_c, MYROW, rsrc\_c, NPROW),$$

$$iccol = \text{indxg2p}(jc, nb\_c, MYCOL, csrc\_c, NPCOL),$$

$$mpc0 = \text{numroc}(m + iroffc, mb\_c, MYROW, icrow, NPROW),$$

$$nqc0 = \text{numroc}(n + icoffc, nb\_c, MYCOL, iccol, NPCOL),$$

$$ilcm, \text{indxg2p} \text{ and } \text{numroc} \text{ are ScaLAPACK tool functions;}$$

$$MYROW, MYCOL, NPROW \text{ and } NPCOL \text{ can be determined by}$$

 calling the subroutine `blacs_gridinfo`.
 If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>c</code>	Overwritten by the product $Q^* \text{sub}(c)$, or $Q'^* \text{sub}(c)$, or $\text{sub}(c) * Q'$, or $\text{sub}(c) * Q$
<code>work(1)</code>	On exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <code>info</code> = - (<i>i</i> *100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <code>info</code> = - <i>i</i> .

p?ggqrf

Computes the generalized QR factorization.

Syntax

```
call psggqrf(n, m, p, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work,
lwork, info)

call pdggqrf(n, m, p, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work,
lwork, info)

call pcggqrf(n, m, p, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work,
lwork, info)

call pzggqrf(n, m, p, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work,
lwork, info)
```

Description

The routine forms the generalized QR factorization of an n -by- m matrix

$\text{sub}(A) = A(ia:ia+n-1, ja:ja+m-1)$

and an n -by- p matrix

$\text{sub}(B) = B(ib:ib+n-1, jb:jb+p-1):$

as

$\text{sub}(A) = Q^* R, \text{sub}(B) = Q^* T^* Z,$

where Q is an n -by- n orthogonal/unitary matrix, Z is a p -by- p orthogonal/unitary matrix, and R and T assume one of the forms:

If $n \geq m$

$$R = \begin{pmatrix} R_{11} & \\ & 0 \end{pmatrix} \begin{matrix} m \\ n - m \\ m \end{matrix}$$

or if $n < m$

$$R = \begin{pmatrix} R_{11} & R_{12} \\ & \end{pmatrix} \begin{matrix} n \\ n \\ m - n \end{matrix}$$

where R_{11} is upper triangular, and

$$T = \begin{pmatrix} 0 & T_{12} \\ & \end{pmatrix} \begin{matrix} n, \text{ if } n \leq p, \\ p - n & n \end{matrix}$$

$$T = \begin{pmatrix} 0 & T_{12} \\ & \end{pmatrix} \begin{matrix} n, \text{ if } n \leq p, \\ p - n & n \end{matrix}$$

$$\text{or } T = \begin{pmatrix} T_{11} \\ T_{21} \end{pmatrix} \begin{pmatrix} n - p \\ p \end{pmatrix}, \text{ if } n > p,$$

p

where T_{12} or T_{21} is an upper triangular matrix.

In particular, if $\text{sub}(B)$ is square and nonsingular, the GQR factorization of $\text{sub}(A)$ and $\text{sub}(B)$ implicitly gives the QR factorization of $\text{inv}(\text{sub}(B)) * \text{sub}(A)$:

$$\text{inv}(\text{sub}(B)) * \text{sub}(A) = Z^H * (\text{inv}(T) * R$$

Input Parameters

n	(global) INTEGER. The number of rows in the distributed matrices $\text{sub}(A)$ and $\text{sub}(B)$ ($n \geq 0$).
m	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(A)$ ($m \geq 0$).
p	INTEGER. The number of columns in the distributed matrix $\text{sub}(B)$ ($p \geq 0$).
a	(local) REAL for psggqrf DOUBLE PRECISION for pdggqrf COMPLEX for pcggqrf DOUBLE COMPLEX for pzggqrf. Pointer into the local memory to an array of dimension $(lld\_a, LOCC(ja+m-1))$. Contains the local pieces of the n -by- m matrix $\text{sub}(A)$ to be factored.
ia, ja	(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix A , respectively.
$desca$	(global and local) INTEGER array, dimension $(dlen\_)$. The array descriptor for the distributed matrix A .
b	(local) REAL for psggqrf DOUBLE PRECISION for pdggqrf COMPLEX for pcggqrf DOUBLE COMPLEX for pzggqrf. Pointer into the local memory to an array of dimension $(lld\_b, LOCC(jb+p-1))$. Contains the local pieces of the n -by- p matrix $\text{sub}(B)$ to be factored.
ib, jb	(global) INTEGER. The row and column indices in the global array b indicating the first row and the first column of the submatrix B , respectively.

descb (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix B.

work (local)
 REAL for psggqrf
 DOUBLE PRECISION for pdggqrf
 COMPLEX for pcggqrf
 DOUBLE COMPLEX for pzggqrf.
 Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER. Dimension of *work*, must be at least

$$lwork \geq \max(nb\_a * (npa0 + mqa0 + nb\_a), \\ \max((nb\_a * (nb\_a - 1)) / 2, \\ (pqb0 + npb0) * nb\_a + nb\_a * nb\_a, \\ mb\_b * (npb0 + pqb0 + mb\_b)),$$

where

$$iroffa = \text{mod}(ia - 1, mb\_a), \\ icoffa = \text{mod}(ja - 1, nb\_a), \\ iarow = \text{indxg2p}(ia, mb\_a, MYROW, rsrc\_a, NPROW), \\ iacol = \text{indxg2p}(ja, nb\_a, MYCOL, csrc\_a, NPCOL), \\ npa0 = \text{numroc}(n + iroffa, mb\_a, MYROW, iarow, NPROW), \\ mqa0 = \text{numroc}(m + icoffa, nb\_a, MYCOL, iacol, NPCOL) \\ iroffb = \text{mod}(ib - 1, mb\_b), \\ icoffb = \text{mod}(jb - 1, nb\_b), \\ ibrow = \text{indxg2p}(ib, mb\_b, MYROW, rsrc\_b, NPROW), \\ ibcol = \text{indxg2p}(jb, nb\_b, MYCOL, csrc\_b, NPCOL), \\ npb0 = \text{numroc}(n + iroffb, mb\_b, MYROW, ibrow, NPROW), \\ pqb0 = \text{numroc}(m + icoffb, nb\_b, MYCOL, ibcol, NPCOL)$$

and numroc, indxg2p are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<i>a</i>	On exit, the elements on and above the diagonal of sub (<i>A</i>) contain the $\min(n, m)$ -by- m upper trapezoidal matrix R (R is upper triangular if $n \geq m$); the elements below the diagonal, with the array <i>taua</i> , represent the orthogonal/unitary matrix Q as a product of $\min(n, m)$ elementary reflectors. (See Application Notes below).
<i>taua, taub</i>	(local) REAL for psggqrf DOUBLE PRECISION for pdggqrf COMPLEX for pcggqrf DOUBLE COMPLEX for pzggqrf. Arrays, DIMENSION $LOCc(ja+\min(n, m)-1)$ for <i>taua</i> and $LOCr(ib+n-1)$ for <i>taub</i>. The array <i>taua</i> contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Q. <i>taua</i> is tied to the distributed matrix <i>A</i>. (See Application Notes below). The array <i>taub</i> contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Z. <i>taub</i> is tied to the distributed matrix <i>B</i>. (See Application Notes below).
<i>work</i> (1)	On exit <i>work</i> (1) contains the minimum value of $lwork$ required for optimum performance.
<i>info</i>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i>-th argument is an array and the <i>j</i>-entry had an illegal value, then $info = -(i*100+j)$, if the <i>i</i>-th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(ja) * H(ja+1) * \dots * H(ja+k-1),$$

where $k = \min(n, m)$.

Each $H(i)$ has the form

$$H(i) = I - \tau_{aua} * v * v'$$

where τ_{aua} is a real/complex scalar, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:n)$ is stored on exit in $A(ia+i:ia+n-1, ja+i-1)$, and τ_{aua} in $\tau_{aua}(ja+i-1)$. To form Q explicitly, use ScaLAPACK subroutine [p?orgqr/p?ungqr](#). To use Q to update another matrix, use ScaLAPACK subroutine [p?ormqr/p?unmqr](#).

The matrix Z is represented as a product of elementary reflectors

$$Z = H(ib) * H(ib+1) * \dots * H(ib+k-1), \text{ where } k = \min(n, p).$$

Each $H(i)$ has the form

$$H(i) = I - \tau_{aub} * v * v'$$

where τ_{aub} is a real/complex scalar, and v is a real/complex vector with $v(p-k+i+1:p) = 0$ and $v(p-k+i) = 1$; $v(1:p-k+i-1)$ is stored on exit in $B(ib+n-k+i-1, jb:jb+p-k+i-2)$, and τ_{aub} in $\tau_{aub}(ib+n-k+i-1)$. To form Z explicitly, use ScaLAPACK subroutine [p?orgqr/p?ungqr](#). To use Z to update another matrix, use ScaLAPACK subroutine [p?ormqr/p?unmqr](#).

p?ggrqf

Computes the generalized RQ factorization.

Syntax

```
call psggrqf(m, p, n, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work,
lwork, info)
```

```
call pdggrqf(m, p, n, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work,
lwork, info)
```

```
call pcggrqf(m, p, n, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work,
lwork, info)
```

```
call pzggrqf(m, p, n, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work,
lwork, info)
```

Description

The routine forms the generalized RQ factorization of an m -by- n matrix $\text{sub}(A) = (ia:ia+m-1, ja:ja+n-1)$ and a p -by- n matrix $\text{sub}(B) = (ib:ib+p-1, ja:ja+n-1)$:

$$\text{sub}(A) = R^*Q, \quad \text{sub}(B) = Z^*T^*Q,$$

where Q is an n -by- n orthogonal/unitary matrix, Z is a p -by- p orthogonal/unitary matrix, and R and T assume one of the forms:

$$R = \begin{pmatrix} m & 0 & R_{12} \\ n-m & m & \end{pmatrix}, \text{ if } m \leq n,$$

or

$$R = \begin{pmatrix} R_{11} & m-n \\ R_{12} & n \end{pmatrix}, \text{ if } m > n$$

where R_{11} or R_{21} is upper triangular, and

$$T = \begin{pmatrix} T_{11} \\ 0 \end{pmatrix} \begin{matrix} n \\ p - n \end{matrix}, \text{ if } p \geq n$$

or

$$T = \begin{pmatrix} T_{11} & T_{12} \\ & \end{pmatrix} \begin{matrix} p \\ n - p \end{matrix}, \text{ if } p < n,$$

where T^{11} is upper triangular.

In particular, if $\text{sub}(B)$ is square and nonsingular, the GRQ factorization of $\text{sub}(A)$ and $\text{sub}(B)$ implicitly gives the RQ factorization of $\text{sub}(A) * \text{inv}(\text{sub}(B))$:

$$\text{sub}(A) * \text{inv}(\text{sub}(B)) = (R * \text{inv}(T)) * Z'$$

where $\text{inv}(\text{sub}(B))$ denotes the inverse of the matrix $\text{sub}(B)$, and Z' denotes the transpose (conjugate transpose) of matrix Z .

Input Parameters

m	(global) INTEGER. The number of rows in the distributed matrices $\text{sub}(A)$ ($m \geq 0$).
p	INTEGER. The number of rows in the distributed matrix $\text{sub}(B)$ ($p \geq 0$).
n	(global) INTEGER. The number of columns in the distributed matrices $\text{sub}(A)$ and $\text{sub}(B)$ ($n \geq 0$).
a	(local) REAL for psggrqf DOUBLE PRECISION for pdggrqf COMPLEX for pcggrqf DOUBLE COMPLEX for pzggrqf. Pointer into the local memory to an array of dimension $(lld\_a, LOCC(ja+n-1))$. Contains the local pieces of the m -by- n distributed matrix $\text{sub}(A)$ to be factored.

<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>b</i>	(local) REAL for psggrqf DOUBLE PRECISION for pdggrqf COMPLEX for pcggrqf DOUBLE COMPLEX for pzggrqf. Pointer into the local memory to an array of dimension (<i>lld_b</i> , <i>LOCc(jb+n-1)</i>). Contains the local pieces of the <i>p</i> -by- <i>n</i> matrix sub(<i>B</i>) to be factored.
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the global array <i>b</i> indicating the first row and the first column of the submatrix <i>B</i> , respectively.
<i>descb</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>B</i> .
<i>work</i>	(local) REAL for psggrqf DOUBLE PRECISION for pdggrqf COMPLEX for pcggrqf DOUBLE COMPLEX for pzggrqf. Workspace array of dimension of <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER. Dimension of <i>work</i> , must be at least <i>lwork</i> ≥ $\max(\text{mb\_a} * (\text{mpa0} + \text{nqa0} + \text{mb\_a}),$ $\max((\text{mb\_a} * (\text{mb\_a} - 1)) / 2, (\text{ppb0} + \text{nqb0}) * \text{mb\_a}) +$ $\text{mb\_a} * \text{mb\_a}, \text{nb\_b} * (\text{ppb0} + \text{nqb0} + \text{nb\_b})), \text{ where}$ $\text{irow} = \text{indxg2p}(\text{ia}, \text{mb\_a}, \text{MYROW}, \text{rsrc\_a}, \text{NPROW}),$ $\text{iacol} = \text{indxg2p}(\text{ja}, \text{nb\_a}, \text{MYCOL}, \text{csrc\_a}, \text{NPCOL}),$ $\text{mpa0} = \text{numroc}(\text{m} + \text{irow}, \text{mb\_a}, \text{MYROW}, \text{iarow},$ $\text{NPROW}),$

```

nqa0 = numroc (m+icoffa, nb_a, MYCOL, iacol,
NPCOL)
iroffb = mod(ib-1, mb_b),
icoffb = mod(jb-1, nb_b),
ibrow = indxg2p(ib, mb_b, MYROW, rsrc_b, NPROW
),
ibcol = indxg2p(jb, nb_b, MYCOL, csrc_b, NPCOL
),
ppb0 = numroc (p+iroffb, mb_b, MYROW, ibrow,
NPROW),
nqb0 = numroc (n+icoffb, nb_b, MYCOL, ibcol,
NPCOL)

```

and numroc, indxg2p are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

a

On exit, if $m \leq n$, the upper triangle of $A(ia:ia+m-1, ja+n-m:ja+n-1)$ contains the m -by- m upper triangular matrix R ; if $m \geq n$, the elements on and above the $(m-n)$ -th subdiagonal contain the m -by- n upper trapezoidal matrix R ; the remaining elements, with the array *taua*, represent the orthogonal/unitary matrix Q as a product of $\min(n, m)$ elementary reflectors (see *Application Notes* below).

taua, taub

(local)

REAL for psggqrf

DOUBLE PRECISION for pdggqrf

COMPLEX for pcggqrf

DOUBLE COMPLEX for pzggqrf.

Arrays, DIMENSION *LOCr*(*ia+m-1*) for *taua* and *LOCc*(*jb+min(p, n)-1*) for *taub*.

The array *taua* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Q*. *taua* is tied to the distributed matrix *A*. (See *Application Notes* below).

The array *taub* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Z*. *taub* is tied to the distributed matrix *B*. (See *Application Notes* below).

work(1)

On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info

(global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

$$Q = H(ia) * H(ia+1) * \dots * H(ia+k-1),$$

where $k = \min(m, n)$.

Each $H(i)$ has the form

$$H(i) = I - \tau_{aua} v v'$$

where τ_{aua} is a real/complex scalar, and v is a real/complex vector with $v(n-k+i+1:n) = 0$ and $v(n-k+i) = 1$; $v(1:n-k+i-1)$ is stored on exit in $A(ia+m-k+i-1, ja:ja+n-k+i-2)$, and τ_{aua} in $\tau_{aua}(ia+m-k+i-1)$. To form *Q* explicitly, use ScaLAPACK subroutine [p?orgqr/p?ungrq](#). To use *Q* to update another matrix, use ScaLAPACK subroutine [p?ormrq/p?unmrq](#).

The matrix *Z* is represented as a product of elementary reflectors

$$Z = H(jb) * H(jb+1) * \dots * H(jb+k-1), \text{ where } k = \min(p, n).$$

Each $H(i)$ has the form

$$H(i) = I - \tau_{aub} v v'$$

where taub is a real/complex scalar, and v is a real/complex vector with $\mathit{v}(1:i-1) = 0$ and $\mathit{v}(i) = 1$; $\mathit{v}(i+1:p)$ is stored on exit in $\mathit{B}(\mathit{ib}+i:\mathit{ib}+p-1, \mathit{jb}+i-1)$, and taub in $\mathit{taub}(\mathit{jb}+i-1)$. To form z explicitly, use ScaLAPACK subroutine `p?orgqr/p?ungqr`. To use z to update another matrix, use ScaLAPACK subroutine `p?ormqr/p?unmqr`.

Symmetric Eigenproblems

To solve a symmetric eigenproblem with ScaLAPACK, you usually need to reduce the matrix to real tridiagonal form T and then find the eigenvalues and eigenvectors of the tridiagonal matrix T . ScaLAPACK includes routines for reducing the matrix to a tridiagonal form by an orthogonal (or unitary) similarity transformation $\mathit{A} = \mathit{Q}\mathit{T}\mathit{Q}^H$ as well as for solving tridiagonal symmetric eigenvalue problems. These routines are listed in [Table 6-4](#).

There are different routines for symmetric eigenproblems, depending on whether you need eigenvalues only or eigenvectors as well, and on the algorithm used (either the $\mathit{Q}\mathit{T}\mathit{Q}$ algorithm, or bisection followed by inverse iteration).

Table 6-4 Computational Routines for Solving Symmetric Eigenproblems

Operation	Dense symmetric/Hermitian matrix	Orthogonal/unitary matrix	Symmetric tridiagonal matrix
Reduce to tridiagonal form $\mathit{A} = \mathit{Q}\mathit{T}\mathit{Q}^H$	<code>p?sytrd/p?hetrd</code>		
Multiply matrix after reduction		<code>p?ormtr/p?unmtr</code>	
Find all eigenvalues and eigenvectors of a tridiagonal matrix T by a $\mathit{Q}\mathit{T}\mathit{Q}$ method			<code>steqr2</code> *)
Find selected eigenvalues of a tridiagonal matrix T via bisection			<code>p?stebz</code>
Find selected eigenvectors of a tridiagonal matrix T by inverse iteration			<code>p?stein</code>

*) This routine is described as part of auxiliary ScaLAPACK routines.

p?sytrd

Reduces a symmetric matrix to real symmetric tridiagonal form by an orthogonal similarity transformation.

Syntax

```
call pssytrd( uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
call pdsytrd( uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
```

Description

This routine reduces a real symmetric matrix $\text{sub}(A)$ to symmetric tridiagonal form T by an orthogonal similarity transformation:

$$Q' * \text{sub}(A) * Q = T,$$

where $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$.

Input Parameters

<i>uplo</i>	(global) CHARACTER. Specifies whether the upper or lower triangular part of the symmetric matrix $\text{sub}(A)$ is stored: If <i>uplo</i> = 'U', upper triangular If <i>uplo</i> = 'L', lower triangular
<i>n</i>	(global) INTEGER. The order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
<i>a</i>	(local) REAL for pssytrd DOUBLE PRECISION for pdsytrd. Pointer into the local memory to an array of dimension (<i>lld_a</i> , <i>LOCc(ja+n-1)</i>). On entry, this array contains the local pieces of the symmetric distributed matrix $\text{sub}(A)$. If <i>uplo</i> = 'U', the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.

	If <i>uplo</i> = 'L', the leading <i>n</i> -by- <i>n</i> lower triangular part of sub(<i>A</i>) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced. See <i>Application Notes</i> below.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>work</i>	(local) REAL for pssytrd DOUBLE PRECISION for pdsytrd. Workspace array of dimension <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, dimension of <i>work</i> , must be at least: $lwork \geq \max(NB * (np + 1), 3 * NB),$ where $NB = mb\_a = nb\_a$, $np = \text{numroc}(n, NB, MYROW, iarow, NPROW)$, $iarow = \text{indxg2p}(ia, NB, MYROW, rsrc\_a, NPROW)$. indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine <code>blacs_gridinfo</code> . If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by <code>pserbla</code> .

Output Parameters

<i>a</i>	On exit, if <i>uplo</i> = 'U', the diagonal and first superdiagonal of sub(<i>A</i>) are overwritten by the corresponding elements of the tridiagonal matrix <i>T</i> , and the elements above the first superdiagonal, with the array <i>tau</i> , represent the orthogonal matrix <i>Q</i> as a product of elementary reflectors; if <i>uplo</i> = 'L', the diagonal and first subdiagonal of sub(<i>A</i>) are overwritten by the corresponding elements of the tridiagonal matrix <i>T</i> , and the elements below the first subdiagonal, with

the array *tau*, represent the orthogonal matrix *Q* as a product of elementary reflectors. See *Application Notes* below.

<i>d</i>	(local) REAL for pssytrd DOUBLE PRECISION for pdsytrd. Arrays, DIMENSION <i>LOCc</i>(<i>ja+n-1</i>) .The diagonal elements of the tridiagonal matrix <i>T</i>: <i>d</i>(<i>i</i>) = <i>A</i>(<i>i</i>,<i>i</i>) . <i>d</i> is tied to the distributed matrix <i>A</i>.
<i>e</i>	(local) REAL for pssytrd DOUBLE PRECISION for pdsytrd. Arrays, DIMENSION <i>LOCc</i>(<i>ja+n-1</i>) if <i>uplo</i> = 'U', <i>LOCc</i>(<i>ja+n-2</i>) otherwise. The off-diagonal elements of the tridiagonal matrix <i>T</i>: <i>e</i>(<i>i</i>) = <i>A</i>(<i>i</i>,<i>i+1</i>) if <i>uplo</i> = 'U', <i>e</i>(<i>i</i>) = <i>A</i>(<i>i+1</i>,<i>i</i>) if <i>uplo</i> = 'L'. <i>e</i> is tied to the distributed matrix <i>A</i>.
<i>tau</i>	(local) REAL for pssytrd DOUBLE PRECISION for pdsytrd. Arrays, DIMENSION <i>LOCc</i>(<i>ja+n-1</i>) . This array contains the scalar factors <i>tau</i> of the elementary reflectors. <i>tau</i> is tied to the distributed matrix <i>A</i>.
<i>work</i> (1)	On exit <i>work</i>(1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i>-th argument is an array and the <i>j</i>-entry had an illegal value, then <i>info</i> = - (<i>i</i>* 100+<i>j</i>), if the <i>i</i>-th argument is a scalar and had an illegal value, then <i>info</i> = -<i>i</i>.

Application Notes

If *uplo* = 'U', the matrix *Q* is represented as a product of elementary reflectors

$$Q = H(n-1) \dots H(2) H(1).$$

Each $H(i)$ has the form

$$H(i) = I - \tau v v',$$

where τ is a real scalar, and v is a real vector with $v(i+1:n) = 0$ and $v(i) = 1$; $v(1:i-1)$ is stored on exit in $A(ia:ia+i-2, ja+i)$, and τ in $\tau(ja+i-1)$.

If $uplo = 'L'$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(1) H(2) \dots H(n-1).$$

Each $H(i)$ has the form

$$H(i) = I - \tau v v',$$

where τ is a real scalar, and v is a real vector with $v(1:i) = 0$ and $v(i+1) = 1$; $v(i+2:n)$ is stored on exit in $A(ia+i+1:ia+n-1, ja+i-1)$, and τ in $\tau(ja+i-1)$.

The contents of $sub(A)$ on exit are illustrated by the following examples with $n = 5$:

If $uplo = 'U'$:

$$\begin{bmatrix} d & e & v2 & v3 & v4 \\ & d & e & v3 & v4 \\ & & d & e & v4 \\ & & & d & e \\ & & & & d \end{bmatrix}$$

If $uplo = 'L'$:

$$\begin{bmatrix} d & & & & \\ e & d & & & \\ v1 & e & d & & \\ v1 & v2 & e & d & \\ v1 & v2 & v3 & e & d \end{bmatrix}$$

where d and e denote diagonal and off-diagonal elements of T , and $v\ i$ denotes an element of the vector defining $H(i)$.

p?ormtr

Multiplies a general matrix by the orthogonal transformation matrix from a reduction to tridiagonal form determined by p?sytrd.

Syntax

```
call psormtr( side, uplo, trans, m, n, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)

call pdormtr( side, uplo, trans, m, n, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

Description

The routine overwrites the general real distributed m -by- n matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
$trans = 'T':$	$Q^T * \text{sub}(C)$	$\text{sub}(C) * Q^T$

where Q is a real orthogonal distributed matrix of order nq , with $nq = m$ if $side = 'L'$ and $nq = n$ if $side = 'R'$.

Q is defined as the product of nq elementary reflectors, as returned by p?sytrd.

If $uplo = 'U'$, $Q = H(nq-1) \dots H(2) H(1)$;

If $uplo = 'L'$, $Q = H(1) H(2) \dots H(nq-1)$.

Input Parameters

$side$	(global) CHARACTER $= 'L'$: Q or Q^T is applied from the left. $= 'R'$: Q or Q^T is applied from the right.
$trans$	(global) CHARACTER $= 'N'$, no transpose, Q is applied. $= 'T'$, transpose, Q^T is applied.
$uplo$	(global) CHARACTER.

	= 'U': Upper triangle of $A(ia:*, ja:*)$ contains elementary reflectors from $p?sytrd$; = 'L': Lower triangle of $A(ia:*, ja:*)$ contains elementary reflectors from $p?sytrd$
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix $sub(C)$ ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix $sub(C)$ ($n \geq 0$).
<i>a</i>	(local) REAL for $psormtr$ DOUBLE PRECISION for $pdormtr$. Pointer into the local memory to an array of dimension $(lld\_a, LOCC(ja+m-1))$ if $side='L'$, or $(lld\_a, LOCC(ja+n-1))$ if $side = 'R'$. Contains the vectors which define the elementary reflectors, as returned by $p?sytrd$. If $side='L', lld\_a \geq \max(1, LOCr(ia+m-1))$; If $side = 'R', lld\_a \geq \max(1, LOCr(ia+n-1))$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension $(dlen\_)$. The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) REAL for $psormtr$ DOUBLE PRECISION for $pdormtr$. Array, DIMENSION of <i>ltau</i> where if $side = 'L'$ and $uplo = 'U'$, $ltau = LOCC(m\_a)$, if $side = 'L'$ and $uplo = 'L'$, $ltau = LOCC(ja+m-2)$, if $side = 'R'$ and $uplo = 'U'$, $ltau = LOCC(n\_a)$, if $side = 'R'$ and $uplo = 'L'$, $ltau = LOCC(ja+n-2)$. $tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by $p?sytrd$. <i>tau</i> is tied to the distributed matrix <i>A</i>.
<i>c</i>	(local) REAL for $psormtr$

	DOUBLE PRECISION for pdormtr. Pointer into the local memory to an array of dimension (<i>lld_a</i> , <i>LOCc</i> (<i>ja+n-1</i>)). Contains the local pieces of the distributed matrix sub (<i>c</i>).
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the global array <i>c</i> indicating the first row and the first column of the submatrix <i>c</i> , respectively.
<i>desc</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>c</i> .
<i>work</i>	(local) REAL for psormtr DOUBLE PRECISION for pdormtr. Workspace array of dimension <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, dimension of <i>work</i> , must be at least: <pre> if uplo = 'U', iaa= ia; jaa= ja+1, icc= ic; jcc= jc; else uplo = 'L', iaa= ia+1, jaa= ja; If side = 'L', icc= ic+1; jcc= jc; else icc= ic; jcc= jc+1; end if end if If side = 'L', mi= m-1; ni= n lwork ≥ max((nb_a*(nb_a-1))/2, (nqc0 + mpc0)*nb_a) + nb_a*nb_a else If side = 'R', mi= m; ni = n-1; lwork ≥ max((nb_a*(nb_a-1))/2, (nqc0 + max(npa0+numroc(numroc(ni+icoffc, nb_a, 0, 0, NPCOL), nb_a, 0, 0, lcmq), mpc0))*nb_a)+ nb_a*nb_a end if </pre>

where $lcmq = lcm/NPCOL$ with $lcm = ilcm(NPROW, NPCOL)$,

```

iroffa = mod(iaa-1, mb_a),
icoffa = mod(jaa-1, nb_a),
iarow = indxg2p(iaa, mb_a, MYROW, rsrc_a, NPROW),
npa0 = numroc(ni+iroffa, mb_a, MYROW, iarow, NPROW),
iroffc = mod(icc-1, mb_c),
icoffc = mod(jcc-1, nb_c),
icrow = indxg2p(icc, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jcc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(mi+iroffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(ni+icoffc, nb_c, MYCOL, iccol, NPCOL),

```

$ilcm$, $indxg2p$ and $numroc$ are ScaLAPACK tool functions; $MYROW$, $MYCOL$, $NPROW$ and $NPCOL$ can be determined by calling the subroutine `blacs_gridinfo`. If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

c	Overwritten by the product $Q*\text{sub}(c)$, or $Q'*\text{sub}(c)$, or $\text{sub}(c)*Q'$, or $\text{sub}(c)*Q$.
$work(1)$	On exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance.
$info$	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -entry had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

p?hetrd

Reduces a Hermitian matrix to Hermitian tridiagonal form by a unitary similarity transformation.

Syntax

```
call pchetrd( uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
call pzhetrd( uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
```

Description

This routine reduces a complex Hermitian matrix $\text{sub}(A)$ to Hermitian tridiagonal form T by a unitary similarity transformation:

$$Q'^* \text{sub}(A) Q = T$$

where $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$.

Input Parameters

<i>uplo</i>	(global) CHARACTER. Specifies whether the upper or lower triangular part of the Hermitian matrix $\text{sub}(A)$ is stored: If <i>uplo</i> = 'U', upper triangular If <i>uplo</i> = 'L', lower triangular
<i>n</i>	(global) INTEGER. The order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
<i>a</i>	(local) COMPLEX for pchetrd DOUBLE COMPLEX for pzhetrd. Pointer into the local memory to an array of dimension (<i>lld_a</i> , <i>LOCc(ja+n-1)</i>). On entry, this array contains the local pieces of the Hermitian distributed matrix $\text{sub}(A)$. If <i>uplo</i> = 'U', the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.

	If <code>uplo = 'L'</code>, the leading n-by-n lower triangular part of <code>sub(A)</code> contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced. (see <i>Application Notes</i> below).
<code>ia, ja</code>	(global) <code>INTEGER</code>. The row and column indices in the global array <code>a</code> indicating the first row and the first column of the submatrix <code>A</code>, respectively.
<code>desca</code>	(global and local) <code>INTEGER</code> array, dimension (<code>dlen_</code>). The array descriptor for the distributed matrix <code>A</code>.
<code>work</code>	(local) <code>COMPLEX</code> for <code>pchetrd</code> <code>DOUBLE COMPLEX</code> for <code>pzhetrdr</code>. Workspace array of dimension <code>lwork</code>.
<code>lwork</code>	(local or global) <code>INTEGER</code>, dimension of <code>work</code>, must be at least: $lwork \geq \max(NB * (np + 1), 3 * NB)$ where $NB = mb\_a = nb\_a$, $np = \text{numroc}(n, NB, MYROW, iarow, NPROW)$, $iarow = \text{indxg2p}(ia, NB, MYROW, rsrc\_a, NPROW)$. <code>indxg2p</code> and <code>numroc</code> are ScaLAPACK tool functions; <code>MYROW</code>, <code>MYCOL</code>, <code>NPROW</code> and <code>NPCOL</code> can be determined by calling the subroutine <code>blacs_gridinfo</code>. If <code>lwork = -1</code>, then <code>lwork</code> is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by <code>pxerbla</code>.

Output Parameters

<code>a</code>	On exit, If <code>uplo = 'U'</code>, the diagonal and first superdiagonal of <code>sub(A)</code> are overwritten by the corresponding elements of the tridiagonal matrix <code>T</code>, and the elements above the first superdiagonal, with the array <code>tau</code>, represent the unitary matrix <code>Q</code> as a product of elementary reflectors; if <code>uplo = 'L'</code>, the diagonal and first subdiagonal of <code>sub(A)</code> are overwritten by the corresponding elements of the tridiagonal

	matrix T , and the elements below the first subdiagonal, with the array τ , represent the unitary matrix Q as a product of elementary reflectors (see <i>Application Notes</i> below).
d	(local) REAL for pchetrd DOUBLE PRECISION for pzhetrd. Arrays, DIMENSION $LOCc(ja+n-1)$. The diagonal elements of the tridiagonal matrix T: $d(i) = A(i, i)$. d is tied to the distributed matrix A.
e	(local) REAL for pchetrd DOUBLE PRECISION for pzhetrd. Arrays, DIMENSION $LOCc(ja+n-1)$ if $uplo = 'U'$; $LOCc(ja+n-2)$ - otherwise. The off-diagonal elements of the tridiagonal matrix T: $e(i) = A(i, i+1)$ if $uplo = 'U'$, $e(i) = A(i+1, i)$ if $uplo = 'L'$. e is tied to the distributed matrix A.
τ	(local) COMPLEX for pchetrd DOUBLE COMPLEX for pzhetrd. Arrays, DIMENSION $LOCc(ja+n-1)$. This array contains the scalar factors τ of the elementary reflectors. τ is tied to the distributed matrix A.
$work(1)$	On exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance.
$info$	(global) INTEGER. = 0: the execution is successful. < 0: if the i-th argument is an array and the j-entry had an illegal value, then $info = -(i*100+j)$, if the i-th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

If $uplo = 'U'$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(n-1) * \dots * H(2) * H(1).$$

Each $H(i)$ has the form

$$H(i) = I - \tau v v',$$

where τ is a complex scalar, and v is a complex vector with $v(i+1:n) = 0$ and $v(i) = 1$; $v(1:i-1)$ is stored on exit in $A(ia:ia+i-2, ja+i)$, and τ in $\tau(ja+i-1)$.

If $uplo = 'L'$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(1) * H(2) * \dots * H(n-1).$$

Each $H(i)$ has the form

$$H(i) = I - \tau v v',$$

where τ is a complex scalar, and v is a complex vector with $v(1:i) = 0$ and $v(i+1) = 1$; $v(i+2:n)$ is stored on exit in $A(ia+i+1:ia+n-1, ja+i-1)$, and τ in $\tau(ja+i-1)$.

The contents of $\text{sub}(A)$ on exit are illustrated by the following examples with $n = 5$:

If $uplo = 'U'$:

$$\begin{bmatrix} d & e & v2 & v3 & v4 \\ & d & e & v3 & v4 \\ & & d & e & v4 \\ & & & d & e \\ & & & & d \end{bmatrix}$$

If $uplo = 'L'$:

$$\begin{bmatrix} d & & & & \\ e & d & & & \\ v1 & e & d & & \\ v1 & v2 & e & d & \\ v1 & v2 & v3 & e & d \end{bmatrix}$$

where d and e denote diagonal and off-diagonal elements of T , and $v\ i$ denotes an element of the vector defining $H(i)$.

p?unmtr

Multiplies a general matrix by the unitary transformation matrix from a reduction to tridiagonal form determined by p?hetrd.

Syntax

```
call pcunmtr( side, uplo, trans, m, n, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)

call pzunmtr( side, uplo, trans, m, n, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

Description

The routine overwrites the general complex distributed m -by- n matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
$trans = 'C':$	$Q^H * \text{sub}(C)$	$\text{sub}(C) * Q^H$

where Q is a complex unitary distributed matrix of order nq , with $nq = m$ if $side = 'L'$ and $nq = n$ if $side = 'R'$.

Q is defined as the product of $nq-1$ elementary reflectors, as returned by p?hetrd.

If $uplo = 'U'$, $Q = H(nq-1) \dots H(2) H(1)$;

If $uplo = 'L'$, $Q = H(1) H(2) \dots H(nq-1)$.

Input Parameters

$side$	(global) CHARACTER $= 'L'$: Q or Q^H is applied from the left. $= 'R'$: Q or Q^H is applied from the right.
$trans$	(global) CHARACTER $= 'N'$, no transpose, Q is applied. $= 'C'$, conjugate transpose, Q^H is applied.
$uplo$	(global) CHARACTER.

	= 'U': Upper triangle of $A(ia:*, ja:*)$ contains elementary reflectors from <code>p?hetrd</code>; = 'L': Lower triangle of $A(ia:*, ja:*)$ contains elementary reflectors from <code>p?hetrd</code>
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix <code>sub(c)</code> ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix <code>sub(c)</code> ($n \geq 0$).
<i>a</i>	(local) REAL for <code>pcunmtr</code> DOUBLE PRECISION for <code>pzunmtr</code>. Pointer into the local memory to an array of dimension $(lld\_a, LOCc(ja+m-1))$ if <code>side='L'</code>, or $(lld\_a, LOCc(ja+n-1))$ if <code>side = 'R'</code>. Contains the vectors which define the elementary reflectors, as returned by <code>p?hetrd</code>. If <code>side='L'</code>, $lld\_a \geq \max(1, LOCr(ia+m-1))$; If <code>side = 'R'</code>, $lld\_a \geq \max(1, LOCr(ia+n-1))$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) COMPLEX for <code>pcunmtr</code> DOUBLE COMPLEX for <code>pzunmtr</code>. Array, DIMENSION of <i>ltau</i> where If <code>side = 'L'</code> and <code>uplo = 'U'</code>, $ltau = LOCc(m\_a)$, if <code>side = 'L'</code> and <code>uplo = 'L'</code>, $ltau = LOCc(ja+m-2)$, if <code>side = 'R'</code> and <code>uplo = 'U'</code>, $ltau = LOCc(n\_a)$, if <code>side = 'R'</code> and <code>uplo = 'L'</code>, $ltau = LOCc(ja+n-2)$. <i>tau(i)</i> must contain the scalar factor of the elementary reflector $H(i)$, as returned by <code>p?hetrd</code>. <i>tau</i> is tied to the distributed matrix <i>A</i>.
<i>c</i>	(local) COMPLEX for <code>pcunmtr</code>

	DOUBLE COMPLEX for pzunmtr. Pointer into the local memory to an array of dimension (<i>lld_a</i> , <i>LOCc</i> (<i>ja</i> + <i>n</i> -1)). Contains the local pieces of the distributed matrix sub (<i>c</i>).
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the global array <i>c</i> indicating the first row and the first column of the submatrix <i>c</i> , respectively.
<i>descc</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>c</i> .
<i>work</i>	(local) COMPLEX for pcunmtr DOUBLE COMPLEX for pzunmtr. Workspace array of dimension <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, dimension of <i>work</i> , must be at least: If <i>uplo</i> = 'U', <i>iaa</i> = <i>ia</i> ; <i>jaa</i> = <i>ja</i> +1, <i>icc</i> = <i>ic</i> ; <i>jcc</i> = <i>jc</i> ; else <i>uplo</i> = 'L', <i>iaa</i> = <i>ia</i> +1, <i>jaa</i> = <i>ja</i> ; If <i>side</i> = 'L', <i>icc</i> = <i>ic</i> +1; <i>jcc</i> = <i>jc</i> ; else <i>icc</i> = <i>ic</i> ; <i>jcc</i> = <i>jc</i> +1; end if end if If <i>side</i> = 'L', <i>mi</i> = <i>m</i> -1; <i>ni</i> = <i>n</i> <i>lwork</i> ≥ max((<i>nb_a</i> *(<i>nb_a</i> -1))/2, (<i>nqc0</i> + <i>mpc0</i>)* <i>nb_a</i>) + <i>nb_a</i> * <i>nb_a</i> else If <i>side</i> = 'R', <i>mi</i> = <i>m</i> ; <i>ni</i> = <i>n</i> -1; <i>lwork</i> ≥ max((<i>nb_a</i> *(<i>nb_a</i> -1))/2, (<i>nqc0</i> + max(<i>npa0</i> +numroc(numroc(<i>ni</i> + <i>icoffc</i> , <i>nb_a</i> , 0, 0, NPCOL), <i>nb_a</i> , 0, 0, <i>lcmq</i>), <i>mpc0</i>))* <i>nb_a</i>) + <i>nb_a</i> * <i>nb_a</i> end if

where $lcmq = lcm/NPCOL$ with $lcm = ilcm(NPROW, NPCOL)$,

```

iroffa = mod(iaa-1, mb_a),
icoffa = mod(jaa-1, nb_a),
iarow = indxg2p(iaa, mb_a, MYROW, rsrc_a, NPROW),
npa0 = numroc(ni+iroffa, mb_a, MYROW, iarow, NPROW),
iroffc = mod(icc-1, mb_c),
icoffc = mod(jcc-1, nb_c),
icrow = indxg2p(icc, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jcc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(mi+iroffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(ni+icoffc, nb_c, MYCOL, iccol, NPCOL),
ilcm, indxg2p and numroc are ScaLAPACK tool functions;
MYROW, MYCOL, NPROW and NPCOL can be determined by
calling the subroutine blacs_gridinfo. If lwork = -1,
then lwork is global input and a workspace query is
assumed; the routine only calculates the minimum and
optimal size for all work arrays. Each of these values is
returned in the first entry of the corresponding work array,
and no error message is issued by p_xerbla.

```

Output Parameters

<i>c</i>	Overwritten by the product $Q \cdot \text{sub}(c)$, or $Q' \cdot \text{sub}(c)$, or $\text{sub}(c) \cdot Q'$, or $\text{sub}(c) \cdot Q$.
<i>work</i> (1)	On exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = - <i>i</i>

p?stebz

Computes the eigenvalues of a symmetric tridiagonal matrix by bisection.

Syntax

```
call psstebz(ictxt, range, order, n, vl, vu, il, iu, abstol, d, e, m, nsplit,
w, iblock, isplit, work, iwork, liwork, info)

call pdstebz(ictxt, range, order, n, vl, vu, il, iu, abstol, d, e, m, nsplit,
w, iblock, isplit, work, iwork, liwork, info)
```

Description

This routine computes the eigenvalues of a symmetric tridiagonal matrix in parallel. These may be all eigenvalues, all eigenvalues in the interval $[vl\ vu]$, or the eigenvalues indexed il through iu . A static partitioning of work is done at the beginning of `p?stebz` which results in all processes finding an (almost) equal number of eigenvalues.

Input Parameters

<code>ictxt</code>	(global) INTEGER. The BLACS context handle.
<code>range</code>	(global) CHARACTER. Must be 'A' or 'V' or 'I'. If <code>range</code> = 'A', the routine computes all eigenvalues. If <code>range</code> = 'V', the routine computes eigenvalues in the interval $[vl, vu]$. If <code>range</code> = 'I', the routine computes eigenvalues with indices il to iu .
<code>order</code>	(global) CHARACTER. Must be 'B' or 'E'. If <code>order</code> = 'B', the eigenvalues are to be ordered from smallest to largest within each split-off block. If <code>order</code> = 'E', the eigenvalues for the entire matrix are to be ordered from smallest to largest.
<code>n</code>	(global) INTEGER. The order of the tridiagonal matrix T ($n \geq 0$).
<code>vl, vu</code>	(global) REAL for <code>psstebz</code> DOUBLE PRECISION for <code>pdstebz</code> .

If *range* = 'V', the routine computes the lower and the upper bounds for the eigenvalues on the interval [*l*, *vu*].
 If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, iu
 (global)
 INTEGER. Constraint: $1 \leq il \leq iu \leq n$.
 If *range* = 'I', the index of the smallest eigenvalue is returned for *il* and of the largest eigenvalue for *iu* (assuming that the eigenvalues are in ascending order) must be returned. *il* must be at least 1. *iu* must be at least *il* and no greater than *n*.
 If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol
 (global)
 REAL for psstebz
 DOUBLE PRECISION for pdstebz.
 The absolute tolerance to which each eigenvalue is required. An eigenvalue (or cluster) is considered to have converged if it lies in an interval of width *abstol*. If *abstol* ≤ 0, then the tolerance is taken as $ulp ||T||$, where *ulp* is the machine precision, and $||T||$ means the 1-norm of *T*. Eigenvalues will be computed most accurately when *abstol* is set to the underflow threshold `slamch('U')`, not 0. Note that if eigenvectors are desired later by inverse iteration (`p?stein`), *abstol* should be set to $2 * p?lamch('S')$.

d
 (global)
 REAL for psstebz
 DOUBLE PRECISION for pdstebz.
 Array, DIMENSION (*n*).
 Contains *n* diagonal elements of the tridiagonal matrix *T*. To avoid overflow, the matrix must be scaled so that its largest entry is no greater than the $overflow^{(1/2)} * underflow^{(1/4)}$ in absolute value, and for greatest accuracy, it should not be much smaller than that.

e
 (global)
 REAL for psstebz
 DOUBLE PRECISION for pdstebz.
 Array, DIMENSION (*n* - 1).

	Contains $(n-1)$ off-diagonal elements of the tridiagonal matrix T . To avoid overflow, the matrix must be scaled so that its largest entry is no greater than $\text{overflow}^{(1/2)} * \text{underflow}^{(1/4)}$ in absolute value, and for greatest accuracy, it should not be much smaller than that.
<i>work</i>	(local) REAL for psstebz DOUBLE PRECISION for pdstebz. Array, DIMENSION $\max(5n, 7)$. This is a workspace array.
<i>lwork</i>	(local) INTEGER. The size of the <i>work</i> array must be $\geq \max(5n, 7)$. If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by p _x erbla.
<i>iwork</i>	(local) INTEGER. Array, DIMENSION $\max(4n, 14)$. This is a workspace array.
<i>liwork</i>	(local) INTEGER. the size of the <i>iwork</i> array must $\geq \max(4n, 14, \text{NPROCS})$. If <i>liwork</i> = -1, then <i>liwork</i> is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by p _x erbla.

Output Parameters

<i>m</i>	(global) INTEGER. The actual number of eigenvalues found. $0 \leq m \leq n$
<i>nsplit</i>	(global) INTEGER. The number of diagonal blocks detected in T . $1 \leq \text{nsplit} \leq n$
<i>w</i>	(global) REAL for psstebz DOUBLE PRECISION for pdstebz.

iblock

Array, DIMENSION (n). On exit, the first m elements of w contain the eigenvalues on all processes.

(global) INTEGER.

Array, DIMENSION (n). At each row/column j where $e(j)$ is zero or small, the matrix T is considered to split into a block diagonal matrix. On exit *iblock*(i) specifies which block (from 1 to the number of blocks) the eigenvalue $w(i)$ belongs to.



NOTE. In the (theoretically impossible) event that bisection does not converge for some or all eigenvalues, *info* is set to 1 and the ones for which it did not are identified by a negative block number.

isplit

(global) INTEGER.

Array, DIMENSION (n).

Contains the splitting points, at which T breaks up into submatrices. The first submatrix consists of rows/columns 1 to *isplit*(1), the second of rows/columns *isplit*(1)+1 through *isplit*(2), etc., and the *nsplit*-th consists of rows/columns *isplit*(*nsplit*-1)+1 through *isplit*(*nsplit*)= n . (Only the first *nsplit* elements are used, but since the *nsplit* values are not known, n words must be reserved for *isplit*.)

info

(global) INTEGER.

If *info* = 0, the execution is successful.

If *info* < 0, if *info* = $-i$, the i -th argument has an illegal value.

If *info* > 0, some or all of the eigenvalues fail to converge or not computed.

If *info* = 1, bisection fails to converge for some eigenvalues; these eigenvalues are flagged by a negative block number. The effect is that the eigenvalues may not be as accurate as the absolute and relative tolerances.

If *info* = 2, mismatch between the number of eigenvalues output and the number desired.

If `info = 3: range='i'`, and the Gershgorin interval initially used is incorrect. No eigenvalues are computed. Probable cause: the machine has a sloppy floating point arithmetic. Increase the `fudge` parameter, recompile, and try again.

p?stein

Computes the eigenvectors of a tridiagonal matrix using inverse iteration.

Syntax

```
call psstein(n, d, e, m, w, iblock, isplit, orfac, z, iz, jz, descz, work,
lwork, iwork, liwork, ifail, iclustr, gap, info)

call pdstein(n, d, e, m, w, iblock, isplit, orfac, z, iz, jz, descz, work,
lwork, iwork, liwork, ifail, iclustr, gap, info)

call pcstein(n, d, e, m, w, iblock, isplit, orfac, z, iz, jz, descz, work,
lwork, iwork, liwork, ifail, iclustr, gap, info)

call pzstein(n, d, e, m, w, iblock, isplit, orfac, z, iz, jz, descz, work,
lwork, iwork, liwork, ifail, iclustr, gap, info)
```

Description

This routine computes the eigenvectors of a symmetric tridiagonal matrix T corresponding to specified eigenvalues, by inverse iteration. `p?stein` does not orthogonalize vectors that are on different processes. The extent of orthogonalization is controlled by the input parameter `lwork`. Eigenvectors that are to be orthogonalized are computed by the same process. `p?stein` decides on the allocation of work among the processes and then calls `?stein2` (modified LAPACK routine) on each individual process. If insufficient workspace is allocated, the expected orthogonalization may not be done.



NOTE. If the eigenvectors obtained are not orthogonal, increase `lwork` and run the code again.

p = `NPROW*NPCOL` is the total number of processes.

Input Parameters

<i>n</i>	(global) INTEGER. The order of the matrix T ($n \geq 0$).
<i>m</i>	(global) INTEGER. The number of eigenvectors to be returned.
<i>d</i> , <i>e</i> , <i>w</i>	(global) REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays: <i>d</i> (*) contains the diagonal elements of T . DIMENSION (<i>n</i>). <i>e</i> (*) contains the off-diagonal elements of T . DIMENSION (<i>n</i> -1). <i>w</i> (*) contains all the eigenvalues grouped by split-off block. The eigenvalues are supplied from smallest to largest within the block. (Here the output array <i>w</i> from p?stebz with order = 'B' is expected. The array should be replicated in all processes.) DIMENSION(<i>m</i>)
<i>iblock</i>	(global) INTEGER. Array, DIMENSION (<i>n</i>). The submatrix indices associated with the corresponding eigenvalues in <i>w</i> --1 for eigenvalues belonging to the first submatrix from the top, 2 for those belonging to the second submatrix, etc. (The output array <i>iblock</i> from p?stebz is expected here).
<i>isplit</i>	(global) INTEGER. Array, DIMENSION (<i>n</i>). The splitting points, at which T breaks up into submatrices. The first submatrix consists of rows/columns 1 to <i>isplit</i> (1), the second of rows/columns <i>isplit</i> (1)+1 through <i>isplit</i> (2), etc., and the <i>nsplit</i> -th consists of rows/columns <i>isplit</i> (<i>nsplit</i> -1)+1 through <i>isplit</i> (<i>nsplit</i>)= <i>n</i> . (The output array <i>isplit</i> from p?stebz is expected here.)
<i>orfac</i>	(global) REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors.

	<i>orfac</i> specifies which eigenvectors should be orthogonalized. Eigenvectors that correspond to eigenvalues within $orfac * T$ of each other are to be orthogonalized. However, if the workspace is insufficient (see <i>lwork</i>), this tolerance may be decreased until all eigenvectors can be stored in one process. No orthogonalization is done if <i>orfac</i> is equal to zero. A default value of 1000 is used if <i>orfac</i> is negative. <i>orfac</i> should be identical on all processes
<i>iz, jz</i>	(global) INTEGER. The row and column indices in the global array <i>z</i> indicating the first row and the first column of the submatrix <i>Z</i> , respectively.
<i>descz</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>Z</i> .
<i>work</i>	(local). REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Workspace array, DIMENSION (<i>lwork</i>).
<i>lwork</i>	(local) INTEGER. <i>lwork</i> controls the extent of orthogonalization which can be done. The number of eigenvectors for which storage is allocated on each process is $nvec = \text{floor}((lwork - \max(5*n, np00*mq00)) / n)$. Eigenvectors corresponding to eigenvalue clusters of size $nvec - \text{ceil}(m/p) + 1$ are guaranteed to be orthogonal (the orthogonality is similar to that obtained from ?stein2).



NOTE. *lwork* must be no smaller than $\max(5*n, np00*mq00) + \text{ceil}(m/p)*n$ and should have the same input value on all processes.

It is the minimum value of *lwork* input on different processes that is significant.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by p_xerbla.

iwork (local) INTEGER.
Workspace array, DIMENSION (3n+p+1).

liwork (local) INTEGER. The size of the array *iwork*. It must be greater than (3*n+p+1).
If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by *pxerbla*.

Output Parameters

z (local)
REAL for *psstein*
DOUBLE PRECISION for *pdstein*
COMPLEX for *pcstein*
DOUBLE COMPLEX for *pzstein*.
Array, DIMENSION (*descz(dlen_)*, n/NPCOL + NB). *z* contains the computed eigenvectors associated with the specified eigenvalues. Any vector which fails to converge is set to its current iterate after MAXIT iterations (See [?stein2](#)). On output, *z* is distributed across the *p* processes in block cyclic format.

work(1) On exit, *work*(1) gives a lower bound on the workspace (*lwork*) that guarantees the user desired orthogonalization (see *orfac*). Note that this may overestimate the minimum workspace needed.

iwork On exit, *iwork*(1) contains the amount of integer workspace required.
On exit, the *iwork*(2) through *iwork*(p+2) indicate the eigenvectors computed by each process. Process *i* computes eigenvectors indexed *iwork*(i+2)+1 through *iwork*(i+3).

ifail (global). INTEGER. Array, DIMENSION (*m*). On normal exit, all elements of *ifail* are zero. If one or more eigenvectors fail to converge after MAXIT iterations (as in [?stein](#)), then *info* > 0 is returned. If mod(*info*, m+1) > 0, then for *i*=1

to $\text{mod}(\text{info}, m+1)$, the eigenvector corresponding to the eigenvalue $w(\text{ifail}(i))$ failed to converge (w refers to the array of eigenvalues on output).

iclustr (global) INTEGER. Array, DIMENSION $(2*p)$
 This output array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be orthogonalized due to insufficient workspace (see *lwork*, *orfac* and *info*). Eigenvectors corresponding to clusters of eigenvalues indexed $\text{iclustr}(2*I-1)$ to $\text{iclustr}(2*I)$, $i = 1$ to $\text{info}/(m+1)$, could not be orthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal. *iclustr* is a zero terminated array
 $--- (\text{iclustr}(2*k) \neq 0 \text{ and } \text{iclustr}(2*k+1) = 0)$
 if and only if k is the number of clusters.

gap (global)
 REAL for single-precision flavors
 DOUBLE PRECISION for double-precision flavors.
 This output array contains the gap between eigenvalues whose eigenvectors could not be orthogonalized. The info/m output values in this array correspond to the $\text{info}/(m+1)$ clusters indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the i -th cluster may be as high as $(O(n) * \text{macheps}) / \text{gap}(i)$.

info (global) INTEGER.
 If $\text{info} = 0$, the execution is successful.
 If $\text{info} < 0$: If the i -th argument is an array and the j -entry had an illegal value, then $\text{info} = -(i*100+j)$,
 If the i -th argument is a scalar and had an illegal value, then $\text{info} = -i$.
 If $\text{info} < 0$: if $\text{info} = -i$, the i -th argument had an illegal value.
 If $\text{info} > 0$: if $\text{mod}(\text{info}, m+1) = i$, then i eigenvectors failed to converge in MAXIT iterations. Their indices are stored in the array *ifail*. If $\text{info}/(m+1) = i$, then eigenvectors corresponding to i clusters of eigenvalues could not be orthogonalized due to insufficient workspace. The indices of the clusters are stored in the array *iclustr*.

Nonsymmetric Eigenvalue Problems

This section describes ScaLAPACK routines for solving nonsymmetric eigenvalue problems, computing the Schur factorization of general matrices, as well as performing a number of related computational tasks.

To solve a nonsymmetric eigenvalue problem with ScaLAPACK, you usually need to reduce the matrix to the upper Hessenberg form and then solve the eigenvalue problem with the Hessenberg matrix obtained.

Table 6-5 lists ScaLAPACK routines for reducing the matrix to the upper Hessenberg form by an orthogonal (or unitary) similarity transformation $A = QHQ^H$, as well as routines for solving eigenproblems with Hessenberg matrices, and multiplying the matrix after reduction.

Table 6-5 Computational Routines for Solving Nonsymmetric Eigenproblems

Operation performed	General matrix	Orthogonal/Unitary Hessenberg matrix
Reduce to Hessenberg form $A = QHQ^H$	<code>p?gehrd</code>	
Multiply the matrix after reduction		<code>p?ormhr/p?unmhr</code>
Find eigenvalues and Schur factorization		<code>p?lahqr</code>

p?gehrd

Reduces a general matrix to upper Hessenberg form.

Syntax

```
call psgehrd( n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
call pdgehrd( n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
call pcgehrd( n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
call pzgehrd( n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine reduces a real/complex general distributed matrix sub (A) to upper Hessenberg form H by an orthogonal or unitary similarity transformation

$$Q'^* \text{sub}(A) * Q = H,$$

where $\text{sub}(A) = A(\text{ia}+n-1:\text{ia}+n-1, \text{ja}+n-1:\text{ja}+n-1)$.

Input Parameters

<i>n</i>	(global) INTEGER. The order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
<i>ilo, ihi</i>	(global) INTEGER. It is assumed that $\text{sub}(A)$ is already upper triangular in rows $\text{ia}:\text{ia}+\text{ilo}-2$ and $\text{ia}+\text{ihi}:\text{ia}+n-1$ and columns $\text{ja}:\text{ja}+\text{ilo}-2$ and $\text{ja}+\text{ihi}:\text{ja}+n-1$. (See <i>Application Notes</i> below). If $n > 0$, $1 \leq \text{ilo} \leq \text{ihi} \leq n$; otherwise set $\text{ilo} = 1$, $\text{ihi} = n$.
<i>a</i>	(local) REAL for psgehrd DOUBLE PRECISION for pdgehrd COMPLEX for pcgehrd DOUBLE COMPLEX for pzgehrd. Pointer into the local memory to an array of dimension $(\text{lld\_a}, \text{LOCc}(\text{ja}+n-1))$. On entry, this array contains the local pieces of the n -by- n general distributed matrix $\text{sub}(A)$ to be reduced.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (dlen_) . The array descriptor for the distributed matrix <i>A</i> .
<i>work</i>	(local) REAL for psgehrd DOUBLE PRECISION for pdgehrd COMPLEX for pcgehrd DOUBLE COMPLEX for pzgehrd. Workspace array of dimension <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, dimension of the array <i>work</i> . <i>lwork</i> is local input and must be at least $\text{lwork} \geq \text{NB} * \text{NB} + \text{NB} * \max(\text{ihlp}+1, \text{ihlp}+\text{inlq})$ where $\text{NB} = \text{mb\_a} = \text{nb\_a}$, $\text{iroffa} = \text{mod}(\text{ia}-1, \text{NB})$,

```

icoffa = mod(ja-1, NB),
ioff = mod(ia+ilo-2, NB), iarow = indxg2p(ia,
NB, MYROW, rsrc_a, NPROW), ihip =
numroc(ihi+ioffa, NB, MYROW, iarow, NPROW),
ilrow = indxg2p(ia+ilo-1, NB, MYROW, rsrc_a,
NPROW),
ihlp = numroc(ihi-ilo+ioff+1, NB, MYROW, ilrow,
NPROW),
ilcol = indxg2p(ja+ilo-1, NB, MYCOL, csrc_a,
NPCOL),
inlq = numroc(n-ilo+ioff+1, NB, MYCOL, ilcol,
NPCOL),
indxg2p and numroc are ScaLAPACK tool functions; MYROW,
MYCOL, NPROW and NPCOL can be determined by calling the
subroutine blacs_gridinfo.

```

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pserbla`.

Output Parameters

<code>a</code>	On exit, the upper triangle and the first subdiagonal of $\text{sub}(A)$ are overwritten with the upper Hessenberg matrix H , and the elements below the first subdiagonal, with the array <code>tau</code> , represent the orthogonal/unitary matrix Q as a product of elementary reflectors (see <i>Application Notes</i> below).
<code>tau</code>	(local). REAL for <code>psgehrd</code> DOUBLE PRECISION for <code>pdgehrd</code> COMPLEX for <code>pcgehrd</code> DOUBLE COMPLEX for <code>pzgehrd</code> . Array, DIMENSION at least $\max(ja+n-2)$. The scalar factors of the elementary reflectors (see <i>Application Notes</i> below). Elements <code>ja:ja+ilo-2</code> and <code>ja+ihi:ja+n-2</code> of <code>tau</code> are set to zero. <code>tau</code> is tied to the distributed matrix A .

work(1) On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info (global) INTEGER.
 = 0: the execution is successful.
 < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix Q is represented as a product of ($ihi-ilo$) elementary reflectors

$$Q = H(ilo)*H(ilo+1)*\dots*H(ihi-1).$$

Each $H(i)$ has the form

$$H(i) = I - \tau v v'$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i) = 0$, $v(i+1) = 1$ and $v(ihi+1:n) = 0$; $v(i+2:ihi)$ is stored on exit in $a(ia+ilo+i:ia+ihi-1, ja+ilo+i-2)$, and τ in $\tau(ja+ilo+i-2)$. The contents of $a(ia:ia+n-1, ja:ja+n-1)$ are illustrated by the following example, with $n = 7$, $ilo = 2$ and $ihi = 6$:

on entry

$$\begin{bmatrix} a & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & & a & a & a & a & a \\ & & & a & a & a & a \\ & & & & a & a & a \\ & & & & & a & a \\ & & & & & & a \end{bmatrix}$$

on exit

$$\begin{bmatrix} a & a & a & h & h & h & a \\ & a & h & h & h & h & a \\ & & h & h & h & h & h \\ & & & v2 & h & h & h \\ & & & & v2 & v3 & h \\ & & & & & v2 & v3 & v4 & h & h & h \\ & & & & & & & & & a \end{bmatrix}$$

where a denotes an element of the original matrix $\text{sub}(A)$, H denotes a modified element of the upper Hessenberg matrix H , and v_i denotes an element of the vector defining $H(ja+ilo+i-2)$.

p?ormhr

Multiplies a general matrix by the orthogonal transformation matrix from a reduction to Hessenberg form determined by p?gehrd.

Syntax

```
call psormhr(side, trans, m, n, ilo, ihi, a, ia, ja, desca, tau, c, ic, jc,
descc, work, lwork, info)
```

```
call pdormhr(side, trans, m, n, ilo, ihi, a, ia, ja, desca, tau, c, ic, jc,
descc, work, lwork, info)
```

Description

The routine overwrites the general real distributed m -by- n matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
$trans = 'T':$	$Q^T * \text{sub}(C)$	$\text{sub}(C) * Q^T$

where Q is a real orthogonal distributed matrix of order nq , with $nq = m$ if $side = 'L'$ and $nq = n$ if $side = 'R'$.

Q is defined as the product of $ihi-ilo$ elementary reflectors, as returned by p?gehrd.

$Q = H(ilo) H(ilo+1) \dots H(ihi-1)$.

Input Parameters

<i>side</i>	(global) CHARACTER ='L': Q or Q^T is applied from the left. ='R': Q or Q^T is applied from the right.
<i>trans</i>	(global) CHARACTER ='N', no transpose, Q is applied. ='T', transpose, Q^T is applied.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix sub (<i>C</i>) ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix sub (<i>C</i>) ($n \geq 0$).
<i>ilo, ihi</i>	(global) INTEGER. <i>ilo</i> and <i>ihi</i> must have the same values as in the previous call of <code>p?gehrd</code> . Q is equal to the unit matrix except for the distributed submatrix $Q(ia+ilo:ia+ihi-1, ia+ilo:ja+ihi-1)$. If <i>side</i> = 'L', $1 \leq ilo \leq ihi \leq \max(1, m)$; If <i>side</i> = 'R', $1 \leq ilo \leq ihi \leq \max(1, n)$; <i>ilo</i> and <i>ihi</i> are relative indexes.
<i>a</i>	(local) REAL for <code>psormhr</code> DOUBLE PRECISION for <code>pdormhr</code> Pointer into the local memory to an array of dimension (<i>lld_a</i> , <i>LOCc</i> (<i>ja</i> + <i>m</i> -1)) if <i>side</i> ='L', and (<i>lld_a</i> , <i>LOCc</i> (<i>ja</i> + <i>n</i> -1)) if <i>side</i> = 'R'. Contains the vectors which define the elementary reflectors, as returned by <code>p?gehrd</code> .
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) REAL for <code>psormhr</code>

	DOUBLE PRECISION for pdormhr Array, DIMENSION $LOCc(ja+m-2)$, if $side = 'L'$, and $LOCc(ja+n-2)$ if $side = 'R'$. This array contains the scalar factors $\tau(j)$ of the elementary reflectors $H(j)$ as returned by p?gehrd. τ is tied to the distributed matrix A .
c	(local) REAL for psormhr DOUBLE PRECISION for pdormhr Pointer into the local memory to an array of dimension ($lld\_c, LOCc(jc+n-1)$). Contains the local pieces of the distributed matrix sub(c).
ic, jc	(global) INTEGER. The row and column indices in the global array c indicating the first row and the first column of the submatrix c , respectively.
$descc$	(global and local) INTEGER array, dimension ($dlen\_$). The array descriptor for the distributed matrix c .
$work$	(local) REAL for psormhr DOUBLE PRECISION for pdormhr Workspace array with dimension $lwork$.
$lwork$	(local or global) INTEGER. The dimension of the array $work$. $lwork$ must be at least $iaa = ia + ilo; jaa =$ $ja + ilo - 1;$ If $side = 'L'$, $mi = ihi - ilo; ni = n; icc = ic + ilo; jcc = jc;$ $lwork \geq \max((nb\_a * (nb\_a - 1)) / 2, (nqc0 + mpc0) * nb\_a$ $+ nb\_a * nb\_a$ else if $side = 'R'$, $mi = m; ni = ihi - ilo; icc = ic; jcc = jc + ilo;$ $lwork \geq \max((nb\_a * (nb\_a - 1)) / 2,$ $(nqc0 + \max(npa0 + \text{numroc}(\text{numroc}(ni + icoffc, nb\_a,$ $0, 0, NPCOL), nb\_a, 0, 0, lcmq), mpc0)) * nb\_a) +$ $nb\_a * nb\_a$ end if

where $lcmq = lcm/NPCOL$ with $lcm = ilcm(NPROW, NPCOL)$,
 $iroffa = \text{mod}(iaa-1, mb\_a)$,
 $icoffa = \text{mod}(jaa-1, nb\_a)$,
 $iarow = \text{indxg2p}(iaa, mb\_a, MYROW, rsrc\_a, NPROW)$,
 $npa0 = \text{numroc}(ni+iroffa, mb\_a, MYROW, iarow, NPROW)$,
 $iroffc = \text{mod}(icc-1, mb\_c)$, $icoffc = \text{mod}(jcc-1, nb\_c)$,
 $icrow = \text{indxg2p}(icc, mb\_c, MYROW, rsrc\_c, NPROW)$,
 $iccol = \text{indxg2p}(jcc, nb\_c, MYCOL, csrc\_c, NPCOL)$,
 $mpc0 = \text{numroc}(mi+iroffc, mb\_c, MYROW, icrow, NPROW)$,
 $nqc0 = \text{numroc}(ni+icoffc, nb\_c, MYCOL, iccol, NPCOL)$,
 $ilcm, \text{indxg2p}$ and numroc are ScaLAPACK tool functions;
 $MYROW, MYCOL, NPROW$ and $NPCOL$ can be determined by
calling the subroutine `blacs_gridinfo`.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `p_xerbla`.

Output Parameters

c	$\text{sub}(c)$ is overwritten by $Q*\text{sub}(c)$, or $Q'*\text{sub}(c)$, or $\text{sub}(c)*Q'$, or $\text{sub}(c)*Q$.
$work(1)$	On exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance.
$info$	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -entry had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

p?unmhr

Multiplies a general matrix by the unitary transformation matrix from a reduction to Hessenberg form determined by p?gehrd.

Syntax

```
call pcunmhr(side, trans, m, n, ilo, ihi, a, ia, ja, desca, tau, c, ic, jc,
descc, work, lwork, info)

call pzunmhr(side, trans, m, n, ilo, ihi, a, ia, ja, desca, tau, c, ic, jc,
descc, work, lwork, info)
```

Description

The routine overwrites the general complex distributed m -by- n matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
$trans = 'H':$	$Q^H * \text{sub}(C)$	$\text{sub}(C) * Q^H$

where Q is a complex unitary distributed matrix of order nq , with $nq = m$ if $side = 'L'$ and $nq = n$ if $side = 'R'$.

Q is defined as the product of $ihi-ilo$ elementary reflectors, as returned by p?gehrd.

$Q = H(ilo) H(ilo+1) \dots H(ihi-1)$.

Input Parameters

$side$	(global) CHARACTER = 'L': Q or Q^H is applied from the left. = 'R': Q or Q^H is applied from the right.
$trans$	(global) CHARACTER = 'N', no transpose, Q is applied. = 'C', conjugate transpose, Q^H is applied.
m	(global) INTEGER. The number of rows in the distributed submatrix $\text{sub}(C)$ ($m \geq 0$).

<i>n</i>	(global) INTEGER. The number of columns in the distributed submatrix sub (<i>c</i>) ($n \geq 0$).
<i>ilo, ihi</i>	(global) INTEGER These must be the same parameters <i>ilo</i> and <i>ihi</i> , respectively, as supplied to p?gehrd. <i>Q</i> is equal to the unit matrix except in the distributed submatrix <i>Q</i> (<i>ia+ilo:ia+ihi-1, ia+ilo:ja+ihi-1</i>). If <i>side</i> = 'L', then $1 \leq ilo \leq ihi \leq \max(1, m)$. If <i>side</i> = 'R', then $1 \leq ilo \leq ihi \leq \max(1, n)$ <i>ilo</i> and <i>ihi</i> are relative indexes.
<i>a</i>	(local) COMPLEX for pcunmhr DOUBLE COMPLEX for pzunmhr. Pointer into the local memory to an array of dimension (<i>lld_a</i> , <i>LOC c(ja+m-1)</i>) if <i>side</i> ='L', and (<i>lld_a</i> , <i>LOCc(ja+n-1)</i>) if <i>side</i> = 'R'. Contains the vectors which define the elementary reflectors, as returned by p?gehrd.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) COMPLEX for pcunmhr DOUBLE COMPLEX for pzunmhr. Array, DIMENSION <i>LOCc(ja+m-2)</i> , if <i>side</i> = 'L', and <i>LOCc(ja+n-2)</i> if <i>side</i> = 'R'. This array contains the scalar factors <i>tau(j)</i> of the elementary reflectors <i>H(j)</i> as returned by p?gehrd. <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>c</i>	(local) COMPLEX for pcunmhr DOUBLE COMPLEX for pzunmhr. Pointer into the local memory to an array of dimension (<i>lld_c</i> , <i>LOCc(jc+n-1)</i>).

<i>ic, jc</i>	Contains the local pieces of the distributed matrix sub(<i>c</i>). (global) INTEGER. The row and column indices in the global array <i>c</i> indicating the first row and the first column of the submatrix <i>c</i> , respectively.
<i>descc</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>c</i> .
<i>work</i>	(local) COMPLEX for pcunmhr DOUBLE COMPLEX for pzunmhr. Workspace array with dimension <i>lwork</i> .
<i>lwork</i>	(local or global) The dimension of the array <i>work</i> . <i>lwork</i> must be at least <i>iaa</i> = <i>ia</i> + <i>ilo</i> ; <i>jaa</i> = <i>ja</i> + <i>ilo</i> - 1; If <i>side</i> = 'L', <i>mi</i> = <i>ihi</i> - <i>ilo</i> ; <i>ni</i> = <i>n</i> ; <i>icc</i> = <i>ic</i> + <i>ilo</i> ; <i>jcc</i> = <i>jc</i> ; <i>lwork</i> ≥ max((<i>nb_a</i> * (<i>nb_a</i> - 1)) / 2, (<i>nqc0</i> + <i>mpc0</i>) * <i>nb_a</i>) + <i>nb_a</i> * <i>nb_a</i> else if <i>side</i> = 'R', <i>mi</i> = <i>m</i> ; <i>ni</i> = <i>ihi</i> - <i>ilo</i> ; <i>icc</i> = <i>ic</i> ; <i>jcc</i> = <i>jc</i> + <i>ilo</i> ; <i>lwork</i> ≥ max((<i>nb_a</i> * (<i>nb_a</i> - 1)) / 2, (<i>nqc0</i> + max(<i>npa0</i> + numroc(numroc(<i>ni</i> + <i>icoffc</i> , <i>nb_a</i> , 0, 0, NPCOL), <i>nb_a</i> , 0, 0, <i>lcmq</i>), <i>mpc0</i>)) * <i>nb_a</i>) + <i>nb_a</i> * <i>nb_a</i> end if where <i>lcmq</i> = <i>lcm</i> / NPCOL with <i>lcm</i> = <i>ilcm</i> (NPROW, NPCOL), <i>iroffa</i> = mod(<i>iaa</i> - 1, <i>mb_a</i>), <i>icoffa</i> = mod(<i>jaa</i> - 1, <i>nb_a</i>), <i>iarow</i> = indxg2p(<i>iaa</i> , <i>mb_a</i> , MYROW, <i>rsrc_a</i> , NPROW), <i>npa0</i> = numroc(<i>ni</i> + <i>iroffa</i> , <i>mb_a</i> , MYROW, <i>iarow</i> , NPROW), <i>iroffc</i> = mod(<i>icc</i> - 1, <i>mb_c</i>), <i>icoffc</i> = mod(<i>jcc</i> - 1, <i>nb_c</i>), <i>icrow</i> = indxg2p(<i>icc</i> , <i>mb_c</i> , MYROW, <i>rsrc_c</i> , NPROW), <i>iccol</i> = indxg2p(<i>jcc</i> , <i>nb_c</i> , MYCOL, <i>csrc_c</i> , NPCOL), <i>mpc0</i> = numroc(<i>mi</i> + <i>iroffc</i> , <i>mb_c</i> , MYROW, <i>icrow</i> , NPROW),

`nqc0 = numroc(ni+icoffc, nb_c, MYCOL, iccol, NPCOL),`
`ilcm, indxg2p` and `numroc` are ScaLAPACK tool functions;
`MYROW, MYCOL, NPROW` and `NPCOL` can be determined by
calling the subroutine `blacs_gridinfo`.
If `lwork = -1`, then `lwork` is global input and a workspace
query is assumed; the routine only calculates the minimum
and optimal size for all work arrays. Each of these values
is returned in the first entry of the corresponding work array,
and no error message is issued by `pserbla`.

Output Parameters

<code>c</code>	<code>c</code> is overwritten by $Q^* \text{sub}(c)$ or $Q'^* \text{sub}(c)$ or $\text{sub}(c)^* Q'$ or $\text{sub}(c)^* Q$.
<code>work(1)</code>	On exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <code>info = - (i* 100+j)</code> , if the <i>i</i> -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

p?lahqr

Computes the Schur decomposition and/or eigenvalues of a matrix already in Hessenberg form.

Syntax

```
call pslahqr(wantt, wantz, n, ilo, ihi, a, desca, wr, wi, iloz, ihiz, z,
descz, work, lwork, iwork, ilwork, info)

call pdlahqr(wantt, wantz, n, ilo, ihi, a, desca, wr, wi, iloz, ihiz, z,
descz, work, lwork, iwork, ilwork, info)
```

Description

This is an auxiliary routine used to find the Schur decomposition and/or eigenvalues of a matrix already in Hessenberg form from columns *ilo* to *ihi*.

Input Parameters

<i>wantt</i>	(global) LOGICAL If <i>wantt</i> = .TRUE., the full Schur form <i>T</i> is required; If <i>wantt</i> = .FALSE., only eigenvalues are required.
<i>wantz</i>	(global) LOGICAL. If <i>wantz</i> = .TRUE., the matrix of Schur vectors <i>z</i> is required; If <i>wantz</i> = .FALSE., Schur vectors are not required.
<i>n</i>	(global) INTEGER. The order of the Hessenberg matrix <i>A</i> (and <i>z</i> if <i>wantz</i>). ($n \geq 0$).
<i>ilo, ihi</i>	(global) INTEGER. It is assumed that <i>A</i> is already upper quasi-triangular in rows and columns <i>ihi</i> +1: <i>n</i> , and that $A(ilo, ilo-1) = 0$ (unless <i>ilo</i> = 1). <i>p?lahqr</i> works primarily with the Hessenberg submatrix in rows and columns <i>ilo</i> to <i>ihi</i> , but applies transformations to all of <i>h</i> if <i>wantt</i> is .TRUE.. $1 \leq ilo \leq \max(1, ihi); ihi \leq n$.
<i>a</i>	(global) REAL for <i>pslahqr</i> DOUBLE PRECISION for <i>pdlahqr</i> Array, DIMENSION (<i>desca</i> (<i>lld_</i>), *). On entry, the upper Hessenberg matrix <i>A</i> .
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>iloz, ihiz</i>	(global) INTEGER. Specify the rows of <i>z</i> to which transformations must be applied if <i>wantz</i> is .TRUE.. $1 \leq iloz \leq ilo; ihi \leq ihiz \leq n$.
<i>z</i>	(global) REAL for <i>pslahqr</i> DOUBLE PRECISION for <i>pdlahqr</i>

	Array. If <i>wantz</i> is <code>.TRUE.</code> , on entry <i>z</i> must contain the current matrix <i>z</i> of transformations accumulated by <i>pdhseqr</i> . If <i>wantz</i> is <code>.FALSE.</code> , <i>z</i> is not referenced.
<i>descz</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>Z</i> .
<i>work</i>	(local) REAL for <i>pslahqr</i> DOUBLE PRECISION for <i>pdlahqr</i> Workspace array with dimension <i>lwork</i> .
<i>lwork</i>	(local) INTEGER. The dimension of <i>work</i> . <i>lwork</i> is assumed big enough so that $lwork \geq 3*n + \max(2*\max(descz(lld_), desca(lld_)) + 2*LOCq(n), 7*ceil(n/hbl)/lcm(NPROW, NPCOL))$. If <i>lwork</i> = -1, then <i>work</i> (1) gets set to the above number and the code returns immediately.
<i>iwork</i>	(global and local) INTEGER array of size <i>ilwork</i> .
<i>ilwork</i>	(local) INTEGER This holds some of the <i>iblk</i> integer arrays.

Output Parameters

<i>a</i>	On exit, if <i>wantt</i> is <code>.TRUE.</code> , <i>A</i> is upper quasi-triangular in rows and columns <i>ilo:ihi</i> , with any 2-by-2 or larger diagonal blocks not yet in standard form. If <i>wantt</i> is <code>.FALSE.</code> , the contents of <i>A</i> are unspecified on exit.
<i>work</i> (1)	On exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>wr</i> , <i>wi</i>	(global replicated output) REAL for <i>pslahqr</i> DOUBLE PRECISION for <i>pdlahqr</i> Arrays, DIMENSION(<i>n</i>) each. The real and imaginary parts, respectively, of the computed eigenvalues <i>ilo</i> to <i>ihi</i> are stored in the corresponding elements of <i>wr</i> and <i>wi</i> . If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of <i>wr</i> and <i>wi</i> , say the <i>i</i> -th and (<i>i</i> +1)-th, with <i>wi</i> (<i>i</i>) > 0 and <i>wi</i> (<i>i</i> +1) < 0. If <i>wantt</i> is <code>.TRUE.</code> , the eigenvalues are stored in the same

order as on the diagonal of the Schur form returned in *A*. *A* may be returned with larger diagonal blocks until the next release.

z On exit *z* has been updated; transformations are applied only to the submatrix *z*(*iloz:ihiz*, *ilo:ihi*).

info (global) INTEGER.
 = 0: the execution is successful.
 < 0: parameter number *-info* incorrect or inconsistent
 > 0: p?lahqr failed to compute all the eigenvalues *ilo* to *ihi* in a total of 30*(*ihi-ilo+1*) iterations; if *info* = *i*, elements *i+1:ihi* of *wr* and *wi* contain those eigenvalues which have been successfully computed.

Singular Value Decomposition

This section describes ScaLAPACK routines for computing the singular value decomposition (SVD) of a general *m*-by-*n* matrix *A* (see “Singular Value Decomposition” in LAPACK chapter).

To find the SVD of a general matrix *A*, this matrix is first reduced to a bidiagonal matrix *B* by a unitary (orthogonal) transformation, and then SVD of the bidiagonal matrix is computed. Note that the SVD of *B* is computed using the LAPACK routine ?bdsqr .

Table 6-6 lists ScaLAPACK computational routines for performing this decomposition.

Table 6-6 Computational Routines for Singular Value Decomposition (SVD)

Operation	General matrix	Orthogonal/unitary matrix
Reduce <i>A</i> to a bidiagonal matrix	p?gebrd	
Multiply matrix after reduction		p?ormbr/p?unmbr

p?gebrd

Reduces a general matrix to bidiagonal form.

Syntax

```
call psgebrd(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
call pdgebrd(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
call pcgebrd(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
call pzgebrd(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
```

Description

The routine reduces a real/complex general m -by- n distributed matrix $\text{sub}(A) = A(\text{ia}:\text{ia}+m-1, \text{ja}:\text{ja}+n-1)$ to upper or lower bidiagonal form B by an orthogonal/unitary transformation:

$$Q' * \text{sub}(A) * P = B.$$

If $m \geq n$, B is upper bidiagonal; if $m < n$, B is lower bidiagonal.

Input Parameters

m	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(A)$ ($m \geq 0$).
n	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
a	(local) REAL for psgebrd DOUBLE PRECISION for pdgebrd COMPLEX for pcgebrd DOUBLE COMPLEX for pzgebrd. Real pointer into the local memory to an array of dimension $(\text{lld\_a}, \text{LOCc}(\text{ja}+n-1))$. On entry, this array contains the distributed matrix $\text{sub}(A)$.
ia, ja	(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix A , respectively.
desca	(global and local) INTEGER array, dimension (dlen_) . The array descriptor for the distributed matrix A .
work	(local) REAL for psgebrd DOUBLE PRECISION for pdgebrd COMPLEX for pcgebrd DOUBLE COMPLEX for pzgebrd. Workspace array of dimension lwork .
lwork	(local or global) INTEGER, dimension of work , must be at least: $\text{lwork} \geq \text{nb} * (\text{mpa0} + \text{nqa0} + 1) + \text{nqa0}$

```

where  $nb = mb\_a = nb\_a$ ,
 $iroffa = \text{mod}(ia-1, nb)$ ,
 $icoffa = \text{mod}(ja-1, nb)$ ,
 $iarow = \text{indxg2p}(ia, nb, MYROW, rsrc\_a, NPROW)$ ,
 $iacol = \text{indxg2p}(ja, nb, MYCOL, csrc\_a, NPCOL)$ ,
 $mpa0 = \text{numroc}(m + iroffa, nb, MYROW, iarow, NPROW)$ ,
 $nqa0 = \text{numroc}(n + icoffa, nb, MYCOL, iacol, NPCOL)$ ,
 $\text{indxg2p}$  and  $\text{numroc}$  are ScaLAPACK tool functions;  $MYROW$ ,  $MYCOL$ ,  $NPROW$  and  $NPCOL$  can be determined by calling the subroutine blacs_gridinfo.

```

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pzerbla`.

Output Parameters

a

On exit, if $m \geq n$, the diagonal and the first superdiagonal of $\text{sub}(A)$ are overwritten with the upper bidiagonal matrix B ; the elements below the diagonal, with the array tauq , represent the orthogonal/unitary matrix Q as a product of elementary reflectors, and the elements above the first superdiagonal, with the array taup , represent the orthogonal matrix P as a product of elementary reflectors. If $m < n$, the diagonal and the first subdiagonal are overwritten with the lower bidiagonal matrix B ; the elements below the first subdiagonal, with the array tauq , represent the orthogonal/unitary matrix Q as a product of elementary reflectors, and the elements above the diagonal, with the array taup , represent the orthogonal matrix P as a product of elementary reflectors. See *Application Notes* below.

d

(local)

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors. Array,
 DIMENSION $LOCc(ja+\min(m,n)-1)$ if $m \geq n$;
 $LOCr(ia+\min(m,n)-1)$ otherwise. The distributed diagonal
 elements of the bidiagonal matrix B : $d(i) = a(i,i)$.
 d is tied to the distributed matrix A .

e (local)
 REAL for single-precision flavors
 DOUBLE PRECISION for double-precision flavors. Array,
 DIMENSION $LOCr(ia+\min(m,n)-1)$ if $m \geq n$;
 $LOCc(ja+\min(m,n)-2)$ otherwise. The distributed
 off-diagonal elements of the bidiagonal distributed matrix
 B :
 If $m \geq n$, $e(i) = a(i,i+1)$ for $i = 1, 2, \dots, n-1$; if m
 $< n$, $e(i) = a(i+1, i)$ for $i = 1, 2, \dots, m-1$. e is tied
 to the distributed matrix A .

tauq, taup (local)
 REAL for psgebrd
 DOUBLE PRECISION for pdgebrd
 COMPLEX for pcgebrd
 DOUBLE COMPLEX for pzgebrd.
 Arrays, DIMENSION $LOCc(ja+\min(m,n)-1)$ for *tauq* and
 $LOCr(ia+\min(m,n)-1)$ for *taup*. Contain the scalar factors
 of the elementary reflectors which represent the
 orthogonal/unitary matrices Q and P , respectively. *tauq* and
taup are tied to the distributed matrix A . See *Application*
Notes below.

work(1) On exit *work(1)* contains the minimum value of *lwork*
 required for optimum performance.

info (global) INTEGER.
 = 0: the execution is successful.
 < 0: if the i -th argument is an array and the j -entry had
 an illegal value, then $info = -(i*100+j)$, if the i -th
 argument is a scalar and had an illegal value, then $info =$
 $-i$.

Application Notes

The matrices Q and P are represented as products of elementary reflectors:

If $m \geq n$,

$$Q = H(1) * H(2) * \dots * H(n), \text{ and } P = G(1) * G(2) * \dots * G(n-1).$$

Each $H(i)$ and $G(i)$ has the form:

$$H(i) = I - \tau_{uq} * v * v' \text{ and } G(i) = I - \tau_{up} * u * u'$$

where τ_{uq} and τ_{up} are real/complex scalars, and v and u are real/complex vectors;

$v(1:i-1) = 0$, $v(i) = 1$, and $v(i+1:m)$ is stored on exit in $A(ia+i:ia+m-1, ja+i-1)$;

$u(1:i) = 0$, $u(i+1) = 1$, and $u(i+2:n)$ is stored on exit in $A(ia+i-1, ja+i+1:ja+n-1)$;

τ_{uq} is stored in $\tau_{uq}(ja+i-1)$ and τ_{up} in $\tau_{up}(ia+i-1)$.

If $m < n$,

$$Q = H(1) * H(2) * \dots * H(m-1), \text{ and } P = G(1) * G(2) * \dots * G(m)$$

Each $H(i)$ and $G(i)$ has the form:

$$H(i) = I - \tau_{uq} * v * v' \text{ and } G(i) = I - \tau_{up} * u * u'$$

here τ_{uq} and τ_{up} are real/complex scalars, and v and u are real/complex vectors;

$v(1:i) = 0$, $v(i+1) = 1$, and $v(i+2:m)$ is stored on exit in $A(ia+i:ia+m-1, ja+i-1)$;

$u(1:i-1) = 0$, $u(i) = 1$, and $u(i+1:n)$ is stored on exit in $A(ia+i-1, ja+i+1:ja+n-1)$;

τ_{uq} is stored in $\tau_{uq}(ja+i-1)$ and τ_{up} in $\tau_{up}(ia+i-1)$.

The contents of sub(A) on exit are illustrated by the following examples:

$m = 6$ and $n = 5$ ($m > n$):

$$\begin{bmatrix} d & e & u1 & u1 & u1 \\ v1 & d & e & u2 & u2 \\ v1 & v2 & d & e & u3 \\ v1 & v2 & v3 & d & e \\ v1 & v2 & v3 & v4 & d \\ v1 & v2 & v3 & v4 & v5 \end{bmatrix}$$

$m = 5$ and $n = 6$ ($m < n$):

$$\begin{bmatrix} d & u1 & u1 & u1 & u1 & u1 \\ e & d & u2 & u2 & u2 & u2 \\ v1 & e & d & u3 & u3 & u3 \\ v1 & v2 & e & d & u4 & u4 \\ v1 & v2 & v3 & e & d & u5 \end{bmatrix}$$

where d and e denote diagonal and off-diagonal elements of B , v_i denotes an element of the vector defining $H(i)$, and u_i an element of the vector defining $G(i)$.

p?ormbr

Multiplies a general matrix by one of the orthogonal matrices from a reduction to bidiagonal form determined by p?gebrd.

Syntax

```
call psormbr(vect, side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc,
descc, work, lwork, info)
```

```
call pdormbr(vect, side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc,
descc, work, lwork, info)
```

Description

If $vect = 'Q'$, the routine overwrites the general real distributed m -by- n matrix $sub(C) = C(c:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q \ sub(C)$	$sub(C) \ Q$
$trans = 'T':$	$Q^T \ sub(C)$	$sub(C) \ Q^T$

If $vect = 'P'$, the routine overwrites $sub(C)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$P \ sub(C)$	$sub(C) \ P$
$trans = 'T':$	$P^T \ sub(C)$	$sub(C) \ P^T$

Here Q and P^T are the orthogonal distributed matrices determined by `p?gebrd` when reducing a real distributed matrix $A(ia:*, ja:*)$ to bidiagonal form: $A(ia:*, ja:*) = Q*B*P^T$. Q and P^T are defined as products of elementary reflectors $H(i)$ and $G(i)$ respectively.

Let $nq = m$ if $side = 'L'$ and $nq = n$ if $side = 'R'$. Thus nq is the order of the orthogonal matrix Q or P^T that is applied.

If $vect = 'Q'$, $A(ia:*, ja:*)$ is assumed to have been an nq -by- k matrix:

If $nq \geq k$, $Q = H(1) H(2) \dots H(k)$;

If $nq < k$, $Q = H(1) H(2) \dots H(nq-1)$.

If $vect = 'P'$, $A(ia:*, ja:*)$ is assumed to have been a k -by- nq matrix:

If $k < nq$, $P = G(1) G(2) \dots G(k)$;

If $k \geq nq$, $P = G(1) G(2) \dots G(nq-1)$.

Input Parameters

<i>vect</i>	(global) CHARACTER. If $vect = 'Q'$, then Q or Q^T is applied. If $vect = 'P'$, then P or P^T is applied.
<i>side</i>	(global) CHARACTER. If $side = 'L'$, then Q or Q^T , P or P^T is applied from the left. If $side = 'R'$, then Q or Q^T , P or P^T is applied from the right.
<i>trans</i>	(global) CHARACTER. If $trans = 'N'$, no transpose, Q or P is applied. If $trans = 'T'$, then Q^T or P^T is applied.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix sub (<i>C</i>).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix sub (<i>C</i>).
<i>k</i>	(global) INTEGER. If $vect = 'Q'$, the number of columns in the original distributed matrix reduced by <code>p?gebrd</code> ; If $vect = 'P'$, the number of rows in the original distributed matrix reduced by <code>p?gebrd</code> . Constraints: $k \geq 0$.

<i>a</i>	(local) REAL for psormbr DOUBLE PRECISION for pdormbr. Pointer into the local memory to an array of dimension (<i>lld_a</i>, <i>LOCc</i>(<i>ja</i>+min(<i>nq</i>,<i>k</i>)-1)) If <i>vect</i>='Q', and (<i>lld_a</i>, <i>LOCc</i>(<i>ja</i>+<i>nq</i>-1)) If <i>vect</i> = 'P'. <i>nq</i> = <i>m</i> if <i>side</i> = 'L', and <i>nq</i> = <i>n</i> otherwise. The vectors which define the elementary reflectors $H(i)$ and $G(i)$, whose products determine the matrices Q and P, as returned by p?gebrd. If <i>vect</i> = 'Q', <i>lld_a</i> ≥ max(1, <i>LOCr</i>(<i>ia</i>+<i>nq</i>-1)); If <i>vect</i> = 'P', <i>lld_a</i> ≥ max(1, <i>LOCr</i>(<i>ia</i>+min(<i>nq</i>, <i>k</i>)-1)).
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) REAL for psormbr DOUBLE PRECISION for pdormbr. Array, DIMENSION <i>LOCc</i>(<i>ja</i>+min(<i>nq</i>, <i>k</i>)-1), if <i>vect</i> = 'Q', and <i>LOCr</i>(<i>ia</i>+min(<i>nq</i>, <i>k</i>)-1), if <i>vect</i> = 'P'. <i>tau</i>(<i>i</i>) must contain the scalar factor of the elementary reflector $H(i)$ or $G(i)$, which determines Q or P, as returned by pdgebrd in its array argument <i>tauq</i> or <i>taup</i>. <i>tau</i> is tied to the distributed matrix <i>A</i>.
<i>c</i>	(local) REAL for psormbr DOUBLE PRECISION for pdormbr Pointer into the local memory to an array of dimension (<i>lld_a</i>, <i>LOCc</i> (<i>jc</i>+<i>n</i>-1)). Contains the local pieces of the distributed matrix sub (<i>c</i>).
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the global array <i>c</i> indicating the first row and the first column of the submatrix <i>C</i> , respectively.

<i>desc</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>c</i> .
<i>work</i>	(local) REAL for <i>psormbr</i> DOUBLE PRECISION for <i>pdormbr</i> . Workspace array of dimension <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, dimension of <i>work</i> , must be at least: If <i>side</i> = 'L' <i>nq</i> = <i>m</i> ; if ((<i>vect</i> = 'Q' and <i>nq</i> ≥ <i>k</i>) or (<i>vect</i> is not equal to 'Q' and <i>nq</i> > <i>k</i>)), <i>iaa</i> = <i>ia</i> ; <i>jaa</i> = <i>ja</i> ; <i>mi</i> = <i>m</i> ; <i>ni</i> = <i>n</i> ; <i>icc</i> = <i>ic</i> ; <i>jcc</i> = <i>jc</i> ; else <i>iaa</i> = <i>ia</i> +1; <i>jaa</i> = <i>ja</i> ; <i>mi</i> = <i>m</i> -1; <i>ni</i> = <i>n</i> ; <i>icc</i> = <i>ic</i> +1; <i>jcc</i> = <i>jc</i> ; end if else If <i>side</i> = 'R', <i>nq</i> = <i>n</i> ; if ((<i>vect</i> = 'Q' and <i>nq</i> ≥ <i>k</i>) or (<i>vect</i> is not equal to 'Q' and <i>nq</i> > <i>k</i>)), <i>iaa</i> = <i>ia</i> ; <i>jaa</i> = <i>ja</i> ; <i>mi</i> = <i>m</i> ; <i>ni</i> = <i>n</i> ; <i>icc</i> = <i>ic</i> ; <i>jcc</i> = <i>jc</i> ; else <i>iaa</i> = <i>ia</i> ; <i>jaa</i> = <i>ja</i> +1; <i>mi</i> = <i>m</i> ; <i>ni</i> = <i>n</i> -1; <i>icc</i> = <i>ic</i> ; <i>jcc</i> = <i>jc</i> +1; end if end if If <i>vect</i> = 'Q', If <i>side</i> = 'L', <i>lwork</i> ≥ max((<i>nb_a</i> *(<i>nb_a</i> -1))/2, (<i>nqc0</i> + <i>mpc0</i>)* <i>nb_a</i>) + <i>nb_a</i> * <i>nb_a</i> else if <i>side</i> = 'R', <i>lwork</i> ≥ max((<i>nb_a</i> *(<i>nb_a</i> -1))/2, (<i>nqc0</i> + max(<i>npa0</i> + numroc(numroc(<i>ni</i> + <i>icoffc</i> , <i>nb_a</i> , 0, 0, NPCOL), <i>nb_a</i> , 0, 0, <i>lcmq</i>), <i>mpc0</i>))* <i>nb_a</i>) + <i>nb_a</i> * <i>nb_a</i> end if else if <i>vect</i> is not equal to 'Q', if <i>side</i> = 'L',

```

lwork ≥ max((mb_a*(mb_a-1))/2, (mpc0 + max(mqa0 +
numroc(numroc(mi+iroffc, mb_a, 0, 0, NPROW),
mb_a, 0, 0, lcmp), nqc0))*mb_a) + mb_a*mb_a
else if side = 'R',
lwork ≥ max((mb_a*(mb_a-1))/2, (mpc0 + nqc0)*mb_a)
+ mb_a*mb_a
end if
end if
where lcmp = lcm/NPROW, lcmq = lcm/NPCOL, with
lcm = ilcm(NPROW, NPCOL),
iroffa = mod(iaa-1, mb_a),
icoffa = mod(jaa-1, nb_a),
iarow = indxg2p(iaa, mb_a, MYROW, rsrc_a, NPROW),
iacol = indxg2p(jaa, nb_a, MYCOL, csrc_a, NPCOL),
mqa0 = numroc(mi+icoffa, nb_a, MYCOL, iacol,
NPCOL),
npa0 = numroc(ni+iroffa, mb_a, MYROW, iarow,
NPROW),
iroffc = mod(icc-1, mb_c),
icoffc = mod(jcc-1, nb_c),
icrow = indxg2p(icc, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jcc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(mi+iroffc, mb_c, MYROW, icrow,
NPROW),
nqc0 = numroc(ni+icoffc, nb_c, MYCOL, iccol,
NPCOL),
indxg2p and numroc are ScaLAPACK tool functions; MYROW,
MYCOL, NPROW and NPCOL can be determined by calling the
subroutine blacs_gridinfo.

```

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by p_xerbla.

Output Parameters

<code>c</code>	On exit, if <code>vect='Q'</code> , <code>sub(c)</code> is overwritten by $Q*\text{sub}(c)$, or $Q'^*\text{sub}(c)$, or $\text{sub}(c)*Q'$, or $\text{sub}(c)*Q$; if <code>vect='P'</code> , <code>sub(c)</code> is overwritten by $P*\text{sub}(c)$, or $P'^*\text{sub}(c)$, or $\text{sub}(c)*P$, or $\text{sub}(c)*P'$.
<code>work(1)</code>	On exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <code>info</code> = - (<i>i</i> * 100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <code>info</code> = - <i>i</i> .

p?unmbr

Multiplies a general matrix by one of the unitary transformation matrices from a reduction to bidiagonal form determined by p?gebrd.

Syntax

```
call pcunmbr(vect, side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc,
descc, work, lwork, info)

call pzunmbr(vect, side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc,
descc, work, lwork, info)
```

Description

If `vect = 'Q'`, the routine overwrites the general complex distributed *m*-by-*n* matrix `sub(c)` = `C(ic:ic+m-1, jc:jc+n-1)` with

	<code>side = 'L'</code>	<code>side = 'R'</code>
<code>trans = 'N':</code>	$Q*\text{sub}(c)$	$\text{sub}(c)*Q$
<code>trans = 'C':</code>	$Q^H*\text{sub}(c)$	$\text{sub}(c)*Q^H$

If `vect = 'P'`, the routine overwrites `sub(c)` with

<code>side = 'L'</code>	<code>side = 'R'</code>

$trans = 'N':$	$P * sub(C)$	$sub(C) * P$
$trans = 'C':$	$P^H * sub(C)$	$sub(C) * P^H$

Here Q and P^H are the unitary distributed matrices determined by `p?gebrd` when reducing a complex distributed matrix $A(ia:*, ja:*)$ to bidiagonal form: $A(ia:*, ja:*) = Q * B * P^H$.

Q and P^H are defined as products of elementary reflectors $H(i)$ and $G(i)$ respectively.

Let $nq = m$ if $side = 'L'$ and $nq = n$ if $side = 'R'$. Thus nq is the order of the unitary matrix Q or P^H that is applied.

If $vect = 'Q'$, $A(ia:*, ja:*)$ is assumed to have been an nq -by- k matrix:

If $nq \geq k$, $Q = H(1) H(2) \dots H(k)$;

If $nq < k$, $Q = H(1) H(2) \dots H(nq-1)$.

If $vect = 'P'$, $A(ia:*, ja:*)$ is assumed to have been a k -by- nq matrix:

If $k < nq$, $P = G(1) G(2) \dots G(k)$;

If $k \geq nq$, $P = G(1) G(2) \dots G(nq-1)$.

Input Parameters

<i>vect</i>	(global) CHARACTER. If $vect = 'Q'$, then Q or Q^H is applied. If $vect = 'P'$, then P or P^H is applied.
<i>side</i>	(global) CHARACTER. If $side = 'L'$, then Q or Q^H , P or P^H is applied from the left. If $side = 'R'$, then Q or Q^H , P or P^H is applied from the right.
<i>trans</i>	(global) CHARACTER. If $trans = 'N'$, no transpose, Q or P is applied. If $trans = 'C'$, conjugate transpose, Q^H or P^H is applied.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix $sub(C)$ $m \geq 0$.
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix $sub(C)$ $n \geq 0$.
<i>k</i>	(global) INTEGER.

	If <code>vect = 'Q'</code>, the number of columns in the original distributed matrix reduced by <code>p?gebrd</code>; If <code>vect = 'P'</code>, the number of rows in the original distributed matrix reduced by <code>p?gebrd</code>. Constraints: $k \geq 0$.
<i>a</i>	(local) COMPLEX for <code>psormbr</code> DOUBLE COMPLEX for <code>pdormbr</code>. Pointer into the local memory to an array of dimension $(lld\_a, LOCc(ja+\min(nq, k)-1))$ if <code>vect='Q'</code>, and $(lld\_a, LOCc(ja+nq-1))$ if <code>vect = 'P'</code>. $nq = m$ if <code>side = 'L'</code>, and $nq = n$ otherwise. The vectors which define the elementary reflectors $H(i)$ and $G(i)$, whose products determine the matrices Q and P, as returned by <code>p?gebrd</code>. If <code>vect = 'Q'</code>, $lld\_a \geq \max(1, LOCr(ia+nq-1))$; If <code>vect = 'P'</code>, $lld\_a \geq \max(1, LOCr(ia+\min(nq, k)-1))$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i>, respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i>.
<i>tau</i>	(local) COMPLEX for <code>pcunmbr</code> DOUBLE COMPLEX for <code>pzunmbr</code>. Array, DIMENSION $LOCc(ja+\min(nq, k)-1)$, if <code>vect = 'Q'</code>, and $LOCr(ia+\min(nq, k)-1)$, if <code>vect = 'P'</code>. <i>tau</i>(<i>i</i>) must contain the scalar factor of the elementary reflector $H(i)$ or $G(i)$, which determines Q or P, as returned by <code>p?gebrd</code> in its array argument <i>tauq</i> or <i>taup</i>. <i>tau</i> is tied to the distributed matrix <i>A</i>.
<i>c</i>	(local) COMPLEX for <code>pcunmbr</code> DOUBLE COMPLEX for <code>pzunmbr</code> Pointer into the local memory to an array of dimension $(lld\_a, LOCc(jc+n-1))$.

<i>ic, jc</i>	Contains the local pieces of the distributed matrix sub (<i>c</i>). (global) INTEGER. The row and column indices in the global array <i>c</i> indicating the first row and the first column of the submatrix <i>c</i> , respectively.
<i>desc</i>	(global and local) INTEGER array, dimension (<i>dlen</i>). The array descriptor for the distributed matrix <i>c</i> .
<i>work</i>	(local) COMPLEX for pcunmbr DOUBLE COMPLEX for pzunmbr. Workspace array of dimension <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, dimension of <i>work</i> , must be at least: If <i>side</i> = 'L' <i>nq</i> = <i>m</i> ; if ((<i>vect</i> = 'Q' and <i>nq</i> ≥ <i>k</i>) or (<i>vect</i> is not equal to 'Q' and <i>nq</i> > <i>k</i>)), <i>iaa</i> = <i>ia</i> ; <i>jaa</i> = <i>ja</i> ; <i>mi</i> = <i>m</i> ; <i>ni</i> = <i>n</i> ; <i>icc</i> = <i>ic</i> ; <i>jcc</i> = <i>jc</i> ; else <i>iaa</i> = <i>ia</i> +1; <i>jaa</i> = <i>ja</i> ; <i>mi</i> = <i>m</i> -1; <i>ni</i> = <i>n</i> ; <i>icc</i> = <i>ic</i> +1; <i>jcc</i> = <i>jc</i> ; end if else If <i>side</i> = 'R', <i>nq</i> = <i>n</i> ; if ((<i>vect</i> = 'Q' and <i>nq</i> ≥ <i>k</i>) or (<i>vect</i> is not equal to 'Q' and <i>nq</i> ≥ <i>k</i>)), <i>iaa</i> = <i>ia</i> ; <i>jaa</i> = <i>ja</i> ; <i>mi</i> = <i>m</i> ; <i>ni</i> = <i>n</i> ; <i>icc</i> = <i>ic</i> ; <i>jcc</i> = <i>jc</i> ; else <i>iaa</i> = <i>ia</i> ; <i>jaa</i> = <i>ja</i> +1; <i>mi</i> = <i>m</i> ; <i>ni</i> = <i>n</i> -1; <i>icc</i> = <i>ic</i> ; <i>jcc</i> = <i>jc</i> +1; end if end if If <i>vect</i> = 'Q', If <i>side</i> = 'L', <i>lwork</i> ≥ max((<i>nb_a</i> *(<i>nb_a</i> -1))/2, (<i>nqc</i> 0+ <i>mpc</i> 0)* <i>nb_a</i>) + <i>nb_a</i> * <i>nb_a</i> else if <i>side</i> = 'R',

```

lwork ≥ max((nb_a*(nb_a-1))/2, (nqc0 +
max(npa0+numroc(numroc(ni+icoffc, nb_a, 0, 0,
NPCOL), nb_a, 0, 0, lcmq), mpc0))*nb_a) +
nb_a*nb_a
end if
else if vect is not equal to 'Q',
if side = 'L',
lwork ≥ max((mb_a*(mb_a-1))/2, (mpc0 +
max(mqa0+numroc(numroc(mi+iroffc, mb_a, 0, 0,
NPROW), mb_a, 0, 0, lcmp), nqc0))*mb_a) +
mb_a*mb_a
else if side = 'R',
lwork ≥ max((mb_a*(mb_a-1))/2, (mpc0 +
nqc0)*mb_a) + mb_a*mb_a
end if
end if
where lcmp = lcm/NPROW, lcmq = lcm/NPCOL, with lcm
= ilcm(NPROW, NPCOL),
iroffa = mod(iaa-1, mb_a),
icoffa = mod(jaa-1, nb_a),
iarow = indxg2p(iaa, mb_a, MYROW, rsrc_a, NPROW),
iacol = indxg2p(jaa, nb_a, MYCOL, csrc_a, NPCOL),
mqa0 = numroc(mi+icoffa, nb_a, MYCOL, iacol, NPCOL),
npa0 = numroc(ni+iroffa, mb_a, MYROW, iarow, NPROW),
iroffc = mod(icc-1, mb_c),
icoffc = mod(jcc-1, nb_c),
icrow = indxg2p(icc, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jcc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(mi+iroffc, mb_c, MYROW, icrow,
NPROW),
nqc0 = numroc(ni+icoffc, nb_c, MYCOL, iccol,
NPCOL),
indxg2p and numroc are ScaLAPACK tool functions; MYROW,
MYCOL, NPROW and NPCOL can be determined by calling the
subroutine blacs_gridinfo.

```

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

c	On exit, if $vect='Q'$, $sub(c)$ is overwritten by $Q*sub(C)$, or $Q'*sub(C)$, or $sub(C)*Q'$, or $sub(C)*Q$; if $vect='P'$, $sub(c)$ is overwritten by $P*sub(C)$, or $P'*sub(C)$, or $sub(C)*P$, or $sub(C)*P'$.
$work(1)$	On exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance.
$info$	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -entry had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Generalized Symmetric-Definite Eigen Problems

This section describes ScaLAPACK routines that allow you to reduce the *generalized symmetric-definite eigenvalue problems* (see [Generalized Symmetric-Definite Eigenvalue Problems](#) in LAPACK chapters) to standard symmetric eigenvalue problem $C_Y = \lambda_Y$, which you can solve by calling ScaLAPACK routines described earlier in this chapter (see [Symmetric Eigenproblems](#)).

[Table 6-7](#) lists these routines.

Table 6-7 Computational Routines for Reducing Generalized Eigenproblems to Standard Problems

Operation	Real symmetric matrices	Complex Hermitian matrices
Reduce to standard problems	p?sygst	p?hegst

p?sygst

Reduces a real symmetric-definite generalized eigenvalue problem to the standard form.

Syntax

```
call pssygst( ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, scale,
info )

call pdsygst( ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, scale,
info )
```

Description

This routine reduces real symmetric-definite generalized eigenproblems to the standard form.

In the following $\text{sub}(A)$ denotes $A(ia:ia+n-1, ja:ja+n-1)$ and $\text{sub}(B)$ denotes $B(ib:ib+n-1, jb:jb+n-1)$.

If $ibtype = 1$, the problem is

$$\text{sub}(A) * x = \lambda * \text{sub}(B) * x,$$

and $\text{sub}(A)$ is overwritten by $\text{inv}(U^T) * \text{sub}(A) * \text{inv}(U)$, or $\text{inv}(L) * \text{sub}(A) * \text{inv}(L^T)$.

If $ibtype = 2$ or 3 , the problem is

$$\text{sub}(A) * \text{sub}(B) * x = \lambda * x, \text{ or } \text{sub}(B) * \text{sub}(A) * x = \lambda * x,$$

and $\text{sub}(A)$ is overwritten by $U * \text{sub}(A) * U^T$, or $L^T * \text{sub}(A) * L$.

$\text{sub}(B)$ must have been previously factorized as $U^T * U$ or $L * L^T$ by p?potrf.

Input Parameters

<i>ibtype</i>	(global) INTEGER. Must be 1 or 2 or 3. If $itype = 1$, compute $\text{inv}(U^T) * \text{sub}(A) * \text{inv}(U)$, or $\text{inv}(L) * \text{sub}(A) * \text{inv}(L^T)$; If $itype = 2$ or 3 , compute $U * \text{sub}(A) * U^T$, or $L^T * \text{sub}(A) * L$.
<i>uplo</i>	(global) CHARACTER. Must be 'U' or 'L'. If $uplo = 'U'$, the upper triangle of $\text{sub}(A)$ is stored and $\text{sub}(B)$ is factored as $U^T * U$.

	If $uplo = 'L'$, the lower triangle of $sub(A)$ is stored and $sub(B)$ is factored as $L * L^T$.
n	(global) INTEGER. The order of the matrices $sub(A)$ and $sub(B)$ ($n \geq 0$).
a	(local) REAL for pssygst DOUBLE PRECISION for pdsygst. Pointer into the local memory to an array of dimension $(lld\_a, LOCC(ja+n-1))$. On entry, the array contains the local pieces of the n-by-n symmetric distributed matrix $sub(A)$. If $uplo = 'U'$, the leading n-by-n upper triangular part of $sub(A)$ contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If $uplo = 'L'$, the leading n-by-n lower triangular part of $sub(A)$ contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.
ia, ja	(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix A, respectively.
$desca$	(global and local) INTEGER array, dimension $(dlen\_)$. The array descriptor for the distributed matrix A.
b	(local) REAL for pssygst DOUBLE PRECISION for pdsygst. Pointer into the local memory to an array of dimension $(lld\_b, LOCC(jb+n-1))$. On entry, the array contains the local pieces of the triangular factor from the Cholesky factorization of $sub(B)$ as returned by p?potrf.
ib, jb	(global) INTEGER. The row and column indices in the global array b indicating the first row and the first column of the submatrix B, respectively.
$descb$	(global and local) INTEGER array, dimension $(dlen\_)$. The array descriptor for the distributed matrix B.

Output Parameters

<i>a</i>	On exit, if <i>info</i> = 0, the transformed matrix, stored in the same format as sub(<i>A</i>).
<i>scale</i>	(global) REAL for pssygst DOUBLE PRECISION for pdsygst. Amount by which the eigenvalues should be scaled to compensate for the scaling performed in this routine. At present, <i>scale</i> is always returned as 1.0, it is returned here to allow for future enhancement.
<i>info</i>	(global) INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> < 0, if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = -(<i>i</i> *100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

p?hegst

Reduces a Hermitian-definite generalized eigenvalue problem to the standard form.

Syntax

```
call pchegst( ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, scale,
info)

call pzhegst( ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, scale,
info)
```

Description

This routine reduces complex Hermitian-definite generalized eigenproblems to the standard form.

In the following sub(*A*) denotes *A*(*ia:ia*+*n*-1, *ja:ja*+*n*-1) and sub(*B*) denotes *B*(*ib:ib*+*n*-1, *jb:jb*+*n*-1).

If *ibtype* = 1, the problem is

$$\text{sub}(A) * x = \lambda * \text{sub}(B) * x,$$

and $\text{sub}(A)$ is overwritten by $\text{inv}(U^H) * \text{sub}(A) * \text{inv}(U)$, or $\text{inv}(L) * \text{sub}(A) * \text{inv}(L^H)$.

If $\text{ibtype} = 2$ or 3 , the problem is

$\text{sub}(A) * \text{sub}(B) * x = \lambda * x$, or $\text{sub}(B) * \text{sub}(A) * x = \lambda * x$,

and $\text{sub}(A)$ is overwritten by $U * \text{sub}(A) * U^H$, or $L^H * \text{sub}(A) * L$.

$\text{sub}(B)$ must have been previously factorized as $U^H * U$ or $L * L^H$ by `p?potrf`.

Input Parameters

<i>ibtype</i>	(global) INTEGER. Must be 1 or 2 or 3. If $\text{itype} = 1$, compute $\text{inv}(U^H) * \text{sub}(A) * \text{inv}(U)$, or $\text{inv}(L) * \text{sub}(A) * \text{inv}(L^H)$; If $\text{itype} = 2$ or 3 , compute $U * \text{sub}(A) * U^H$, or $L^H * \text{sub}(A) * L$.
<i>uplo</i>	(global) CHARACTER. Must be 'U' or 'L'. If $\text{uplo} = 'U'$, the upper triangle of $\text{sub}(A)$ is stored and $\text{sub}(B)$ is factored as $U^H * U$. If $\text{uplo} = 'L'$, the lower triangle of $\text{sub}(A)$ is stored and $\text{sub}(B)$ is factored as $L * L^H$.
<i>n</i>	(global) INTEGER. The order of the matrices $\text{sub}(A)$ and $\text{sub}(B)$ ($n \geq 0$).
<i>a</i>	(local) COMPLEX for <code>pchegst</code> DOUBLE COMPLEX for <code>pzhgst</code> . Pointer into the local memory to an array of dimension $(\text{lld\_a}, \text{LOC}(ja+n-1))$. On entry, the array contains the local pieces of the n -by- n Hermitian distributed matrix $\text{sub}(A)$. If $\text{uplo} = 'U'$, the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If $\text{uplo} = 'L'$, the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.

<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>b</i>	(local) COMPLEX for <i>pchegst</i> DOUBLE COMPLEX for <i>pzhegst</i> . Pointer into the local memory to an array of dimension (<i>lld_b</i> , <i>LOCc(jb+n-1)</i>). On entry, the array contains the local pieces of the triangular factor from the Cholesky factorization of sub (<i>B</i>) as returned by <i>p?potrf</i> .
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the global array <i>b</i> indicating the first row and the first column of the submatrix <i>B</i> , respectively.
<i>descb</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>B</i> .

Output Parameters

<i>a</i>	On exit, if <i>info</i> = 0, the transformed matrix, stored in the same format as sub(<i>A</i>).
<i>scale</i>	(global) REAL for <i>pchegst</i> DOUBLE PRECISION for <i>pzhegst</i> . Amount by which the eigenvalues should be scaled to compensate for the scaling performed in this routine. At present, <i>scale</i> is always returned as 1.0, it is returned here to allow for future enhancement.
<i>info</i>	(global) INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> < 0, if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = -(<i>i</i> 100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

Driver Routines

Table 6-8 lists ScaLAPACK driver routines available for solving systems of linear equations, linear least-squares problems, standard eigenvalue and singular value problems, and generalized symmetric definite eigenproblems.

Table 6-8 ScaLAPACK Driver Routines

Type of Problem	Matrix type, storage scheme	Driver
Linear equations	general (partial pivoting)	p?gesv (simple driver) p?gesvx (expert driver)
	general band (partial pivoting)	p?gbsv (simple driver)
	general band (no pivoting)	p?dbsv (simple driver)
	general tridiagonal (no pivoting)	p?dtsv (simple driver)
	symmetric/Hermitian positive-definite	p?posv (simple driver) p?posvx (expert driver)
	symmetric/Hermitian positive-definite, band	p?pbsv (simple driver)
	symmetric/Hermitian positive-definite, tridiagonal	p?ptsv (simple driver)
Linear least squares problem	general m -by- n	p?gels
Symmetric eigenvalue problem	symmetric/Hermitian	p?syev (simple driver) p?syevx / p?heevx (expert driver)
Singular value decomposition	general m -by- n	p?gesvd
Generalized symmetric definite eigenvalue problem	symmetric/Hermitian, one matrix also positive-definite	p?sygvx / p?hegvx (expert driver)

p?gesv

Computes the solution to the system of linear equations with a square distributed matrix and multiple right-hand sides.

Syntax

```
call psgesv(n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
call pdgesv(n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
call pcgesv(n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
call pzgesv(n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
```

Description

The routine `p?gesv` computes the solution to a real or complex system of linear equations $\text{sub}(A) * X = \text{sub}(B)$, where $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$ is an n -by- n distributed matrix and X and $\text{sub}(B) = B(\text{ib}:\text{ib}+n-1, \text{jb}:\text{jb}+nrhs-1)$ are n -by- $nrhs$ distributed matrices.

The LU decomposition with partial pivoting and row interchanges is used to factor $\text{sub}(A)$ as $\text{sub}(A) = P * L * U$, where P is a permutation matrix, L is unit lower triangular, and U is upper triangular. L and U are stored in $\text{sub}(A)$. The factored form of $\text{sub}(A)$ is then used to solve the system of equations $\text{sub}(A) * X = \text{sub}(B)$.

Input Parameters

n	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).
$nrhs$	(global) INTEGER. The number of right hand sides, that is, the number of columns of the distributed submatrices B and X ($nrhs \geq 0$).
a, b	(local) REAL for <code>psgesv</code> DOUBLE PRECISION for <code>pdgesv</code> COMPLEX for <code>pcgesv</code> DOUBLE COMPLEX for <code>pzgesv</code> . Pointers into the local memory to arrays of local dimension $a(\text{lld\_a}, \text{LOCc}(\text{ja}+n-1))$ and $b(\text{lld\_b}, \text{LOCc}(\text{jb}+nrhs-1))$, respectively. On entry, the array a contains the local pieces of the n -by- n distributed matrix $\text{sub}(A)$ to be factored. On entry, the array b contains the right hand side distributed matrix $\text{sub}(B)$.
ia, ja	(global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of $\text{sub}(A)$, respectively.
$desca$	(global and local) INTEGER array, dimension (dlen_) . The array descriptor for the distributed matrix A .

<i>ib, jb</i>	(global) INTEGER. The row and column indices in the global array <i>B</i> indicating the first row and the first column of <i>sub(B)</i> , respectively.
<i>descb</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>B</i> .

Output Parameters

<i>a</i>	Overwritten by the factors <i>L</i> and <i>U</i> from the factorization $\text{sub}(A) = P^*L^*U$; the unit diagonal elements of <i>L</i> are not stored .
<i>b</i>	Overwritten by the solution distributed matrix <i>x</i> .
<i>ipiv</i>	(local) INTEGER array. The dimension of <i>ipiv</i> is $(\text{LOCr}(m\_a) + mb\_a)$. This array contains the pivoting information. The (local) row <i>i</i> of the matrix was interchanged with the (global) row <i>ipiv(i)</i> . This array is tied to the distributed matrix <i>A</i> .
<i>info</i>	(global) INTEGER. If <i>info</i> =0, the execution is successful. <i>info</i> < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <i>info</i> = $-(i*100+j)$; if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = $-i$. <i>info</i> > 0: If <i>info</i> = <i>k</i> , $U(ia+k-1, ja+k-1)$ is exactly zero. The factorization has been completed, but the factor <i>U</i> is exactly singular, so the solution could not be computed.

p?gesvx

Uses the LU factorization to compute the solution to the system of linear equations with a square matrix A and multiple right-hand sides, and provides error bounds on the solution.

Syntax

```
call psgesvx(fact, trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf,
            ipiv, equed, r, c, b, ib, jb, descb, x, ix, jx, descx, rcond, ferr, berr,
            work, lwork, iwork, liwork, info)
```

```
call pdgesvx(fact, trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf,
            ipiv, equed, r, c, b, ib, jb, descb, x, ix, jx, descx, rcond, ferr, berr,
            work, lwork, iwork, liwork, info)
```

```
call pcgesvx(fact, trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf,
            ipiv, equed, r, c, b, ib, jb, descb, x, ix, jx, descx, rcond, ferr, berr,
            work, lwork, rwork, lrwork, info)
```

```
call pzgesvx(fact, trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf,
            ipiv, equed, r, c, b, ib, jb, descb, x, ix, jx, descx, rcond, ferr, berr,
            work, lwork, rwork, lrwork, info)
```

Description

This routine uses the LU factorization to compute the solution to a real or complex system of linear equations $AX = B$, where A denotes the n -by- n submatrix $A(ia:ia+n-1, ja:ja+n-1)$, B denotes the n -by- $nrhs$ submatrix $B(ib:ib+n-1, jb:jb+nrhs-1)$ and X denotes the n -by- $nrhs$ submatrix $X(ix:ix+n-1, jx:jx+nrhs-1)$.

Error bounds on the solution and a condition estimate are also provided.

In the following description, af stands for the subarray $af(iaf:iaf+n-1, jaf:jaf+n-1)$.

The routine p?gesvx performs the following steps:

1. If $fact = 'E'$, real scaling factors R and C are computed to equilibrate the system:

$$trans = 'N': \text{diag}(R) * A * \text{diag}(C) * \text{diag}(C)^{-1} * X = \text{diag}(R) * B$$

$$trans = 'T': (\text{diag}(R) * A * \text{diag}(C))^T * \text{diag}(R)^{-1} * X = \text{diag}(C) * B$$

$$trans = 'C': (\text{diag}(R) * A * \text{diag}(C))^H * \text{diag}(R)^{-1} * X = \text{diag}(C) * B$$

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(R) * A * \text{diag}(C)$ and B by $\text{diag}(R) * B$ (if $\text{trans} = 'N'$) or $\text{diag}(C) * B$ (if $\text{trans} = 'T'$ or $'C'$).

2. If $\text{fact} = 'N'$ or $'E'$, the LU decomposition is used to factor the matrix A (after equilibration if $\text{fact} = 'E'$) as $A = P L U$, where P is a permutation matrix, L is a unit lower triangular matrix, and U is upper triangular.
3. The factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than relative machine precision, steps 4 - 6 are skipped.
4. The system of equations is solved for X using the factored form of A .
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix X is premultiplied by $\text{diag}(C)$ (if $\text{trans} = 'N'$) or $\text{diag}(R)$ (if $\text{trans} = 'T'$ or $'C'$) so that it solves the original system before equilibration.

Input Parameters

fact (global) CHARACTER*1. Must be 'F', 'N', or 'E'.
 Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.
 If $\text{fact} = 'F'$ then, on entry, af and $ipiv$ contain the factored form of A . If equed is not 'N', the matrix A has been equilibrated with scaling factors given by r and c . Arrays a , af , and $ipiv$ are not modified.
 If $\text{fact} = 'N'$, the matrix A is copied to af and factored.
 If $\text{fact} = 'E'$, the matrix A is equilibrated if necessary, then copied to af and factored.

trans (global) CHARACTER*1. Must be 'N', 'T', or 'C'.
 Specifies the form of the system of equations:
 If $\text{trans} = 'N'$, the system has the form $A * X = B$ (No transpose);
 If $\text{trans} = 'T'$, the system has the form $A^T * X = B$ (Transpose);
 If $\text{trans} = 'C'$, the system has the form $A^H * X = B$ (Conjugate transpose);

<i>n</i>	(global) INTEGER. The number of linear equations; the order of the submatrix <i>A</i> ($n \geq 0$).
<i>nrhs</i>	(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrices <i>B</i> and <i>X</i> ($nrhs \geq 0$).
<i>a, af, b, work</i>	(local) REAL for psgesvx DOUBLE PRECISION for pdgesvx COMPLEX for pcgesvx DOUBLE COMPLEX for pzgesvx. Pointers into the local memory to arrays of local dimension <i>a</i>(<i>lld_a</i>, <i>LOCc</i>(<i>ja</i>+<i>n</i>-1)), <i>af</i>(<i>lld_af</i>, <i>LOCc</i>(<i>ja</i>+<i>n</i>-1)), <i>b</i>(<i>lld_b</i>, <i>LOCc</i>(<i>jb</i>+<i>nrhs</i>-1)), <i>work</i>(<i>lwork</i>), respectively. The array <i>a</i> contains the matrix <i>A</i>. If <i>fact</i> = 'F' and <i>equed</i> is not 'N', then <i>A</i> must have been equilibrated by the scaling factors in <i>r</i> and/or <i>c</i>. The array <i>af</i> is an input argument if <i>fact</i> = 'F'. In this case it contains on entry the factored form of the matrix <i>A</i>, that is, the factors <i>L</i> and <i>U</i> from the factorization $A = P^*L^*U$ as computed by p?getrf. If <i>equed</i> is not 'N', then <i>af</i> is the factored form of the equilibrated matrix <i>A</i>. The array <i>b</i> contains on entry the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations. <i>work</i>(*) is a workspace array. The dimension of <i>work</i> is (<i>lwork</i>).
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>A</i> indicating the first row and the first column of the submatrix <i>A</i> (<i>ia</i> : <i>ia</i> + <i>n</i> -1, <i>ja</i> : <i>ja</i> + <i>n</i> -1), respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>iaf, jaf</i>	(global) INTEGER. The row and column indices in the global array <i>af</i> indicating the first row and the first column of the subarray <i>af</i> (<i>iaf</i> : <i>iaf</i> + <i>n</i> -1, <i>jaf</i> : <i>jaf</i> + <i>n</i> -1), respectively.
<i>descaf</i>	(global and local) INTEGER array, dimension (<i>dlen</i>). The array descriptor for the distributed matrix <i>AF</i> .

<i>ib, jb</i>	(global) INTEGER. The row and column indices in the global array <i>B</i> indicating the first row and the first column of the submatrix <i>B(ib:ib+n-1, jb:jb+nrhs-1)</i> , respectively.
<i>descb</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>B</i> .
<i>ipiv</i>	(local) INTEGER array. The dimension of <i>ipiv</i> is $LOCr(m\_a) + mb\_a$. The array <i>ipiv</i> is an input argument if <i>fact</i> = 'F'. On entry, it contains the pivot indices from the factorization $A = P * L * U$ as computed by <i>p?getrf</i> ; (local) row <i>i</i> of the matrix was interchanged with the (global) row <i>ipiv(i)</i> . This array must be aligned with <i>A(ia:ia+n-1, *)</i> .
<i>equed</i>	(global) CHARACTER*1. Must be 'N', 'R', 'C', or 'B'. <i>equed</i> is an input argument if <i>fact</i> = 'F'. It specifies the form of equilibration that was done: If <i>equed</i> = 'N', no equilibration was done (always true if <i>fact</i> = 'N'); If <i>equed</i> = 'R', row equilibration was done, that is, <i>A</i> has been premultiplied by <i>diag(r)</i> ; If <i>equed</i> = 'C', column equilibration was done, that is, <i>A</i> has been postmultiplied by <i>diag(c)</i> ; If <i>equed</i> = 'B', both row and column equilibration was done; <i>A</i> has been replaced by <i>diag(r) * A * diag(c)</i> .
<i>r, c</i>	(local) REAL for single precision flavors; DOUBLE PRECISION for double precision flavors. Arrays, dimension $LOCr(m\_a)$ and $LOCc(n\_a)$, respectively. The array <i>r</i> contains the row scale factors for <i>A</i> , and the array <i>c</i> contains the column scale factors for <i>A</i> . These arrays are input arguments if <i>fact</i> = 'F' only; otherwise they are output arguments. If <i>equed</i> = 'R' or 'B', <i>A</i> is multiplied on the left by <i>diag(r)</i> ; if <i>equed</i> = 'N' or 'C', <i>r</i> is not accessed. If <i>fact</i> = 'F' and <i>equed</i> = 'R' or 'B', each element of <i>r</i> must be positive. If <i>equed</i> = 'C' or 'B', <i>A</i> is multiplied on the right by <i>diag(c)</i> ; if <i>equed</i> = 'N' or 'R', <i>c</i> is not accessed.

If *fact* = 'F' and *equed* = 'C' or 'B', each element of *c* must be positive. Array *r* is replicated in every process column, and is aligned with the distributed matrix *A*. Array *c* is replicated in every process row, and is aligned with the distributed matrix *A*.

<i>ix, jx</i>	(global) INTEGER. The row and column indices in the global array <i>x</i> indicating the first row and the first column of the submatrix <i>X(ix:ix+n-1, jx:jx+nrhs-1)</i> , respectively.
<i>descx</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>x</i> .
<i>lwork</i>	(local or global) INTEGER. The dimension of the array <i>work</i> ; must be at least $\max(\text{p?gecon}(\text{lwork}), \text{p?gerfs}(\text{lwork})) + \text{LOCr}(\text{n\_a})$.
<i>iwork</i>	(local, psgesvx/pdgesvx only) INTEGER. Workspace array. The dimension of <i>iwork</i> is (<i>liwork</i>).
<i>liwork</i>	(local, psgesvx/pdgesvx only) INTEGER. The dimension of the array <i>iwork</i> , must be at least $\text{LOCr}(\text{n\_a})$.
<i>rwork</i>	(local) REAL for pcgesvx DOUBLE PRECISION for pzgesvx. Workspace array, used in complex flavors only. The dimension of <i>rwork</i> is (<i>lrwork</i>).
<i>lrwork</i>	(local or global, pcgesvx/pzgesvx only) INTEGER. The dimension of the array <i>rwork</i> ; must be at least $2 * \text{LOCc}(\text{n\_a})$.

Output Parameters

<i>x</i>	(local) REAL for psgesvx DOUBLE PRECISION for pdgesvx COMPLEX for pcgesvx DOUBLE COMPLEX for pzgesvx. Pointer into the local memory to an array of local dimension $x(\text{lld\_x}, \text{LOCc}(\text{jx} + \text{nrhs} - 1))$.

	If $info = 0$, the array x contains the solution matrix X to the <i>original</i> system of equations. Note that A and B are modified on exit if $equed \neq 'N'$, and the solution to the <i>equilibrated</i> system is: $diag(C)^{-1} * X$, if $trans = 'N'$ and $equed = 'C'$ or $'B'$; and $diag(R)^{-1} * X$, if $trans = 'T'$ or $'C'$ and $equed = 'R'$ or $'B'$.
a	Array a is not modified on exit if $fact = 'F'$ or $'N'$, or if $fact = 'E'$ and $equed = 'N'$. If $equed \neq 'N'$, A is scaled on exit as follows: $equed = 'R'$: $A = diag(R) * A$ $equed = 'C'$: $A = A * diag(c)$ $equed = 'B'$: $A = diag(R) * A * diag(c)$
af	If $fact = 'N'$ or $'E'$, then af is an output argument and on exit returns the factors L and U from the factorization $A = P * L * U$ of the original matrix A (if $fact = 'N'$) or of the equilibrated matrix A (if $fact = 'E'$). See the description of a for the form of the equilibrated matrix.
b	Overwritten by $diag(R) * B$ if $trans = 'N'$ and $equed = 'R'$ or $'B'$; overwritten by $diag(c) * B$ if $trans = 'T'$ and $equed = 'C'$ or $'B'$; not changed if $equed = 'N'$.
r, c	These arrays are output arguments if $fact \neq 'F'$. See the description of r, c in <i>Input Arguments</i> section.
$rcond$	(global) REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal condition number of the matrix A after equilibration (if done). The routine sets $rcond = 0$ if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime $rcond$ is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
$ferr, berr$	(local) REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

	Arrays, DIMENSION $LOCc(n\_b)$ each. Contain the component-wise forward and relative backward errors, respectively, for each solution vector. Arrays <i>ferr</i> and <i>berr</i> are both replicated in every process row, and are aligned with the matrices <i>B</i> and <i>X</i> .
<i>ipiv</i>	If <i>fact</i> = 'N' or 'E', then <i>ipiv</i> is an output argument and on exit contains the pivot indices from the factorization $A = P*L*U$ of the original matrix <i>A</i> (if <i>fact</i> = 'N') or of the equilibrated matrix <i>A</i> (if <i>fact</i> = 'E').
<i>equed</i>	If <i>fact</i> ≠ 'F' , then <i>equed</i> is an output argument. It specifies the form of equilibration that was done (see the description of <i>equed</i> in <i>Input Arguments</i> section).
<i>work</i> (1)	If <i>info</i> =0, on exit <i>work</i> (1) returns the minimum value of <i>lwork</i> required for optimum performance.
<i>iwork</i> (1)	If <i>info</i> =0, on exit <i>iwork</i> (1) returns the minimum value of <i>liwork</i> required for optimum performance.
<i>rwork</i> (1)	If <i>info</i> =0, on exit <i>rwork</i> (1) returns the minimum value of <i>lrwork</i> required for optimum performance.
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. <i>info</i> < 0: if the <i>i</i> th argument is an array and the <i>j</i> th entry had an illegal value, then <i>info</i> = $-(i*100+j)$; if the <i>i</i> th argument is a scalar and had an illegal value, then <i>info</i> = $-i$. If <i>info</i> = <i>i</i> , and $i \leq n$, then $U(i,i)$ is exactly zero. The factorization has been completed, but the factor <i>U</i> is exactly singular, so the solution and error bounds could not be computed. If <i>info</i> = <i>i</i> , and $i = n + 1$, then <i>U</i> is nonsingular, but <i>rcond</i> is less than machine precision. The factorization has been completed, but the matrix is singular to working precision and the solution and error bounds have not been computed.

p?gbsv

Computes the solution to the system of linear equations with a general banded distributed matrix and multiple right-hand sides.

Syntax

```
call psgbsv(n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, work, lwork,
info)

call pdgbsv(n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, work, lwork,
info)

call pcgbsv(n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, work, lwork,
info)

call pzgbsv(n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, work, lwork,
info)
```

Description

The routine p?gbsv computes the solution to a real or complex system of linear equations

$$\text{sub}(A) * X = \text{sub}(B),$$

where $\text{sub}(A) = A(1:n, ja:ja+n-1)$ is an n -by- n real/complex general banded distributed matrix with bwl subdiagonals and bwu superdiagonals, and X and $\text{sub}(B) = B(ib:ib+n-1, 1:rhs)$ are n -by- $nrhs$ distributed matrices.

The LU decomposition with partial pivoting and row interchanges is used to factor $\text{sub}(A)$ as $\text{sub}(A) = P * L * U * Q$, where P and Q are permutation matrices, and L and U are banded lower and upper triangular matrices, respectively. The matrix Q represents reordering of columns for the sake of parallelism, while P represents reordering of rows for numerical stability using classic partial pivoting.

Input Parameters

n	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).
bwl	(global) INTEGER. The number of subdiagonals within the band of A ($0 \leq bwl \leq n-1$).

<i>bwu</i>	(global) INTEGER. The number of superdiagonals within the band of <i>A</i> ($0 \leq bwu \leq n-1$).
<i>nrhs</i>	(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix <i>sub(B)</i> ($nrhs \geq 0$).
<i>a, b</i>	(local) REAL for psgbsv DOUBLE PRECISION for pdgbsv COMPLEX for pcgbsv DOUBLE COMPLEX for pzgbsv. Pointers into the local memory to arrays of local dimension <i>a(lld_a, LOCC(ja+n-1))</i> and <i>b(lld_b, LOCC(nrhs))</i> , respectively. On entry, the array <i>a</i> contains the local pieces of the global array <i>A</i> . On entry, the array <i>b</i> contains the right hand side distributed matrix <i>sub(B)</i> .
<i>ja</i>	(global) INTEGER. The index in the global array <i>A</i> that points to the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> . If <i>desca(dtype_)</i> = 501, then <i>dlen_</i> ≥ 7; else if <i>desca(dtype_)</i> = 1, then <i>dlen_</i> ≥ 9.
<i>ib</i>	(global) INTEGER. The row index in the global array <i>B</i> that points to the first row of the matrix to be operated on (which may be either all of <i>B</i> or a submatrix of <i>B</i>).
<i>descb</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>B</i> . If <i>descb(dtype_)</i> = 502, then <i>dlen_</i> ≥ 7; else if <i>descb(dtype_)</i> = 1, then <i>dlen_</i> ≥ 9.
<i>work</i>	(local) REAL for psgbsv DOUBLE PRECISION for pdgbsv COMPLEX for pcgbsv

DOUBLE COMPLEX for pzgbsv.
 Workspace array of dimension (*lwork*).
lwork (local or global) INTEGER. The size of the array *work*, must be at least $lwork \geq (NB+bwu) * (bwl+bwu) + 6 * (bwl+bwu) * (bwl+2 * bwu) + \max(nrhs * (NB+2 * bwl+4 * bwu), 1)$.

Output Parameters

a On exit, contains details of the factorization. Note that the resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

b On exit, this array contains the local pieces of the solution distributed matrix *x*.

ipiv (local) INTEGER array.
 The dimension of *ipiv* must be at least *desca*(NB). This array contains pivot indices for local factorizations. You should not alter the contents between factorization and solve.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info INTEGER. If *info*=0, the execution is successful. *info* < 0:
 If the *i*th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.
info > 0:
 If *info* = *k* ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not nonsingular, and the factorization was not completed. If *info* = *k* > NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not nonsingular, and the factorization was not completed.

p?dbsv

Solves a general band system of linear equations.

Syntax

```
call psdbsv(n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
call pddbsv(n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
call pcdbsv(n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
call pzdbsv(n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
```

Description

This routine solves the system of linear equations

$$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs),$$

where $A(1:n, ja:ja+n-1)$ is an n -by- n real/complex banded diagonally dominant-like distributed matrix with bandwidth bwl, bwu .

Gaussian elimination without pivoting is used to factor a reordering of the matrix into LU .

Input Parameters

n	(global) INTEGER. The order of the distributed submatrix A , ($n \geq 0$).
bwl	(global) INTEGER. Number of subdiagonals. $0 \leq bwl \leq n-1$.
bwu	(global) INTEGER. Number of subdiagonals. $0 \leq bwu \leq n-1$.
$nrhs$	(global) INTEGER. The number of right-hand sides; the number of columns of the distributed submatrix B , ($nrhs \geq 0$).
a	(local). REAL for psdbsv DOUBLE PRECISION for pddbsv COMPLEX for pcdbsv DOUBLE COMPLEX for pzdbsv.

	Pointer into the local memory to an array with first dimension $lld\_a \geq (bwl+bwu+1)$ (stored in <i>desca</i>). On entry, this array contains the local pieces of the distributed matrix.
<i>ja</i>	(global) INTEGER. The index in the global array <i>a</i> that points to the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	(global and local) INTEGER array of dimension <i>dlen</i>. If 1d type (<i>dtype_a</i>=501 or 502), <i>dlen</i> ≥ 7; If 2d type (<i>dtype_a</i>=1), <i>dlen</i> ≥ 9. The array descriptor for the distributed matrix <i>A</i>. Contains information of mapping of <i>A</i> to memory.
<i>b</i>	(local) REAL for psdbsv DOUBLE PRECISION for pddbsv COMPLEX for pcdbsv DOUBLE COMPLEX for pzdbsv. Pointer into the local memory to an array of local lead dimension $lld\_b \geq nb$. On entry, this array contains the local pieces of the right hand sides $B(ib:ib+n-1, 1:nrhs)$.
<i>ib</i>	(global) INTEGER. The row index in the global array <i>b</i> that points to the first row of the matrix to be operated on (which may be either all of <i>b</i> or a submatrix of <i>B</i>).
<i>descb</i>	(global and local) INTEGER array of dimension <i>dlen</i>. If 1d type (<i>dtype_b</i> =502), <i>dlen</i> ≥ 7; If 2d type (<i>dtype_b</i> =1), <i>dlen</i> ≥ 9. The array descriptor for the distributed matrix <i>B</i>. Contains information of mapping of <i>B</i> to memory.
<i>work</i>	(local). REAL for psdbsv DOUBLE PRECISION for pddbsv COMPLEX for pcdbsv DOUBLE COMPLEX for pzdbsv.

Temporary workspace. This space may be overwritten in between calls to routines. *work* must be the size given in *lwork*.

lwork

(local or global) INTEGER. Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.

$lwork \geq$

$nb(bwl+bwu) + 6 \max(bwl, bwu) * \max(bwl, bwu) + \max(\max(bwl, bwu) nrhs, \max(bwl, bwu) * \max(bwl, bwu))$

Output Parameters

a

On exit, this array contains information containing details of the factorization.

Note that permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.

b

On exit, this contains the local piece of the solutions distributed matrix *x*.

work

On exit, *work*(1) contains the minimal *lwork*.

info

(local) INTEGER. If *info*=0, the execution is successful.

< 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = $-(i*100+j)$, if the *i*-th argument is a scalar and had an illegal value, then *info* = $-i$.

> 0: If *info* = *k* < NPROCS, the submatrix stored on processor *info* and factored locally was not positive definite, and the factorization was not completed.

If *info* = *k* > NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not positive definite, and the factorization was not completed.

p?dtsv

Solves a general tridiagonal system of linear equations.

Syntax

```
call psdtsv(n, nrhs, dl, d, du, ja, desca, b, ib, descb, work, lwork, info)
call pddtsv(n, nrhs, dl, d, du, ja, desca, b, ib, descb, work, lwork, info)
call pcdtsv(n, nrhs, dl, d, du, ja, desca, b, ib, descb, work, lwork, info)
call pzdtvs(n, nrhs, dl, d, du, ja, desca, b, ib, descb, work, lwork, info)
```

Description

This routine solves a system of linear equations

$$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs),$$

where $A(1:n, ja:ja+n-1)$ is an n -by- n complex tridiagonal diagonally dominant-like distributed matrix.

Gaussian elimination without pivoting is used to factor a reordering of the matrix into $L U$.

Input Parameters

n	(global) INTEGER. The order of the distributed submatrix A ($n \geq 0$).
$nrhs$	INTEGER. The number of right hand sides; the number of columns of the distributed matrix B ($nrhs \geq 0$).
dl	(local). REAL for psdtsv DOUBLE PRECISION for pddtsv COMPLEX for pcdtsv DOUBLE COMPLEX for pzdtvs. Pointer to local part of global vector storing the lower diagonal of the matrix. Globally, $dl(1)$ is not referenced, and dl must be aligned with d . Must be of size $> desca(nb\_)$.
d	(local). REAL for psdtsv DOUBLE PRECISION for pddtsv COMPLEX for pcdtsv

	DOUBLE COMPLEX for pzdtsv. Pointer to local part of global vector storing the main diagonal of the matrix.
<i>du</i>	(local). REAL for psdtsv DOUBLE PRECISION for pddtsv COMPLEX for pcdtsv DOUBLE COMPLEX for pzdtsv. Pointer to local part of global vector storing the upper diagonal of the matrix. Globally, <i>du</i> (<i>n</i>) is not referenced, and <i>du</i> must be aligned with <i>d</i> .
<i>ja</i>	(global) INTEGER. The index in the global array <i>a</i> that points to the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	(global and local) INTEGER array of dimension <i>dlen</i> . If 1d type (<i>dtype_a</i> =501 or 502), <i>dlen</i> ≥ 7; If 2d type (<i>dtype_a</i> =1), <i>dlen</i> ≥ 9. The array descriptor for the distributed matrix <i>A</i> . Contains information of mapping of <i>A</i> to memory.
<i>b</i>	(local) REAL for psdtsv DOUBLE PRECISION for pddtsv COMPLEX for pcdtsv DOUBLE COMPLEX for pzdtsv. Pointer into the local memory to an array of local lead dimension <i>lld_b</i> > <i>nb</i> . On entry, this array contains the local pieces of the right hand sides <i>B</i> (<i>ib:ib+n-1</i> , 1: <i>nrhs</i>).
<i>ib</i>	(global) INTEGER. The row index in the global array <i>b</i> that points to the first row of the matrix to be operated on (which may be either all of <i>b</i> or a submatrix of <i>B</i>).
<i>descb</i>	(global and local) INTEGER array of dimension <i>dlen</i> . If 1d type (<i>dtype_b</i> =502), <i>dlen</i> ≥ 7; If 2d type (<i>dtype_b</i> =1), <i>dlen</i> ≥ 9. The array descriptor for the distributed matrix <i>B</i> . Contains information of mapping of <i>B</i> to memory.
<i>work</i>	(local).

REAL for psdtsv
 DOUBLE PRECISION for pddtsv
 COMPLEX for pcdtsv
 DOUBLE COMPLEX for pzdtstv. Temporary workspace. This space may be overwritten in between calls to routines. *work* must be the size given in *lwork*.

lwork (local or global) INTEGER. Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned. *lwork* > (12*NPCOL+3*nb)+max((10+2*min(100, nrhs))*NPCOL+4*nrhs, 8*NPCOL)

Output Parameters

dl On exit, this array contains information containing the * factors of the matrix.

d On exit, this array contains information containing the * factors of the matrix. Must be of size > *desca*(*nb_*).

du On exit, this array contains information containing the * factors of the matrix. Must be of size > *desca*(*nb_*).

b On exit, this contains the local piece of the solutions distributed matrix *X*.

work On exit, *work*(1) contains the minimal *lwork*.

info (local) INTEGER. If *info*=0, the execution is successful.
 < 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.
 > 0: If *info* = *k* < NPROCS, the submatrix stored on processor *info* and factored locally was not positive definite, and the factorization was not completed.
 If *info* = *k* > NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not positive definite, and the factorization was not completed.

p?posv

Solves a symmetric positive definite system of linear equations.

Syntax

```
call psposv(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pdposv(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pcposv(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pzposv(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
```

Description

This routine computes the solution to a real/complex system of linear equations

$$\text{sub}(A) * X = \text{sub}(B),$$

where $\text{sub}(A)$ denotes $A(ia:ia+n-1, ja:ja+n-1)$ and is an n -by- n symmetric/Hermitian distributed positive definite matrix and X and $\text{sub}(B)$ denoting $B(ib:ib+n-1, jb:jb+nrhs-1)$ are n -by- $nrhs$ distributed matrices. The Cholesky decomposition is used to factor $\text{sub}(A)$ as

$$\text{sub}(A) = U^T * U, \text{ if } uplo = 'U', \text{ or}$$

$$\text{sub}(A) = L * L^T, \text{ if } uplo = 'L',$$

where U is an upper triangular matrix and L is a lower triangular matrix. The factored form of $\text{sub}(A)$ is then used to solve the system of equations.

Input Parameters

<i>uplo</i>	(global). CHARACTER. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of $\text{sub}(A)$ is stored.
<i>n</i>	(global) INTEGER. The order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns of the distributed submatrix $\text{sub}(B)$ ($nrhs \geq 0$).
<i>a</i>	(local) REAL for psposv DOUBLE PRECISION for pdposv

	COMPLEX for pcpovs COMPLEX*16 for pzposv. Pointer into the local memory to an array of dimension $(lld\_a, LOC(ja+n-1))$. On entry, this array contains the local pieces of the n-by-n symmetric distributed matrix $\text{sub}(A)$ to be factored. If $uplo = 'U'$, the leading n-by-n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If $uplo = 'L'$, the leading n-by-n lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the distributed matrix, and its strictly upper triangular part is not referenced.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension $(dlen\_)$. The array descriptor for the distributed matrix <i>A</i> .
<i>b</i>	(local) REAL for pspovs DOUBLE PRECISION for pdposv COMPLEX for pcpovs COMPLEX*16 for pzposv. Pointer into the local memory to an array of dimension $(lld\_b, LOC(jb+nrhs-1))$. On entry, the local pieces of the right hand sides distributed matrix $\text{sub}(B)$.
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the global array <i>b</i> indicating the first row and the first column of the submatrix <i>B</i> , respectively.
<i>descb</i>	(global and local) INTEGER array, dimension $(dlen\_)$. The array descriptor for the distributed matrix <i>B</i> .

Output Parameters

<i>a</i>	On exit, if $info = 0$, this array contains the local pieces of the factor <i>U</i> or <i>L</i> from the Cholesky factorization $\text{sub}(A) = U^H * U$, or $L * L^H$.

<i>b</i>	On exit, if <i>info</i> = 0, sub(<i>B</i>) is overwritten by the solution distributed matrix <i>X</i> .
<i>info</i>	(global) INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = -(<i>i</i> *100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> . If <i>info</i> > 0: If <i>info</i> = <i>k</i> , the leading minor of order <i>k</i> , <i>A</i> (<i>ia:ia+k-1, ja:ja+k-1</i>) is not positive definite, and the factorization could not be completed, and the solution has not been computed.

p?posvx

Solves a symmetric or Hermitian positive definite system of linear equations.

Syntax

```
call psposvx(fact, uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf,
equed, sr, sc, b, ib, jb, descb, x, ix, jx, descx, rcond, ferr, berr, work,
lwork, iwork, liwork, info)
```

```
call pdposvx(fact, uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf,
equed, sr, sc, b, ib, jb, descb, x, ix, jx, descx, rcond, ferr, berr, work,
lwork, iwork, liwork, info)
```

```
call pcposvx(fact, uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf,
equed, sr, sc, b, ib, jb, descb, x, ix, jx, descx, rcond, ferr, berr, work,
lwork, iwork, liwork, info)
```

```
call pzposvx(fact, uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf,
equed, sr, sc, b, ib, jb, descb, x, ix, jx, descx, rcond, ferr, berr, work,
lwork, iwork, liwork, info)
```

Description

This routine uses the Cholesky factorization $A=U^T*U$ or $A=L*L^T$ to compute the solution to a real or complex system of linear equations

$$A(ia:ia+n-1, ja:ja+n-1)*X = B(ib:ib+n-1, jb:jb+nrhs-1),$$

where $A(ia:ia+n-1, ja:ja+n-1)$ is a n -by- n matrix and X and $B(ib:ib+n-1, jb:jb+nrhs-1)$ are n -by- $nrhs$ matrices.

Error bounds on the solution and a condition estimate are also provided.

In the following comments y denotes $Y(iy:iy+m-1, jy:jy+k-1)$ a m -by- k matrix where y can be a, af, b and x .

The routine `p?posvx` performs the following steps:

1. If $fact = 'E'$, real scaling factors s are computed to equilibrate the system:

$$\text{diag}(sr) * A * \text{diag}(sc) * \text{inv}(\text{diag}(sc)) * X = \text{diag}(sr) * B$$

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(sr) * A * \text{diag}(sc)$ and B by $\text{diag}(sr) * B$.

2. If $fact = 'N'$ or $'E'$, the Cholesky decomposition is used to factor the matrix A (after equilibration if $fact = 'E'$) as

$$A = U^T * U, \text{ if } uplo = 'U', \text{ or}$$

$$A = L * L^T, \text{ if } uplo = 'L',$$

where U is an upper triangular matrix and L is a lower triangular matrix.

3. The factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, steps 4-6 are skipped
4. The system of equations is solved for x using the factored form of A .
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix x is premultiplied by $\text{diag}(sr)$ so that it solves the original system before equilibration.

Input Parameters

fact (global) CHARACTER. Must be 'F', 'N', or 'E'.
Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.
If $fact = 'F'$: on entry, af contains the factored form of A . If $equed = 'Y'$, the matrix A has been equilibrated with scaling factors given by s . a and af will not be modified.

	If <i>fact</i> = 'N', the matrix <i>A</i> will be copied to <i>af</i> and factored. If <i>fact</i> = 'E', the matrix <i>A</i> will be equilibrated if necessary, then copied to <i>af</i> and factored.
<i>uplo</i>	(global) CHARACTER. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of <i>A</i> is stored.
<i>n</i>	(global) INTEGER. The order of the distributed submatrix <i>sub(A)</i> ($n \geq 0$).
<i>nrhs</i>	(global) INTEGER. The number of right-hand sides; the number of columns of the distributed submatrices <i>B</i> and <i>X</i>. ($nrhs \geq 0$).
<i>a</i>	(local) REAL for <i>psposvx</i> DOUBLE PRECISION for <i>pdposvx</i> COMPLEX for <i>pcposvx</i> DOUBLE COMPLEX for <i>pzposvx</i>. Pointer into the local memory to an array of local dimension (<i>lld_a</i>, <i>LOCc(ja+n-1)</i>). On entry, the symmetric/Hermitian matrix <i>A</i>, except if <i>fact</i> = 'F' and <i>equed</i> = 'Y', then <i>A</i> must contain the equilibrated matrix <i>diag(sr)*A*diag(sc)</i>. If <i>uplo</i> = 'U', the leading <i>n</i>-by-<i>n</i> upper triangular part of <i>A</i> contains the upper triangular part of the matrix <i>A</i>, and the strictly lower triangular part of <i>A</i> is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i>-by-<i>n</i> lower triangular part of <i>A</i> contains the lower triangular part of the matrix <i>A</i>, and the strictly upper triangular part of <i>A</i> is not referenced. <i>A</i> is not modified if <i>fact</i> = 'F' or 'N', or if <i>fact</i> = 'E' and <i>equed</i> = 'N' on exit.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i>, respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i>.
<i>af</i>	(local)

REAL for psposvx
 DOUBLE PRECISION for pdposvx
 COMPLEX for pcposvx
 DOUBLE COMPLEX for pzposvx.
 Pointer into the local memory to an array of local dimension
 (lld_af , $LOCc(ja+n-1)$).
 If $fact = 'F'$, then af is an input argument and on entry
 contains the triangular factor U or L from the Cholesky
 factorization $A = U^T * U$ or $A = L * L^T$, in the same storage
 format as A . If $equed \neq 'N'$, then af is the factored form
 of the equilibrated matrix $diag(sr) * A * diag(sc)$.
iaf, jaf (global) INTEGER. The row and column indices in the global
 array af indicating the first row and the first column of the
 submatrix AF , respectively.
descaf (global and local) INTEGER array, dimension ($dlen\_$). The
 array descriptor for the distributed matrix AF .
equed (global). CHARACTER. Must be 'N' or 'Y'.
equed is an input argument if $fact = 'F'$. It specifies the
 form of equilibration that was done:
 If $equed = 'N'$, no equilibration was done (always true if
 $fact = 'N'$);
 If $equed = 'Y'$, equilibration was done and A has been
 replaced by $diag(sr) * A * diag(sc)$.
sr (local)
 REAL for psposvx
 DOUBLE PRECISION for pdposvx
 COMPLEX for pcposvx
 DOUBLE COMPLEX for pzposvx.
 Array, DIMENSION (lld_a).
 The array s contains the scale factors for A . This array is an
 input argument if $fact = 'F'$ only; otherwise it is an output
 argument.
 If $equed = 'N'$, s is not accessed.
 If $fact = 'F'$ and $equed = 'Y'$, each element of s must
 be positive.
b (local)
 REAL for psposvx

	DOUBLE PRECISION for pdposvx COMPLEX for pcposvx DOUBLE COMPLEX for pzposvx. Pointer into the local memory to an array of local dimension $(lld\_b, LOCc(jb+nrhs-1))$. On entry, the n -by- $nrhs$ right-hand side matrix B .
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the global array b indicating the first row and the first column of the submatrix B , respectively.
<i>descb</i>	(global and local) INTEGER. Array, dimension $(dlen\_)$. The array descriptor for the distributed matrix B .
<i>x</i>	(local) REAL for psposvx DOUBLE PRECISION for pdposvx COMPLEX for pcposvx DOUBLE COMPLEX for pzposvx. Pointer into the local memory to an array of local dimension $(lld\_x, LOCc(jx+nrhs-1))$.
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the global array x indicating the first row and the first column of the submatrix x , respectively.
<i>descx</i>	(global and local) INTEGER array, dimension $(dlen\_)$. The array descriptor for the distributed matrix x .
<i>work</i>	(local) REAL for psposvx DOUBLE PRECISION for pdposvx COMPLEX for pcposvx DOUBLE COMPLEX for pzposvx. Workspace array, DIMENSION $(lwork)$.
<i>lwork</i>	(local or global) INTEGER. The dimension of the array <i>work</i> . <i>lwork</i> is local input and must be at least $lwork = \max(p?pocon(lwork), p?porfs(lwork)) + LOCr(n\_a)$. $lwork = 3*desca(lld\_)$.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

iwork

(local) INTEGER. Workspace array, dimension ($liwork$).

liwork

(local or global)

INTEGER. The dimension of the array *iwork*. *liwork* is local input and must be at least $liwork = desca(lld\_)$ $liwork = LOCr(n\_a)$.

If $liwork = -1$, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a

On exit, if $fact = 'E'$ and $equed = 'Y'$, *a* is overwritten by $diag(sr) * a * diag(sc)$.

af

If $fact = 'N'$, then *af* is an output argument and on exit returns the triangular factor *U* or *L* from the Cholesky factorization $A = U^T * U$ or $A = L * L^T$ of the original matrix *A*.

If $fact = 'E'$, then *af* is an output argument and on exit returns the triangular factor *U* or *L* from the Cholesky factorization $A = U^T * U$ or $A = L * L^T$ of the equilibrated matrix *A* (see the description of *A* for the form of the equilibrated matrix).

equed

If $fact \neq 'F'$, then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

sr

This array is an output argument if $fact \neq 'F'$. See the description of *sr* in *Input Arguments* section.

sc

This array is an output argument if $fact \neq 'F'$.

	See the description of <i>sc</i> in <i>Input Arguments</i> section.
<i>b</i>	On exit, if <i>equed</i> = 'N', <i>b</i> is not modified; if <i>trans</i> = 'N' and <i>equed</i> = 'R' or 'B', <i>b</i> is overwritten by $\text{diag}(r) * b$; if <i>trans</i> = 'T' or 'C' and <i>equed</i> = 'C' or 'B', <i>b</i> is overwritten by $\text{diag}(c) * b$.
<i>x</i>	(local) REAL for <i>psposvx</i> DOUBLE PRECISION for <i>pdposvx</i> COMPLEX for <i>pcposvx</i> DOUBLE COMPLEX for <i>pzposvx</i> . If <i>info</i> = 0 the <i>n</i> -by- <i>nrhs</i> solution matrix <i>x</i> to the original system of equations. Note that <i>A</i> and <i>B</i> are modified on exit if <i>equed</i> ≠ 'N', and the solution to the equilibrated system is $\text{inv}(\text{diag}(sc)) * X$ if <i>trans</i> = 'N' and <i>equed</i> = 'C' or 'B', or $\text{inv}(\text{diag}(sr)) * X$ if <i>trans</i> = 'T' or 'C' and <i>equed</i> = 'R' or 'B'.
<i>rcond</i>	(global) REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal condition number of the matrix <i>A</i> after equilibration (if done). If <i>rcond</i> is less than the machine precision (in particular, if <i>rcond</i> =0), the matrix is singular to working precision. This condition is indicated by a return code of <i>info</i> > 0.
<i>ferr</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least $\max(LOC, n\_b)$. The estimated forward error bounds for each solution vector <i>x</i> (<i>j</i>) (the <i>j</i> -th column of the solution matrix <i>x</i>). If <i>xtrue</i> is the true solution, <i>ferr</i> (<i>j</i>) bounds the magnitude of the largest entry in $(X(j) - xtrue)$ divided by the magnitude of the largest entry in <i>x</i> (<i>j</i>). The quality of the error bound depends on the quality of the estimate of $\text{norm}(\text{inv}(A))$ computed in the code; if the estimate of $\text{norm}(\text{inv}(A))$ is accurate, the error bound is guaranteed.

berr (local)
 REAL for single precision flavors.
 DOUBLE PRECISION for double precision flavors.
 Arrays, DIMENSION at least $\max(LOC, n\_b)$. The componentwise relative backward error of each solution vector $x(j)$ (the smallest relative change in any entry of A or B that makes $x(j)$ an exact solution).

work(1) (local) On exit, *work(1)* returns the minimal and optimal *liwork*.

info (global) INTEGER.
 If *info*=0, the execution is successful.
 < 0: if *info* = -*i*, the *i*-th argument had an illegal value
 > 0: if *info* = *i*, and *i* is ≤ *n*: if *info* = *i*, the leading minor of order *i* of *a* is not positive definite, so the factorization could not be completed, and the solution and error bounds could not be computed.
 = *n*+1: *rcond* is less than machine precision. The factorization has been completed, but the matrix is singular to working precision, and the solution and error bounds have not been computed.

p?pbsv

Solves a symmetric/Hermitian positive definite banded system of linear equations.

Syntax

```
call pspbsv(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
call pdpbsv(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
call pcpbsv(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
call pzpbsv(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
```

Description

This routine solves a system of linear equations

$$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs),$$

where $A(1:n, ja:ja+n-1)$ is an n -by- n real/complex banded symmetric positive definite distributed matrix with bandwidth bw .

Cholesky factorization is used to factor a reordering of the matrix into L^*L' .

Input Parameters

<i>uplo</i>	(global) CHARACTER. Must be 'U' or 'L'. Indicates whether the upper or lower triangular of A is stored. If <i>uplo</i> = 'U', the upper triangular A is stored If <i>uplo</i> = 'L', the lower triangular of A is stored.
<i>n</i>	(global) INTEGER. The order of the distributed matrix A ($n \geq 0$).
<i>bw</i>	(global) INTEGER. The number of subdiagonals in L or U . $0 \leq bw \leq n-1$.
<i>nrhs</i>	(global) INTEGER. The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).
<i>a</i>	(local). REAL for pspbsv DOUBLE PRECISION for pdpbsv COMPLEX for pcpbsv DOUBLE COMPLEX for pzpbsv. Pointer into the local memory to an array with first dimension $lld\_a \geq (bw+1)$ (stored in <i>desca</i>). On entry, this array contains the local pieces of the distributed matrix $sub(A)$ to be factored.
<i>ja</i>	(global) INTEGER. The index in the global array <i>a</i> that points to the start of the matrix to be operated on (which may be either all of A or a submatrix of A).
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix A .
<i>b</i>	(local) REAL for pspbsv DOUBLE PRECISION for pdpbsv COMPLEX for pcpbsv DOUBLE COMPLEX for pzpbsv.

	Pointer into the local memory to an array of local lead dimension $lld\_b \geq nb$. On entry, this array contains the local pieces of the right hand sides $B(ib:ib+n-1, 1:nrhs)$.
<i>ib</i>	(global) INTEGER. The row index in the global array <i>b</i> that points to the first row of the matrix to be operated on (which may be either all of <i>b</i> or a submatrix of <i>B</i>).
<i>descb</i>	(global and local) INTEGER array of dimension <i>dlen</i> . If 1D type (<i>dtype_b</i> = 502), $dlen \geq 7$; If 2D type (<i>dtype_b</i> = 1), $dlen \geq 9$. The array descriptor for the distributed matrix <i>B</i> . Contains information of mapping of <i>B</i> to memory.
<i>work</i>	(local). REAL for pspbsv DOUBLE PRECISION for pdpbsv COMPLEX for pcpbsv DOUBLE COMPLEX for pzpbsv. Temporary workspace. This space may be overwritten in between calls to routines. <i>work</i> must be the size given in <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER. Size of user-input workspace <i>work</i> . If <i>lwork</i> is too small, the minimal acceptable size will be returned in <i>work</i> (1) and an error code is returned. $lwork \geq (nb+2*bw)*bw + \max(bw*nrhs, bw*bw)$

Output Parameters

<i>a</i>	On exit, this array contains information containing details of the factorization. Note that permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.
<i>b</i>	On exit, contains the local piece of the solutions distributed matrix <i>x</i> .
<i>work</i>	On exit, <i>work</i> (1) contains the minimal <i>lwork</i> .
<i>info</i>	(global). INTEGER. If <i>info</i> =0, the execution is successful.

< 0: If the i -th argument is an array and the j -entry had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

> 0: If $info = k \leq \text{NPROCS}$, the submatrix stored on processor $info$ and factored locally was not positive definite, and the factorization was not completed.

If $info = k > \text{NPROCS}$, the submatrix stored on processor $info - \text{NPROCS}$ representing interactions with other processors was not positive definite, and the factorization was not completed.

p?ptsv

Solves a symmetric or Hermitian positive definite tridiagonal system of linear equations.

Syntax

```
call psptsv(n, nrhs, d, e, ja, desca, b, ib, descb, work, lwork, info)
call pdptsv(n, nrhs, d, e, ja, desca, b, ib, descb, work, lwork, info)
call pcptsv(n, nrhs, d, e, ja, desca, b, ib, descb, work, lwork, info)
call pzptsv(n, nrhs, d, e, ja, desca, b, ib, descb, work, lwork, info)
```

Description

This routine solves a system of linear equations

$$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs),$$

where $A(1:n, ja:ja+n-1)$ is an n -by- n real tridiagonal symmetric positive definite distributed matrix.

Cholesky factorization is used to factor a reordering of the matrix into $L * L'$.

Input Parameters

n (global) INTEGER. The order of matrix A ($n \geq 0$).

<i>nrhs</i>	(global) INTEGER. The number of right-hand sides; the number of columns of the distributed submatrix <i>B</i> (<i>nrhs</i> ≥ 0).
<i>d</i>	(local) REAL for psptsv DOUBLE PRECISION for pdptsv COMPLEX for pcptsv DOUBLE COMPLEX for pzptsv. Pointer to local part of global vector storing the main diagonal of the matrix.
<i>e</i>	(local) REAL for psptsv DOUBLE PRECISION for pdptsv COMPLEX for pcptsv DOUBLE COMPLEX for pzptsv. Pointer to local part of global vector storing the upper diagonal of the matrix. Globally, <i>du</i> (<i>n</i>) is not referenced, and <i>du</i> must be aligned with <i>d</i> .
<i>ja</i>	(global) INTEGER. The index in the global array <i>A</i> that points to the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	(global and local) INTEGER array of dimension <i>dlen</i> . If 1 <i>d</i> type (<i>dtype_a</i> =501 or 502), <i>dlen</i> ≥ 7; If 2 <i>d</i> type (<i>dtype_a</i> =1), <i>dlen</i> ≥ 9. The array descriptor for the distributed matrix <i>A</i> . Contains information of mapping of <i>A</i> to memory.
<i>b</i>	(local) REAL for psptsv DOUBLE PRECISION for pdptsv COMPLEX for pcptsv DOUBLE COMPLEX for pzptsv. Pointer into the local memory to an array of local lead dimension <i>lld_b</i> ≥ <i>nb</i> . On entry, this array contains the local pieces of the right hand sides <i>B</i> (<i>ib</i> : <i>ib</i> + <i>n</i> -1, 1: <i>nrhs</i>).

<i>ib</i>	(global) INTEGER. The row index in the global array <i>b</i> that points to the first row of the matrix to be operated on (which may be either all of <i>b</i> or a submatrix of <i>B</i>).
<i>descb</i>	(global and local) INTEGER array of dimension <i>dlen</i> . If 1d type (<i>dtype_b</i> = 502), <i>dlen</i> ≥ 7; If 2d type (<i>dtype_b</i> = 1), <i>dlen</i> ≥ 9. The array descriptor for the distributed matrix <i>B</i> . Contains information of mapping of <i>B</i> to memory.
<i>work</i>	(local). REAL for psptsv DOUBLE PRECISION for pdptsv COMPLEX for pcptsv DOUBLE COMPLEX for pzptsv. Temporary workspace. This space may be overwritten in between calls to routines. <i>work</i> must be the size given in <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER. Size of user-input workspace <i>work</i> . If <i>lwork</i> is too small, the minimal acceptable size will be returned in <i>work</i> (1) and an error code is returned. <i>lwork</i> > (12*NPCOL+3*nb)+max((10+2*min(100, nrhs))*NPCOL+4*nrhs, 8*NPCOL).

Output Parameters

<i>d</i>	On exit, this array contains information containing the factors of the matrix. Must be of size greater than or equal to <i>desca</i> (<i>nb_</i>).
<i>e</i>	On exit, this array contains information containing the factors of the matrix. Must be of size greater than or equal to <i>desca</i> (<i>nb_</i>).
<i>b</i>	On exit, this contains the local piece of the solutions distributed matrix <i>x</i> .
<i>work</i>	On exit, <i>work</i> (1) contains the minimal <i>lwork</i> .
<i>info</i>	(local) INTEGER. If <i>info</i> =0, the execution is successful.

< 0 : If the i -th argument is an array and the j -entry had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

> 0 : If $info = k \leq NPROCS$, the submatrix stored on processor $info$ and factored locally was not positive definite, and the factorization was not completed.

If $info = k > NPROCS$, the submatrix stored on processor $info-NPROCS$ representing interactions with other processors was not positive definite, and the factorization was not completed.

p?gels

Solves overdetermined or underdetermined linear systems involving a matrix of full rank.

Syntax

```
call psgels(trans, m, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, work, lwork, info)
```

```
call pdgels(trans, m, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, work, lwork, info)
```

```
call pcgels(trans, m, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, work, lwork, info)
```

```
call pzgels(trans, m, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, work, lwork, info)
```

Description

This routine solves overdetermined or underdetermined real/ complex linear systems involving an m -by- n matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$, or its transpose/ conjugate-transpose, using a QTQ or LQ factorization of $\text{sub}(A)$. It is assumed that $\text{sub}(A)$ has full rank.

The following options are provided:

1. If $trans = 'N'$ and $m \geq n$: find the least squares solution of an overdetermined system, that is, solve the least squares problem

```
minimize ||sub(B) - sub(A)*X||
```

2. If $trans = 'N'$ and $m < n$: find the minimum norm solution of an underdetermined system
 $sub(A) * X = sub(B)$.
3. If $trans = 'T'$ and $m \geq n$: find the minimum norm solution of an undetermined system
 $sub(A)^T * X = sub(B)$.
4. If $trans = 'T'$ and $m < n$: find the least squares solution of an overdetermined system, that is, solve the least squares problem
 $minimize ||sub(B) - sub(A)^T * X||,$
 where $sub(B)$ denotes $B(ib:ib+m-1, jb:jb+nrhs-1)$ when $trans = 'N'$ and $B(ib:ib+n-1, jb:jb+nrhs-1)$ otherwise. Several right hand side vectors b and solution vectors x can be handled in a single call; when $trans = 'N'$, the solution vectors are stored as the columns of the n -by- $nrhs$ right hand side matrix $sub(B)$ and the m -by- $nrhs$ right hand side matrix $sub(B)$ otherwise.

Input Parameters

<i>trans</i>	(global) CHARACTER. Must be 'N', or 'T'. If $trans = 'N'$, the linear system involves matrix $sub(A)$; If $trans = 'T'$, the linear system involves the transposed matrix A^T (for real flavors only).
<i>m</i>	(global) INTEGER. The number of rows in the distributed submatrix $sub(A)$ ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed submatrix $sub(A)$ ($n \geq 0$).
<i>nrhs</i>	(global) INTEGER. The number of right-hand sides; the number of columns in the distributed submatrices $sub(B)$ and X . ($nrhs \geq 0$).
<i>a</i>	(local) REAL for psgels DOUBLE PRECISION for pdgels COMPLEX for pcgels DOUBLE COMPLEX for pzgels. Pointer into the local memory to an array of dimension $(lld\_a, LOCC(ja+n-1))$. On entry, contains the m -by- n matrix A .

<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>b</i>	(local) REAL for psgels DOUBLE PRECISION for pdgels COMPLEX for pcgels DOUBLE COMPLEX for pzgels. Pointer into the local memory to an array of local dimension (<i>lld_b</i> , <i>LOCc(jb+nrhs-1)</i>). On entry, this array contains the local pieces of the distributed matrix <i>B</i> of right-hand side vectors, stored columnwise; sub(<i>B</i>) is <i>m</i> -by- <i>nrhs</i> if <i>trans</i> ='N', and <i>n</i> -by- <i>nrhs</i> otherwise.
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the global array <i>b</i> indicating the first row and the first column of the submatrix <i>B</i> , respectively.
<i>descb</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>B</i> .
<i>work</i>	(local) REAL for psgels DOUBLE PRECISION for pdgels COMPLEX for pcgels DOUBLE COMPLEX for pzgels. Workspace array with dimension <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER. The dimension of the array <i>work</i> <i>lwork</i> is local input and must be at least $lwork \geq ltau + \max(lwf, lws)$, where if $m > n$, then $ltau = \text{numroc}(ja + \min(m, n) - 1, nb\_a, MYCOL, csrc\_a, NPCOL),$ $lwf = nb\_a * (mpa0 + nqa0 + nb\_a)$ $lws = \max((nb\_a * (nb\_a - 1)) / 2, (nrhsqb0 + mpb0) * nb\_a) + nb\_a * nb\_a$ else

```

ltau = numroc(ia+min(m,n)-1, mb_a, MYROW,
rsrc_a, NPROW),
lwf = mb_a * (mpa0 + nqa0 + mb_a)
lws = max((mb_a*(mb_a-1))/2, (npb0 + max(nqa0 +
numroc(numroc(n+iroffb, mb_a, 0, 0, NPROW),
mb_a, 0, 0, lcmp), nrhsqb0))*mb_a) + mb_a*mb_a
end if,
where lcmp = lcm/NPROW with lcm = ilcm(NPROW,
NPCOL),
iroffa = mod(ia-1, mb_a),
icoffa = mod(ja-1, nb_a),
iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
iacol= indxg2p(ja, nb_a, MYROW, rsrc_a, NPROW)
mpa0 = numroc(m+iroffa, mb_a, MYROW, iarow,
NPROW),
nqa0 = numroc(n+icoffa, nb_a, MYCOL, iacol,
NPCOL),
iroffb = mod(ib-1, mb_b),
icoffb = mod(jb-1, nb_b),
ibrow = indxg2p(ib, mb_b, MYROW, rsrc_b, NPROW),
ibcol = indxg2p(jb, nb_b, MYCOL, csrc_b, NPCOL),
mpb0 = numroc(m+iroffb, mb_b, MYROW, icrow,
NPROW),
nqb0 = numroc(n+icoffb, nb_b, MYCOL, ibcol,
NPCOL),
ilcm, indxg2p and numroc are ScaLAPACK tool functions;
MYROW, MYCOL, NPROW, and NPCOL can be determined by
calling the subroutine blacs_gridinfo.

```

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by *pxerbla*.

Output Parameters

<i>a</i>	On exit, If $m \geq n$, $\text{sub}(A)$ is overwritten by the details of its QR factorization as returned by <code>p?geqrf</code> ; if $m < n$, $\text{sub}(A)$ is overwritten by details of its LQ factorization as returned by <code>p?gelqf</code> .
<i>b</i>	On exit, $\text{sub}(B)$ is overwritten by the solution vectors, stored columnwise: if $trans = 'N'$ and $m \geq n$, rows 1 to n of $\text{sub}(B)$ contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements $n+1$ to m in that column; If $trans = 'N'$ and $m < n$, rows 1 to n of $\text{sub}(B)$ contain the minimum norm solution vectors; If $trans = 'T'$ and $m \geq n$, rows 1 to m of $\text{sub}(B)$ contain the minimum norm solution vectors; if $trans = 'T'$ and $m < n$, rows 1 to m of $\text{sub}(B)$ contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements $m+1$ to n in that column.
<i>work(1)</i>	On exit, <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -entry had an illegal value, then $info = - (i * 100 + j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

p?syev

Computes selected eigenvalues and eigenvectors of a symmetric matrix.

Syntax

```
call pssyev( jobz, uplo, n, a, ia, ja, desca, w, z, iz, jz, descz, work, lwork,
info )
```

```
call pdsyev( jobz, uplo, n, a, ia, ja, desca, w, z, iz, jz, descz, work, lwork,
info )
```

Description

This routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A* by calling the recommended sequence of ScaLAPACK routines.

In its present form, the routine assumes a homogeneous system and makes no checks for consistency of the eigenvalues or eigenvectors across the different processes. Because of this, it is possible that a heterogeneous system may return incorrect results without any error messages.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

jobz (global). CHARACTER. Must be 'N' or 'V'. Specifies if it is necessary to compute the eigenvectors:

If *jobz* = 'N', then only eigenvalues are computed.

If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

uplo (global). CHARACTER. Must be 'U' or 'L'. Specifies whether the upper or lower triangular part of the symmetric matrix *A* is stored:

If *uplo* = 'U', *a* stores the upper triangular part of *A*.

If *uplo* = 'L', *a* stores the lower triangular part of *A*.

n (global) INTEGER. The number of rows and columns of the matrix *A* ($n \geq 0$).

a (local)

REAL for pssyev.
 DOUBLE PRECISION for pdsyev.
 Block cyclic array of global dimension (n, n) and local dimension $(lld\_a, LOC\ c(ja+n-1))$. On entry, the symmetric matrix A .
 If $uplo = 'U'$, only the upper triangular part of A is used to define the elements of the symmetric matrix.
 If $uplo = 'L'$, only the lower triangular part of A is used to define the elements of the symmetric matrix.

ia, ja (global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix A , respectively.

desca (global and local) INTEGER array, dimension $(dlen\_)$. The array descriptor for the distributed matrix A .

iz, jz (global) INTEGER. The row and column indices in the global array z indicating the first row and the first column of the submatrix Z , respectively.

descz (global and local) INTEGER array, dimension $(dlen\_)$. The array descriptor for the distributed matrix Z .

work (local)
 REAL for pssyev.
 DOUBLE PRECISION for pdsyev.
 Array, DIMENSION $(lwork)$.

lwork (local) INTEGER. See below for definitions of variables used to define $lwork$.
 If no eigenvectors are requested ($jobz = 'N'$), then $lwork \geq 5*n + sizesytrd + 1$,
 where $sizesytrd$ is the workspace for `p?sytrd` and is $\max(NB*(np + 1), 3*NB)$.
 If eigenvectors are requested ($jobz = 'V'$) then the amount of workspace required to guarantee that all eigenvectors are computed is:
 $qrmem = 2*n-2$
 $lwmin = 5*n + n*ldc + \max(sizemqrleft, qrmem) + 1$
 Variable definitions:

```

nb = desca(mb_) = desca(nb_) = descz(mb_) =
descz(nb_);
nn = max(n, nb, 2);
desca(rsrc_) = desca(rsrc_) = descz(rsrc_) =
descz(csrc_) = 0
np = numroc(nn, nb, 0, 0, NPROW)
nq = numroc(max(n, nb, 2), nb, 0, 0, NPCOL)
nrc = numroc(n, nb, myprowc, 0, NPROCS)
ldc = max(1, nrc)
sizemqrleft is the workspace for p?ormtr when its side
argument is 'L'.
myprowc is defined when a new context is created as follows:
call blacs_get(desca(ctxt_), 0, contextc)
call blacs_gridinit(contextc, 'R', NPROCS, 1)
call blacs_gridinfo(contextc, nprowc, npcolc,
myprowc, mypcolc)

```

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by *pxerbla*.

Output Parameters

<i>a</i>	On exit, the lower triangle (if <i>uplo</i> ='L') or the upper triangle (if <i>uplo</i> ='U') of <i>A</i> , including the diagonal, is destroyed.
<i>w</i>	(global). REAL for <i>pssyev</i> DOUBLE PRECISION for <i>pdsyev</i> Array, DIMENSION (<i>n</i>). On normal exit, the first <i>m</i> entries contain the selected eigenvalues in ascending order.
<i>z</i>	(local). REAL for <i>pssyev</i> DOUBLE PRECISION for <i>pdsyev</i> Array, global dimension (<i>n</i> , <i>n</i>), local dimension (<i>lld_z</i> , <i>LOCc(jz+n-1)</i>). If <i>jobz</i> = 'V', then on normal exit the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues.

If *jobz* = 'N', then *z* is not referenced.

work(1) On output, *work(1)* returns the workspace needed to guarantee completion. If the input parameters are incorrect, *work(1)* may also be incorrect.

If *jobz* = 'N' *work(1)* = minimal (optimal) amount of workspace

If *jobz* = 'V' *work(1)* = minimal workspace required to generate all the eigenvectors.

info (global) INTEGER.

If *info* = 0, the execution is successful.

If *info* < 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = $-(i*100+j)$, if the *i*-th argument is a scalar and had an illegal value, then *info* = $-i$.

If *info* > 0:

If *info* = 1 through *n*, the *i*-th eigenvalue did not converge in *steqr2* after a total of $30n$ iterations.

If *info* = *n*+1, then *p?syev* has detected heterogeneity by finding that eigenvalues were not identical across the process grid. In this case, the accuracy of the results from *p?syev* cannot be guaranteed.

p?syevx

Computes selected eigenvalues and, optionally, eigenvectors of a symmetric matrix.

Syntax

```
call pssyevx(jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu, abstol,
m, nz, w, orfac, z, iz, jz, descz, work, lwork, iwork, liwork, ifail, iclustr,
gap, info)
```

```
call pdsyevx(jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu, abstol,
m, nz, w, orfac, z, iz, jz, descz, work, lwork, iwork, liwork, ifail, iclustr,
gap, info)
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix A by calling the recommended sequence of ScaLAPACK routines. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

$jobz$ (global). CHARACTER*1. Must be 'N' or 'V'. Specifies if it is necessary to compute the eigenvectors:
 If $jobz = 'N'$, then only eigenvalues are computed.
 If $jobz = 'V'$, then eigenvalues and eigenvectors are computed.

$range$ (global). CHARACTER*1. Must be 'A', 'V', or 'I'.
 If $range = 'A'$, all eigenvalues will be found.
 If $range = 'V'$, all eigenvalues in the half-open interval $[vl, vu]$ will be found.
 If $range = 'I'$, the eigenvalues with indices il through iu will be found.

$uplo$ (global). CHARACTER*1. Must be 'U' or 'L'.
 Specifies whether the upper or lower triangular part of the symmetric matrix A is stored:
 If $uplo = 'U'$, a stores the upper triangular part of A .
 If $uplo = 'L'$, a stores the lower triangular part of A .

n (global) INTEGER. The number of rows and columns of the matrix A ($n \geq 0$).

a (local). REAL for pssyevx
 DOUBLE PRECISION for pdsyevx.
 Block cyclic array of global dimension (n, n) and local dimension $(lld\_a, LOCC(ja+n-1))$. On entry, the symmetric matrix A .
 If $uplo = 'U'$, only the upper triangular part of A is used to define the elements of the symmetric matrix.
 If $uplo = 'L'$, only the lower triangular part of A is used to define the elements of the symmetric matrix.

<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>vl, vu</i>	(global) REAL for pssyevx DOUBLE PRECISION for pdsyevx. If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues; $vl \leq vu$. Not referenced if <i>range</i> = 'A' or 'I'.
<i>il, iu</i>	(global) INTEGER. If <i>range</i> = 'I', the indices of the smallest and largest eigenvalues to be returned. Constraints: $il \geq 1$ $\min(il, n) \leq iu \leq n$ Not referenced if <i>range</i> = 'A' or 'V'.
<i>abstol</i>	(global). REAL for pssyevx DOUBLE PRECISION for pdsyevx. If <i>jobz</i> ='V', setting <i>abstol</i> to $p?lamch(context, 'U')$ yields the most orthogonal eigenvectors. The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a, b]$ of width less than or equal to $abstol + eps * \max(a , b)$, where <i>eps</i> is the machine precision. If <i>abstol</i> is less than or equal to zero, then $eps * \text{norm}(T)$ will be used in its place, where $\text{norm}(T)$ is the 1-norm of the tridiagonal matrix obtained by reducing <i>A</i> to tridiagonal form. Eigenvalues will be computed most accurately when <i>abstol</i> is set to twice the underflow threshold $2 * p?lamch('S')$ not zero. If this routine returns with $((\text{mod}(info, 2) \neq 0) \text{ or } * (\text{mod}(info/8, 2) \neq 0))$, indicating that some eigenvalues or eigenvectors did not converge, try setting <i>abstol</i> to $2 * p?lamch('S')$.

<i>orfac</i>	(global). REAL for pssyevx DOUBLE PRECISION for pdsyevx. Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within $tol=orfac*norm(A)$ of each other are to be reorthogonalized. However, if the workspace is insufficient (see <i>lwork</i>), <i>tol</i> may be decreased until all eigenvectors to be reorthogonalized can be stored in one process. No reorthogonalization will be done if <i>orfac</i> equals zero. A default value of 1.0e-3 is used if <i>orfac</i> is negative. <i>orfac</i> should be identical on all processes.
<i>iz, jz</i>	(global) INTEGER. The row and column indices in the global array <i>z</i> indicating the first row and the first column of the submatrix <i>z</i>, respectively.
<i>descz</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>z</i>. <i>descz(ctxt_)</i> must equal <i>desca(ctxt_)</i>.
<i>work</i>	(local) REAL for pssyevx. DOUBLE PRECISION for pdsyevx. Array, DIMENSION (<i>lwork</i>).
<i>lwork</i>	(local) INTEGER. The dimension of the array <i>work</i>. See below for definitions of variables used to define <i>lwork</i>. If no eigenvectors are requested (<i>jobz</i> = 'N'), then <i>lwork</i> $\geq 5*n + \max(5*nn, NB*(np0 + 1))$. If eigenvectors are requested (<i>jobz</i> = 'V'), then the amount of workspace required to guarantee that all eigenvectors are computed is: $lwork \geq 5*n + \max(5*nn, np0*mq0 + 2*NB*NB) +$ $iceil(neig, NPROW*NPCOL)*nn$ The computed eigenvectors may not be orthogonal if the minimal workspace is supplied and <i>orfac</i> is too small. If you want to guarantee orthogonality (at the cost of potentially poor performance) you should add the following to <i>lwork</i>: $(clustersize-1)*n,$

where *clustersize* is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:

$$\{w(k), \dots, w(k+clustersize-1) \mid w(j+1) \leq w(j)\} + orfac*2*norm(A),$$

where

neig = number of eigenvectors requested

nb = *desca*(*mb_*) = *desca*(*nb_*) = *descz*(*mb_*) = *descz*(*nb_*);

nn = max(*n*, *nb*, 2);

desca(*rsrc_*) = *desca*(*nb_*) = *descz*(*rsrc_*) = *descz*(*csrc_*) = 0;

np0 = numroc(*nn*, *nb*, 0, 0, NPROW);

mq0 = numroc(max(*neig*, *nb*, 2), *nb*, 0, 0, NPCOL)

iceil(*x*, *y*) is a ScaLAPACK function returning ceiling(*x/y*)

If *lwork* is too small to guarantee orthogonality, *p?syevx* attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues.

If *lwork* is too small to compute all the eigenvectors requested, no computation is performed and *info*= -23 is returned.

Note that when *range*='V', number of requested eigenvectors are not known until the eigenvalues are computed. In this case and if *lwork* is large enough to compute the eigenvalues, *p?sygvx* computes the eigenvalues and as many eigenvectors as possible.

Relationship between workspace, orthogonality & performance:

Greater performance can be achieved if adequate workspace is provided. In some situations, performance can decrease as the provided workspace increases above the workspace amount shown below:

$$lwork \geq \max(lwork, 5*n + nsytrd_lwopt),$$

where *lwork*, as defined previously, depends upon the number of eigenvectors requested, and

$$nsytrd\_lwopt = n + 2*(anb+1)*(4*nps+2) + (nps + 3)*nps;$$

```
anb = pjlaenv(desca(ctxt_), 3, 'p?sytttrd', 'L',
0, 0, 0, 0);
```

```
sqnpc = int(sqrt(dble(NPROW * NPCOL)));
```

```
nps = max(numroc(n, 1, 0, 0, sqnpc), 2*anb);
```

numroc is a ScaLAPACK tool functions;

pjlaenv is a ScaLAPACK environmental inquiry function

MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

For large n , no extra workspace is needed, however the biggest boost in performance comes for small n , so it is wise to provide the extra workspace (typically less than a megabyte per process).

If $clustersize > n/\sqrt{NPROW \cdot NPCOL}$, then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. At the limit (that is, $clustersize = n-1$) p?stein will perform no better than ?stein on single processor.

For $clustersize = n/\sqrt{NPROW \cdot NPCOL}$

reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.

For $clustersize > n/\sqrt{NPROW \cdot NPCOL}$ execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by p?erbla.

iwork

(local) INTEGER. Workspace array.

liwork

(local) INTEGER, dimension of *iwork*. $liwork \geq 6 \cdot nnp$

Where: $nnp = \max(n, NPROW \cdot NPCOL + 1, 4)$

If $liwork = -1$, then $liwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of

these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<i>a</i>	On exit, the lower triangle (if <i>uplo</i> = 'L') or the upper triangle (if <i>uplo</i> = 'U') of <i>A</i> , including the diagonal, is overwritten.
<i>m</i>	(global) INTEGER. The total number of eigenvalues found; $0 \leq m \leq n$.
<i>nz</i>	(global) INTEGER. Total number of eigenvectors computed. $0 \leq nz \leq m$. The number of columns of <i>z</i> that are filled. If <i>jobz</i> $\neq$ 'V', <i>nz</i> is not referenced. If <i>jobz</i> = 'V', <i>nz</i> = <i>m</i> unless the user supplies insufficient space and <code>p?syevx</code> is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in <i>z</i> (<i>m.le.descz(n_)</i>) and sufficient workspace to compute them. (See <i>lwork</i>). <code>p?syevx</code> is always able to detect insufficient space without computation unless <i>range.eq.</i> 'V'.
<i>w</i>	(global). REAL for <code>pssyevx</code> DOUBLE PRECISION for <code>pdsyevx</code> . Array, DIMENSION (<i>n</i>). The first <i>m</i> elements contain the selected eigenvalues in ascending order.
<i>z</i>	(local). REAL for <code>pssyevx</code> DOUBLE PRECISION for <code>pdsyevx</code> . Array, global dimension (<i>n</i> , <i>n</i>), local dimension (<i>lld_z</i> , <i>LOCc(jz+n-1)</i>). If <i>jobz</i> = 'V', then on normal exit the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of <i>z</i> contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in <i>ifail</i> .

	If <i>jobz</i> = 'N', then <i>z</i> is not referenced.
<i>work</i> (1)	On exit, returns workspace adequate workspace to allow optimal performance.
<i>iwork</i> (1)	On return, <i>iwork</i> (1) contains the amount of integer workspace required.
<i>ifail</i>	(global) INTEGER. Array, DIMENSION (<i>n</i>). If <i>jobz</i> = 'V', then on normal exit, the first <i>m</i> elements of <i>ifail</i> are zero. If (mod(<i>info</i> ,2) .ne.0) on exit, then <i>ifail</i> contains the indices of the eigenvectors that failed to converge. If <i>jobz</i> = 'N', then <i>ifail</i> is not referenced.
<i>iclustr</i>	(global) INTEGER. Array, DIMENSION (2*NPROW*NPCOL) This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see <i>lwork</i> , <i>orfac</i> and <i>info</i>). Eigenvectors corresponding to clusters of eigenvalues indexed <i>iclustr</i> (2*i-1) to <i>iclustr</i> (2*i), could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal. <i>iclustr</i> () is a zero terminated array. (<i>iclustr</i> (2*k).ne.0. and. <i>iclustr</i> (2*k+1).eq.0) if and only if <i>k</i> is the number of clusters. <i>iclustr</i> is not referenced if <i>jobz</i> = 'N'.
<i>gap</i>	(global) REAL for pssyevx DOUBLE PRECISION for pdsyevx. Array, DIMENSION (NPROW*NPCOL) This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array <i>iclustr</i> . As a result, the dot product between eigenvectors corresponding to the <i>i</i> th cluster may be as high as (C*n)/ <i>gap</i> (<i>i</i>) where <i>C</i> is a small constant.
<i>info</i>	(global) INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> < 0:

If the i -th argument is an array and the j -entry had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

If $info > 0$: if $(\text{mod}(info,2) \neq 0)$, then one or more eigenvectors failed to converge. Their indices are stored in $ifail$. Ensure $abstol=2.0*p?lamch('U')$.

If $(\text{mod}(info/2,2) \neq 0)$, then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace. The indices of the clusters are stored in the array $iclustr$.

If $(\text{mod}(info/4,2) \neq 0)$, then space limit prevented $p?syevxf$ from computing all of the eigenvectors between vl and vu . The number of eigenvectors computed is returned in nz .

If $(\text{mod}(info/8,2) \neq 0)$, then $p?stebz$ failed to compute eigenvalues. Ensure $abstol=2.0*p?lamch('U')$.

p?heevx

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix.

Syntax

```
call pcheevx( jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu, abstol,
m, nz, w, orfac, z, iz, jz, descz, work, lwork, rwork, lrwork, iwork, liwork,
ifail, iclustr, gap, info )
```

```
call pzheevx( jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu, abstol,
m, nz, w, orfac, z, iz, jz, descz, work, lwork, rwork, lrwork, iwork, liwork,
ifail, iclustr, gap, info )
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A by calling the recommended sequence of ScaLAPACK routines. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

<i>jobz</i>	(global). CHARACTER*1. Must be 'N' or 'V'. Specifies if it is necessary to compute the eigenvectors: If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	(global). CHARACTER*1. Must be 'A', 'V', or 'I'. If <i>range</i> = 'A', all eigenvalues will be found. If <i>range</i> = 'V', all eigenvalues in the half-open interval $[vl, vu]$ will be found. If <i>range</i> = 'I', the eigenvalues with indices <i>il</i> through <i>iu</i> will be found.
<i>uplo</i>	(global). CHARACTER*1. Must be 'U' or 'L'. Specifies whether the upper or lower triangular part of the Hermitian matrix <i>A</i> is stored: If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	(global) INTEGER. The number of rows and columns of the matrix <i>A</i> ($n \geq 0$).
<i>a</i>	(local). COMPLEX for pcheevx DOUBLE COMPLEX for pzheevx. Block cyclic array of global dimension (<i>n</i> , <i>n</i>) and local dimension (<i>lld_a</i> , <i>LOC c(ja+n-1)</i>). On entry, the Hermitian matrix <i>A</i> . If <i>uplo</i> = 'U', only the upper triangular part of <i>A</i> is used to define the elements of the symmetric matrix. If <i>uplo</i> = 'L', only the lower triangular part of <i>A</i> is used to define the elements of the Hermitian matrix.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.

<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> . If <i>desca(ctxt_)</i> is incorrect, <i>p?heevx</i> cannot guarantee correct error reporting
<i>vl, vu</i>	(global) REAL for <i>pcheevx</i> DOUBLE PRECISION for <i>pzheevx</i> . If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues; not referenced if <i>range</i> = 'A' or 'I'.
<i>il, iu</i>	(global) INTEGER. If <i>range</i> = 'I', the indices of the smallest and largest eigenvalues to be returned. Constraints: $il \geq 1; \min(il, n) \leq iu \leq n$. Not referenced if <i>range</i> = 'A' or 'V'.
<i>abstol</i>	(global). REAL for <i>pcheevx</i> DOUBLE PRECISION for <i>pzheevx</i> . If <i>jobz</i> ='V', setting <i>abstol</i> to <i>p?lamch(context, 'U')</i> yields the most orthogonal eigenvectors. The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a, b]$ of width less than or equal to $abstol + eps * \max(a , b)$, where <i>eps</i> is the machine precision. If <i>abstol</i> is less than or equal to zero, then $eps * \text{norm}(T)$ will be used in its place, where $\text{norm}(T)$ is the 1-norm of the tridiagonal matrix obtained by reducing <i>A</i> to tridiagonal form. Eigenvalues are computed most accurately when <i>abstol</i> is set to twice the underflow threshold $2 * p?lamch('S')$, not zero. If this routine returns with $((\text{mod}(\text{info}, 2) \neq 0) \text{ or } (\text{mod}(\text{info}/8, 2) \neq 0))$, indicating that some eigenvalues or eigenvectors did not converge, try setting <i>abstol</i> to $2 * p?lamch('S')$.
<i>orfac</i>	(global). REAL for <i>pcheevx</i> DOUBLE PRECISION for <i>pzheevx</i> .

	Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within $tol = orfac * \text{norm}(A)$ of each other are to be reorthogonalized. However, if the workspace is insufficient (see <i>lwork</i>), <i>tol</i> may be decreased until all eigenvectors to be reorthogonalized can be stored in one process. No reorthogonalization will be done if <i>orfac</i> equals zero. A default value of 1.0e-3 is used if <i>orfac</i> is negative. <i>orfac</i> should be identical on all processes.
<i>iz, jz</i>	(global) INTEGER. The row and column indices in the global array <i>z</i> indicating the first row and the first column of the submatrix <i>Z</i> , respectively.
<i>descz</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>Z</i> . <i>descz</i> (<i>ctxt_</i>) must equal <i>desca</i> (<i>ctxt_</i>).
<i>work</i>	(local) COMPLEX for <i>pcheevx</i> DOUBLE COMPLEX for <i>pzheevx</i> . Array, DIMENSION (<i>lwork</i>).
<i>lwork</i>	(local). INTEGER. The dimension of the array <i>work</i> . If only eigenvalues are requested: $lwork \geq n + \max(nb * (np0 + 1), 3)$ If eigenvectors are requested: $lwork \geq n + (np0 + mq0 + nb) * nb$ with $nq0 = \text{numroc}(nn, nb, 0, 0, \text{NPCOL})$. $lwork \geq 5 * n + \max(5 * nn, np0 * mq0 + 2 * nb * nb) + \text{iceil}(neig, \text{NPROW} * \text{NPCOL}) * nn$ For optimal performance, greater workspace is needed, that is $lwork \geq \max(lwork, nhetrd\_lwork)$ where <i>lwork</i> is as defined above, and <i>nhetrd_lwork</i> = $n + 2 * (anb + 1) * (4 * nps + 2) + (nps + 1) * nps$ $ictxt = \text{desca}(ctxt\_)$ $anb = \text{pjlavenv}(ictxt, 3, 'pchettrd', 'L', 0, 0, 0, 0)$ $sqnpc = \text{sqrt}(\text{dble}(\text{NPROW} * \text{NPCOL}))$

```

nps = max(numroc(n, 1, 0, 0, sqnpc), 2*anb)
If lwork = -1, then lwork is global input and a workspace
query is assumed; the routine only calculates the size
required for optimal performance for all work arrays. Each
of these values is returned in the first entry of the
corresponding work arrays, and no error message is issued
by pxxerbla.

rwork      (local)
            REAL for pcheevx
            DOUBLE PRECISION for pzheevx.
            Workspace array, DIMENSION (lrwork).

lrwork     (local) INTEGER. The dimension of the array work.
            See below for definitions of variables used to define lrwork.
            If no eigenvectors are requested (jobz = 'N'), then
            lrwork ≥ 5*nn+4*n.
            If eigenvectors are requested (jobz = 'V'), then the
            amount of workspace required to guarantee that all
            eigenvectors are computed is:
            lrwork ≥ 4*n + max(5*nn, np0*mq0+2*nb*nb) +
            iceil(neig, NPROW*NPCOL)*nn
            The computed eigenvectors may not be orthogonal if the
            minimal workspace is supplied and orfac is too small. If
            you want to guarantee orthogonality (at the cost of
            potentially poor performance) you should add the following
            values to lrwork:
            (clustersize-1)*n,
            where clustersize is the number of eigenvalues in the
            largest cluster, where a cluster is defined as a set of close
            eigenvalues:
            {w(k), ..., w(k+clustersize-1) | w(j+1) ≤
            w(j)+orfac*2*norm(A)}.
            Variable definitions:
            neig = number of eigenvectors requested;
            nb = desca(mb_) = desca(nb_) = descz(mb_) =
            descz(nb_);
            nn = max(n, NB, 2);

```

```
desca(rsrc_) = desca(nb_) = descz(rsrc_) =
descz(csrc_) = 0;
np0 = numroc(nn, nb, 0, 0, NPROW);
mq0 = numroc(max(neig, nb, 2), nb, 0, 0, NPCOL);
iceil(x, y) is a ScaLAPACK function returning ceiling(x/y)
When lwork is too small:
```

If *lwork* is too small to guarantee orthogonality, p?heevx attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues. If *lwork* is too small to compute all the eigenvectors requested, no computation is performed and *info*= -23 is returned. Note that when *range*='V', p?heevx does not know how many eigenvectors are requested until the eigenvalues are computed. Therefore, when *range*='V' and as long as *lwork* is large enough to allow p?heevx to compute the eigenvalues, p?heevx will compute the eigenvalues and as many eigenvectors as it can.

Relationship between workspace, orthogonality and performance:

If $clustersize \geq n/\sqrt{NPROW \cdot NPCOL}$, then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. In the limit (that is, $clustersize = n-1$) p?stein will perform no better than ?stein on 1 processor.

For $clustersize = n/\sqrt{NPROW \cdot NPCOL}$ reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.

For $clustersize > n/\sqrt{NPROW \cdot NPCOL}$ execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by p?xerbla.

iwork (local) INTEGER. Workspace array.
liwork (local) INTEGER, dimension of *iwork*.
 $liwork \geq 6 * nnp$
 Where: $nnp = \max(n, NPROW * NPCOL + 1, 4)$
 If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by p_xerbla.

Output Parameters

a On exit, the lower triangle (if *uplo* = 'L'), or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.

m (global) INTEGER. The total number of eigenvalues found;
 $0 \leq m \leq n$.

nz (global) INTEGER. Total number of eigenvectors computed.
 $0 \leq nz \leq m$.
 The number of columns of *z* that are filled.
 If *jobz* ≠ 'V', *nz* is not referenced.
 If *jobz* = 'V', *nz* = *m* unless the user supplies insufficient space and p_?heevx is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in *z* (*m.le.descz(n_)*) and sufficient workspace to compute them. (See *lwork*). p_?heevx is always able to detect insufficient space without computation unless *range.eq.'V'*.

w (global).
 REAL for pcheevx
 DOUBLE PRECISION for pzheevx.
 Array, DIMENSION (*n*). The first *m* elements contain the selected eigenvalues in ascending order.

z (local).

COMPLEX for pcheevx
DOUBLE COMPLEX for pzheevx.
Array, global dimension (n, n), local dimension ($l1d\_z, LOcc(jz+n-1)$).
If $jobz = 'V'$, then on normal exit the first m columns of z contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in $ifail$.
If $jobz = 'N'$, then z is not referenced.

work(1) On exit, returns workspace adequate workspace to allow optimal performance.

rwork (local).
REAL for pcheevx
DOUBLE PRECISION for pzheevx.
Array, DIMENSION ($lrwork$). On return, $rwork(1)$ contains the optimal amount of workspace required for efficient execution.
If $jobz='N'$ $rwork(1)$ = optimal amount of workspace required to compute eigenvalues efficiently.
If $jobz='V'$ $rwork(1)$ = optimal amount of workspace required to compute eigenvalues and eigenvectors efficiently with no guarantee on orthogonality.
If $range='V'$, it is assumed that all eigenvectors may be required.

iwork(1) (local)
On return, $iwork(1)$ contains the amount of integer workspace required.

ifail (global) INTEGER.
Array, DIMENSION (n).
If $jobz = 'V'$, then on normal exit, the first m elements of $ifail$ are zero. If $(mod(info, 2) \neq 0)$ on exit, then $ifail$ contains the indices of the eigenvectors that failed to converge.
If $jobz = 'N'$, then $ifail$ is not referenced.

iclustr (global) INTEGER.
Array, DIMENSION ($2*NPROW*NPCOL$).

This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see *lwork*, *orfac* and *info*). Eigenvectors corresponding to clusters of eigenvalues indexed *iclustr*(2*i-1) to *iclustr*(2*i), could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal. *iclustr*() is a zero terminated array. (*iclustr*(2*k).ne.0. and. *iclustr*(2*k+1).eq.0) if and only if *k* is the number of clusters. *iclustr* is not referenced if *jobz* = 'N'.

gap

(global)

REAL for pcheevx

DOUBLE PRECISION for pzheevx.

Array, DIMENSION (NPROW*NPCOL)

This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the *i*-th cluster may be as high as $(C*n)/gap(i)$ where *C* is a small constant.

info

(global) INTEGER.

If *info* = 0, the execution is successful.

If *info* < 0:

If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(i*100+j), if the *i*-th argument is a scalar and had an illegal value, then *info* = -i.

If *info* > 0:

If (mod(*info*,2).ne.0), then one or more eigenvectors failed to converge. Their indices are stored in *ifail*. Ensure *abstol*=2.0*p?lamch('U')

If (mod(*info*/2,2).ne.0), then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace. The indices of the clusters are stored in the array *iclustr*.

If $(\text{mod}(\text{info}/4, 2) \neq 0)$, then space limit prevented `p?syevx` from computing all of the eigenvectors between `vl` and `vu`. The number of eigenvectors computed is returned in `nz`.

If $(\text{mod}(\text{info}/8, 2) \neq 0)$, then `p?stebz` failed to compute eigenvalues. Ensure `abstol=2.0*p?lamch('U')`.

p?gesvd

Computes the singular value decomposition of a general matrix, optionally computing the left and/or right singular vectors.

Syntax

```
call psgesvd( jobu, jobvt, m, n, a, ia, ja, desca, s, u, iu, ju, descu, vt,
  ivt, jvt, descvt, work, lwork, info )

call pdgesvd( jobu, jobvt, m, n, a, ia, ja, desca, s, u, iu, ju, descu, vt,
  ivt, jvt, descvt, work, lwork, info )

call pcgesvd( jobu, jobvt, m, n, a, ia, ja, desca, s, u, iu, ju, descu, vt,
  ivt, jvt, descvt, work, lwork, rwork, info )

call pzgesvd( jobu, jobvt, m, n, a, ia, ja, desca, s, u, iu, ju, descu, vt,
  ivt, jvt, descvt, work, lwork, rwork, info )
```

Description

This routine computes the singular value decomposition (SVD) of an m -by- n matrix A , optionally computing the left and/or right singular vectors. The SVD is written

$$A = U \Sigma V^T,$$

where Σ is an m -by- n matrix that is zero except for its $\min(m, n)$ diagonal elements, U is an m -by- m orthogonal matrix, and V is an n -by- n orthogonal matrix. The diagonal elements of Σ are the singular values of A and the columns of U and V are the corresponding right and left singular vectors, respectively. The singular values are returned in array `s` in decreasing order and only the first $\min(m, n)$ columns of U and rows of `vt` = V^T are computed.

Input Parameters

`mp` = number of local rows in A and U

nq = number of local columns in A and VT

$size = \min(m, n)$

$sizeq$ = number of local columns in U

$sizep$ = number of local rows in VT

jobu (global). CHARACTER*1. Specifies options for computing all or part of the matrix U .

If *jobu* = 'V', the first *size* columns of U (the left singular vectors) are returned in the array *u*;

If *jobu* = 'N', no columns of U (no left singular vectors) are computed.

jobvt (global) CHARACTER*1.

Specifies options for computing all or part of the matrix V^T .

If *jobvt* = 'V', the first *size* rows of V^T (the right singular vectors) are returned in the array *vt*;

If *jobvt* = 'N', no rows of V^T (no right singular vectors) are computed.

m (global) INTEGER. The number of rows of the matrix A ($m \geq 0$).

n (global) INTEGER. The number of columns in A ($n \geq 0$).

a (local). REAL for *psgesvd*
DOUBLE PRECISION for *pdgesvd*
COMPLEX for *pcgesvd*
COMPLEX*16 for *pzgesvd*
Block cyclic array, global dimension (m, n), local dimension (mp, nq).

work(lwork) is a workspace array.

ia, ja (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix A , respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix A .

iu, ju (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix U , respectively.

<i>descu</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>U</i> .
<i>ivt, jvt</i>	(global) INTEGER. The row and column indices in the global array <i>vt</i> indicating the first row and the first column of the submatrix <i>VT</i> , respectively.
<i>descvt</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>VT</i> .
<i>work</i>	(local). REAL for <i>psgesvd</i> DOUBLE PRECISION for <i>pdgesvd</i> COMPLEX for <i>pcgesvd</i> COMPLEX*16 for <i>pzgesvd</i> Workspace array, dimension (<i>lwork</i>)
<i>lwork</i>	(local) INTEGER. The dimension of the array <i>work</i> ; $lwork > 2 + 6 * sizeb + \max(watobd, wbdtosvd)$, where $sizeb = \max(m, n)$, and <i>watobd</i> and <i>wbdtosvd</i> refer, respectively, to the workspace required to bidiagonalize the matrix <i>A</i> and to go from the bidiagonal matrix to the singular value decomposition $U S VT$. For <i>watobd</i> , the following holds: $watobd = \max(\max(wp?lange, wp?gebrd), \max(wp?lared2d, wp?lared1d))$, where <i>wp?lange</i> , <i>wp?lared1d</i> , <i>wp?lared2d</i> , <i>wp?gebrd</i> are the workspaces required respectively for the subprograms <i>p?lange</i> , <i>p?lared1d</i> , <i>p?lared2d</i> , <i>p?gebrd</i> . Using the standard notation $mp = \text{numroc}(m, mb, MYROW, \text{desca}(ctxt\_), NPROW)$, $nq = \text{numroc}(n, nb, MYCOL, \text{desca}(lld\_), NPCOL)$, the workspaces required for the above subprograms are $wp?lange = mp$, $wp?lared1d = nq0$, $wp?lared2d = mp0$, $wp?gebrd = nb * (mp + nq + 1) + nq$, where <i>nq0</i> and <i>mp0</i> refer, respectively, to the values obtained at <i>MYCOL</i> = 0 and <i>MYROW</i> = 0. In general, the upper limit for the workspace is given by a workspace required on processor (0,0): $watobd \leq nb * (mp0 + nq0 + 1) + nq0$.

In case of a homogeneous process grid this upper limit can be used as an estimate of the minimum workspace for every processor.

For *wbdtosvd*, the following holds:

$$wbdtosvd = size * (wantu * nru + wantvt * ncvt) + \max(w?bdsqr, \max(wantu * wp?ormbrqln, wantvt * wp?ormbrprt)),$$

where

$wantu(wantvt) = 1$, if left/right singular vectors are wanted, and $wantu(wantvt) = 0$, otherwise. *w?bdsqr*, *wp?ormbrqln*, and *wp?ormbrprt* refer respectively to the workspace required for the subprograms *?bdsqr*, *p?ormbr(qln)*, and *p?ormbr(prt)*, where *qln* and *prt* are the values of the arguments *vect*, *side*, and *trans* in the call to *p?ormbr*. *nru* is equal to the local number of rows of the matrix *U* when distributed 1-dimensional "column" of processes. Analogously, *ncvt* is equal to the local number of columns of the matrix *VT* when distributed across 1-dimensional "row" of processes. Calling the LAPACK procedure *?bdsqr* requires

$$w?bdsqr = \max(1, 2 * size + (2 * size - 4) * \max(wantu, wantvt))$$

on every processor. Finally,

$$\begin{aligned} wp?ormbrqln &= \max((nb * (nb - 1)) / 2, \\ &\quad (sizeq + mp) * nb) + nb * nb, \\ wp?ormbrprt &= \max((mb * (mb - 1)) / 2, \\ &\quad (sizep + nq) * mb) + mb * mb, \end{aligned}$$

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum size for the work array. The required workspace is returned as the first element of *work* and no error message is issued by *pxerbla*.

rwork

REAL for *psgesvd*
 DOUBLE PRECISION for *pdgesvd*
 COMPLEX for *pcgesvd*
 COMPLEX*16 for *pzgesvd*
 Workspace array, dimension $(1 + 4 * sizeb)$

Output Parameters

<i>a</i>	On exit, the contents of <i>a</i> are destroyed.
<i>s</i>	(global). REAL for psgesvd DOUBLE PRECISION for pdgesvd COMPLEX for pcgesvd COMPLEX*16 for pzgesvd Array, DIMENSION (<i>size</i>). Contains the singular values of <i>A</i> sorted so that $s(i) \geq s(i+1)$.
<i>u</i>	(local). REAL for psgesvd DOUBLE PRECISION for pdgesvd COMPLEX for pcgesvd COMPLEX*16 for pzgesvd local dimension (<i>mp</i> , <i>sizeq</i>), global dimension (<i>m</i> , <i>size</i>) If <i>jobu</i> = 'V', <i>u</i> contains the first min(<i>m</i> , <i>n</i>) columns of <i>U</i> . If <i>jobu</i> = 'N' or 'O', <i>u</i> is not referenced.
<i>vt</i>	(local). REAL for psgesvd DOUBLE PRECISION for pdgesvd COMPLEX for pcgesvd COMPLEX*16 for pzgesvd local dimension (<i>sizep</i> , <i>nq</i>), global dimension (<i>size</i> , <i>n</i>) If <i>jobvt</i> = 'V', <i>vt</i> contains the first <i>size</i> rows of V^T if <i>jobu</i> = 'N', <i>vt</i> is not referenced.
<i>work</i>	On exit, if <i>info</i> = 0, then <i>work</i> (1) returns the required minimal size of <i>lwork</i> .
<i>rwork</i>	On exit, if <i>info</i> = 0, then <i>rwork</i> (1) returns the required size of <i>rwork</i> .
<i>info</i>	(global) INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> < 0, If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> > 0 <i>i</i> , then if ?bdsqr did not converge, If <i>info</i> = min(<i>m</i> , <i>n</i>) + 1, then p?gesvd has detected heterogeneity by finding that eigenvalues were not identical across the process grid. In this case, the accuracy of the results from p?gesvd cannot be guaranteed.

p?sygvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem.

Syntax

```
call pssygvx(ibtype, jobz, range, uplo, n, a, ia, ja, desca, b, ib, jb, descb,
             vl, vu, il, iu, abstol, m, nz, w, orfac, z, iz, jz, descz, work, lwork, iwork,
             liwork, ifail, iclustr, gap, info)
```

```
call pdsygvx(ibtype, jobz, range, uplo, n, a, ia, ja, desca, b, ib, jb, descb,
             vl, vu, il, iu, abstol, m, nz, w, orfac, z, iz, jz, descz, work, lwork, iwork,
             liwork, ifail, iclustr, gap, info)
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$\text{sub}(A) * x = \lambda * \text{sub}(B) * x, \text{ sub}(A) \text{ sub}(B) * x = \lambda * x, \text{ or } \text{sub}(B) * \text{sub}(A) * x = \lambda * x.$$

Here x denotes eigen vectors, λ (*lambda*) denotes eigenvalues, $\text{sub}(A)$ denoting $A(ia:ia+n-1, ja:ja+n-1)$ is assumed to symmetric, and $\text{sub}(B)$ denoting $B(ib:ib+n-1, jb:jb+n-1)$ is also positive definite.

Input Parameters

<i>ibtype</i>	(global) INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: If <i>ibtype</i> = 1, the problem type is $\text{sub}(A) * x = \lambda * \text{sub}(B) * x$; If <i>ibtype</i> = 2, the problem type is $\text{sub}(A) * \text{sub}(B) * x = \lambda * x$; If <i>ibtype</i> = 3, the problem type is $\text{sub}(B) * \text{sub}(A) * x = \lambda * x$.
<i>jobz</i>	(global). CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>range</i>	(global). CHARACTER*1. Must be 'A' or 'V' or 'I'.

	If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues in the interval: [<i>vl</i>, <i>vu</i>] If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> through <i>iu</i>.
<i>uplo</i>	(global). CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>a</i> and <i>b</i> store the upper triangles of sub(<i>A</i>) and sub(<i>B</i>); If <i>uplo</i> = 'L', arrays <i>a</i> and <i>b</i> store the lower triangles of sub(<i>A</i>) and sub(<i>B</i>).
<i>n</i>	(global). INTEGER. The order of the matrices sub(<i>A</i>) and sub(<i>B</i>), $n \geq 0$.
<i>a</i>	(local) REAL for pssygvx DOUBLE PRECISION for pdsygvx. Pointer into the local memory to an array of dimension (<i>lld_a</i>, <i>LOCc(ja+n-1)</i>). On entry, this array contains the local pieces of the <i>n</i>-by-<i>n</i> symmetric distributed matrix sub(<i>A</i>). If <i>uplo</i> = 'U', the leading <i>n</i>-by-<i>n</i> upper triangular part of sub(<i>A</i>) contains the upper triangular part of the matrix. If <i>uplo</i> = 'L', the leading <i>n</i>-by-<i>n</i> lower triangular part of sub(<i>A</i>) contains the lower triangular part of the matrix.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i>, respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i>. If <i>desca(ctxt_)</i> is incorrect, p?sygvx cannot guarantee correct error reporting.
<i>b</i>	(local). REAL for pssygvx DOUBLE PRECISION for pdsygvx. Pointer into the local memory to an array of dimension (<i>lld_b</i>, <i>LOCc(jb+n-1)</i>). On entry, this array contains the local pieces of the <i>n</i>-by-<i>n</i> symmetric distributed matrix sub(<i>B</i>).

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *sub(B)* contains the upper triangular part of the matrix.
 If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *sub(A)* contains the lower triangular part of the matrix.

ib, jb (global) INTEGER. The row and column indices in the global array *b* indicating the first row and the first column of the submatrix *B*, respectively.

descb (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*. *descb(ctxt_)* must be equal to *desca(ctxt_)*.

vl, vu (global)
 REAL for pssygvx
 DOUBLE PRECISION for pdsygvx.
 If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.
 If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, iu (global)
 INTEGER.
 If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint:
 $il \geq 1, \min(il, n) \leq iu \leq n$
 If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol (global)
 REAL for pssygvx
 DOUBLE PRECISION for pdsygvx.
 If *jobz*='V', setting *abstol* to *p?lamch(context, 'U')* yields the most orthogonal eigenvectors.
 The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a, b]$ of width less than or equal to
 $abstol + eps * \max(|a|, |b|)$,
 where *eps* is the machine precision. If *abstol* is less than or equal to zero, then *eps*norm(T)* will be used in its place, where *norm(T)* is the 1-norm of the tridiagonal matrix obtained by reducing *A* to tridiagonal form.

	Eigenvalues will be computed most accurately when <i>abstol</i> is set to twice the underflow threshold $2 * p?lamch('S')$ not zero. If this routine returns with $((mod(info,2).ne.0).or.*(mod(info/8,2).ne.0))$, indicating that some eigenvalues or eigenvectors did not converge, try setting <i>abstol</i> to $2 * p?lamch('S')$.
<i>orfac</i>	(global). REAL for pssygvx DOUBLE PRECISION for pdsygvx. Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within $tol = orfac * norm(A)$ of each other are to be reorthogonalized. However, if the workspace is insufficient (see <i>lwork</i>), <i>tol</i> may be decreased until all eigenvectors to be reorthogonalized can be stored in one process. No reorthogonalization will be done if <i>orfac</i> equals zero. A default value of 1.0e-3 is used if <i>orfac</i> is negative. <i>orfac</i> should be identical on all processes.
<i>iz, jz</i>	(global) INTEGER. The row and column indices in the global array <i>z</i> indicating the first row and the first column of the submatrix <i>z</i>, respectively.
<i>descz</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>z</i>. <i>descz(ctxt_)</i> must equal <i>desca(ctxt_)</i>.
<i>work</i>	(local) REAL for pssygvx DOUBLE PRECISION for pdsygvx. Workspace array, dimension (<i>lwork</i>)
<i>lwork</i>	(local) INTEGER. Dimension of the array <i>work</i>. See below for definitions of variables used to define <i>lwork</i>. If no eigenvectors are requested (<i>jobz</i> = 'N'), then <i>lwork</i> $\geq 5 * n + \max(5 * nn, NB * (np0 + 1))$. If eigenvectors are requested (<i>jobz</i> = 'V'), then the amount of workspace required to guarantee that all eigenvectors are computed is:

```

lwork ≥ 5*n + max(5*nn, np0*mq0 + 2*nb*nb) +
iceil(neig, NPROW*NPCOL)*nn.

```

The computed eigenvectors may not be orthogonal if the minimal workspace is supplied and *orfac* is too small. If you want to guarantee orthogonality at the cost of potentially poor performance you should add the following to *lwork*:

```
(clustersize-1)*n,
```

where *clustersize* is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:

```
{w(k), ..., w(k+clustersize-1) | w(j+1) ≤ w(j) +
orfac*2*norm(A) }
```

Variable definitions:

neig = number of eigenvectors requested,

```
nb = desca(mb_) = desca(nb_) = descz(mb_) =
descz(nb_),
```

```
nn = max(n, nb, 2),
```

```
desca(rsrc_) = desca(nb_) = descz(rsrc_) =
descz(csrc_) = 0,
```

```
np0 = numroc(nn, nb, 0, 0, NPROW),
```

```
mq0 = numroc(max(neig, nb, 2), nb, 0, 0, NPCOL)
```

iceil(*x*, *y*) is a ScaLAPACK function returning $\lceil x/y \rceil$

If *lwork* is too small to guarantee orthogonality, *p?syevx* attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues.

If *lwork* is too small to compute all the eigenvectors requested, no computation is performed and *info*= -23 is returned.

Note that when *range*='V', number of requested eigenvectors are not known until the eigenvalues are computed. In this case and if *lwork* is large enough to compute the eigenvalues, *p?sygvx* computes the eigenvalues and as many eigenvectors as possible. Greater performance can be achieved if adequate workspace is provided. In some situations, performance can decrease as the provided workspace increases above the workspace amount shown below:

```

lwork ≥ max(lwork, 5*n + nsytrd_lwopt,
nsygst_lwopt), where
lwork, as defined previously, depends upon the number of
eigenvectors requested, and
nsytrd_lwopt = n + 2*(anb+1)*(4*nps+2) +
(nps+3)*nps
nsygst_lwopt = 2*np0*nb + nq0*nb + nb*nb
anb = pjlalenv(desca(ctxt_), 3, p?sytrd 'L',
0, 0, 0, 0)
sqnpc = int(sqrt(dble(NPROW * NPCOL)))
nps = max(numroc(n, 1, 0, 0, sqnpc), 2*anb)
NB = desca(mb_)
np0 = numroc(n, nb, 0, 0, NPROW)
nq0 = numroc(n, nb, 0, 0, NPCOL)
numroc is a ScaLAPACK tool functions;

```

pjlalenv is a ScaLAPACK environmental inquiry function
MYROW, MYCOL, NPROW and NPCOL can be determined by
calling the subroutine blacs_gridinfo.

For large n , no extra workspace is needed, however the
biggest boost in performance comes for small n , so it is wise
to provide the extra workspace (typically less than a
Megabyte per process).

If $clustersize \geq n/\sqrt{NPROW \cdot NPCOL}$, then providing
enough space to compute all the eigenvectors orthogonally
will cause serious degradation in performance. At the limit
(that is, $clustersize = n-1$) p?stein will perform no
better than ?stein on a single processor.

For $clustersize = n/\sqrt{NPROW \cdot NPCOL}$
reorthogonalizing all eigenvectors will increase the total
execution time by a factor of 2 or more.

For $clustersize > n/\sqrt{NPROW \cdot NPCOL}$ execution time
will grow as the square of the cluster size, all other factors
remaining equal and assuming enough workspace. Less
workspace means less reorthogonalization but faster
execution.

If $lwork = -1$, then $lwork$ is global input and a workspace
query is assumed; the routine only calculates the size
required for optimal performance for all work arrays. Each

of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by `pxerbla`.

iwork

(local) INTEGER. Workspace array.

liwork

(local) INTEGER, dimension of *iwork*.

$liwork \geq 6 * nnp$

Where:

$nnp = \max(n, NPROW * NPCOL + 1, 4)$

If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a

On exit,

If *jobz* = 'V', and if *info* = 0, `sub(A)` contains the distributed matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:

for *ibtype* = 1 or 2, $Z^T * \text{sub}(B) * Z = I$;

for *ibtype* = 3, $Z^T * \text{inv}(\text{sub}(B)) * Z = I$.

If *jobz* = 'N', then on exit the upper triangle (if *uplo*='U') or the lower triangle (if *uplo*='L') of `sub(A)`, including the diagonal, is destroyed.

b

On exit, if *info* ≤ *n*, the part of `sub(B)` containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization $\text{sub}(B) = U^T * U$ or $\text{sub}(B) = L * L^T$.

m

(global) INTEGER. The total number of eigenvalues found, $0 \leq m \leq n$.

nz

(global) INTEGER.

Total number of eigenvectors computed. $0 \leq nz \leq m$. The number of columns of *z* that are filled.

If *jobz* ≠ 'V', *nz* is not referenced.

If *jobz* = 'V', *nz* = *m* unless the user supplies insufficient space and *p?sygvx* is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in *z* (*m.le.descz(n_)*) and sufficient workspace to compute them. (See *lwork* below.) *p?sygvx* is always able to detect insufficient space without computation unless *range.eq.'V'*.

w

(global)

REAL for *pssygvx*

DOUBLE PRECISION for *pdsygvx*.

Array, DIMENSION (*n*). On normal exit, the first *m* entries contain the selected eigenvalues in ascending order.

z

(local).

REAL for *pssygvx*

DOUBLE PRECISION for *pdsygvx*.

global dimension (*n*, *n*), local dimension (*lld_z*, *LOCc(jz+n-1)*).

If *jobz* = 'V', then on normal exit the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

If *jobz* = 'N', then *z* is not referenced.

work

If *jobz*='N' *work*(1) = optimal amount of workspace required to compute eigenvalues efficiently

If *jobz* = 'V' *work*(1) = optimal amount of workspace required to compute eigenvalues and eigenvectors efficiently with no guarantee on orthogonality.

If *range*='V', it is assumed that all eigenvectors may be required.

ifail

(global) INTEGER.

Array, DIMENSION (*n*).

ifail provides additional information when *info.ne.0*

If $(\text{mod}(\text{info}/16, 2) \neq 0)$ then *ifail*(1) indicates the order of the smallest minor which is not positive definite. If $(\text{mod}(\text{info}, 2) \neq 0)$ on exit, then *ifail* contains the indices of the eigenvectors that failed to converge. If neither of the above error conditions hold and *jobz* = 'V', then the first *m* elements of *ifail* are set to zero.

iclustr

(global) INTEGER.

Array, DIMENSION $(2 * \text{NPROW} * \text{NPCOL})$. This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see *lwork*, *orfac* and *info*).

Eigenvectors corresponding to clusters of eigenvalues indexed *iclustr*(2*i-1) to *iclustr*(2*i), could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal. *iclustr*() is a zero terminated array.

$(\text{iclustr}(2*k) \neq 0 \text{ and } \text{iclustr}(2*k+1) \neq 0)$ if and only if *k* is the number of clusters *iclustr* is not referenced if *jobz* = 'N'.

gap

(global)

REAL for pssygvx

DOUBLE PRECISION for pdsygvx.

Array, DIMENSION $(\text{NPROW} * \text{NPCOL})$. This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the *i*-th cluster may be as high as $(C * n) / \text{gap}(i)$, where *C* is a small constant.

info

(global) INTEGER.

If *info* = 0, the execution is successful.

If *info* < 0: the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = $-(i * 100 + j)$, if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

If *info* > 0:

If $(\text{mod}(\text{info}, 2) \neq 0)$, then one or more eigenvectors failed to converge. Their indices are stored in *ifail*.

If $(\text{mod}(\text{info}/2, 2) \neq 0)$, then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace. The indices of the clusters are stored in the array *iclustr*.

If $(\text{mod}(\text{info}/4, 2) \neq 0)$, then space limit prevented *p?sygvx* from computing all of the eigenvectors between *vl* and *vu*. The number of eigenvectors computed is returned in *nz*.

If $(\text{mod}(\text{info}/8, 2) \neq 0)$, then *p?stebz* failed to compute eigenvalues.

If $(\text{mod}(\text{info}/16, 2) \neq 0)$, then *B* was not positive definite. *ifail*(1) indicates the order of the smallest minor which is not positive definite.

p?hegvx

Computes selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem.

Syntax

```
call pchegvx(ibtype, jobz, range, uplo, n, a, ia, ja, desca, b, ib, jb, descb,
vl, vu, il, iu, abstol, m, nz, w, orfac, z, iz, jz, descz, work, lwork, rwork,
lrwork, iwork, liwork, ifail, iclustr, gap, info)
```

```
call pzhegvx(ibtype, jobz, range, uplo, n, a, ia, ja, desca, b, ib, jb, descb,
vl, vu, il, iu, abstol, m, nz, w, orfac, z, iz, jz, descz, work, lwork, rwork,
lrwork, iwork, liwork, ifail, iclustr, gap, info)
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form

$$\text{sub}(A) * x = \lambda * \text{sub}(B) * x, \quad \text{sub}(A) * \text{sub}(B) * x = \lambda * x, \quad \text{or} \quad \text{sub}(B) * \text{sub}(A) * x = \lambda * x.$$

Here *sub(A)* denoting *A*(*ia:ia+n-1, ja:ja+n-1*) and *sub(B)* are assumed to be Hermitian and *sub(B)* denoting *B*(*ib:ib+n-1, jb:jb+n-1*) is also positive definite.

Input Parameters

<i>ibtype</i>	(global) INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: If <i>ibtype</i> = 1, the problem type is $\text{sub}(A) * x = \text{lambda} * \text{sub}(B) * x$; If <i>ibtype</i> = 2, the problem type is $\text{sub}(A) * \text{sub}(B) * x = \text{lambda} * x$; If <i>ibtype</i> = 3, the problem type is $\text{sub}(B) * \text{sub}(A) * x = \text{lambda} * x$.
<i>jobz</i>	(global). CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>range</i>	(global). CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues in the interval: [<i>vl</i>, <i>vu</i>] If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> through <i>iu</i>.
<i>uplo</i>	(global). CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>a</i> and <i>b</i> store the upper triangles of $\text{sub}(A)$ and $\text{sub}(B)$; If <i>uplo</i> = 'L', arrays <i>a</i> and <i>b</i> store the lower triangles of $\text{sub}(A)$ and $\text{sub}(B)$.
<i>n</i>	(global). INTEGER. The order of the matrices $\text{sub}(A)$ and $\text{sub}(B)$ ($n \geq 0$).
<i>a</i>	(local) COMPLEX for pchegvx DOUBLE COMPLEX for pzhegvx. Pointer into the local memory to an array of dimension (<i>lld_a</i>, <i>LOCc(ja+n-1)</i>). On entry, this array contains the local pieces of the <i>n</i>-by-<i>n</i> Hermitian distributed matrix $\text{sub}(A)$. If <i>uplo</i> = 'U', the leading <i>n</i>-by-<i>n</i> upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix. If <i>uplo</i> = 'L', the leading <i>n</i>-by-<i>n</i> lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the matrix.

<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i>, respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i>. If <i>desca(ctxt_)</i> is incorrect, p?hegvx cannot guarantee correct error reporting.
<i>b</i>	(local). COMPLEX for pchegvx DOUBLE COMPLEX for pzhegvx. Pointer into the local memory to an array of dimension (<i>lld_b</i>, <i>LOCc(jb+n-1)</i>). On entry, this array contains the local pieces of the <i>n</i>-by-<i>n</i> Hermitian distributed matrix sub(<i>B</i>). If <i>uplo</i> = 'U', the leading <i>n</i>-by-<i>n</i> upper triangular part of sub(<i>B</i>) contains the upper triangular part of the matrix. If <i>uplo</i> = 'L', the leading <i>n</i>-by-<i>n</i> lower triangular part of sub(<i>B</i>) contains the lower triangular part of the matrix.
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the global array <i>b</i> indicating the first row and the first column of the submatrix <i>B</i>, respectively.
<i>descb</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>B</i>. <i>descb(ctxt_)</i> must be equal to <i>desca(ctxt_)</i>.
<i>vl, vu</i>	(global) REAL for pchegvx DOUBLE PRECISION for pzhegvx. If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il, iu</i>	(global) INTEGER. If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $il \geq 1, \min(il, n) \leq iu \leq n$

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol (global)
 REAL for pchegvx
 DOUBLE PRECISION for pzhegvx.
 If *jobz*='V', setting *abstol* to $p?lamch(context, 'U')$ yields the most orthogonal eigenvectors.
 The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a, b]$ of width less than or equal to
 $abstol + eps * \max(|a|, |b|)$,
 where *eps* is the machine precision. If *abstol* is less than or equal to zero, then $eps * \text{norm}(T)$ will be used in its place, where $\text{norm}(T)$ is the 1-norm of the tridiagonal matrix obtained by reducing *A* to tridiagonal form.
 Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold $2 * p?lamch('S')$ not zero. If this routine returns with
 $(\text{mod}(info, 2) \neq 0) \text{ or } * (\text{mod}(info/8, 2) \neq 0)$,
 indicating that some eigenvalues or eigenvectors did not converge, try setting *abstol* to $2 * p?lamch('S')$.

orfac (global).
 REAL for pchegvx
 DOUBLE PRECISION for pzhegvx.
 Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within $tol = orfac * \text{norm}(A)$ of each other are to be reorthogonalized. However, if the workspace is insufficient (see *lwork*), *tol* may be decreased until all eigenvectors to be reorthogonalized can be stored in one process. No reorthogonalization will be done if *orfac* equals zero. A default value of 1.0E-3 is used if *orfac* is negative. *orfac* should be identical on all processes.

iz, jz (global) INTEGER. The row and column indices in the global array *z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *Z*. *descz*(*ctxt_*) must equal *desca*(*ctxt_*).

work (local)
COMPLEX for *pchegvx*
DOUBLE COMPLEX for *pzhegvx*.
Workspace array, dimension (*lwork*)

lwork (local).
INTEGER. The dimension of the array *work*.
If only eigenvalues are requested:
 $lwork \geq n + \max(NB * (np0 + 1), 3)$
If eigenvectors are requested:
 $lwork \geq n + (np0 + mq0 + NB) * NB$
with $nq0 = \text{numroc}(nn, NB, 0, 0, NPCOL)$.
For optimal performance, greater workspace is needed, that is
 $lwork \geq \max(lwork, n, nhetr\_lwork, nhegst\_lwork)$
where *lwork* is as defined above, and
 $nhetr\_lwork = 2 * (anb + 1) * (4 * nps + 2) + (nps + 1) * nps$;
 $nhegst\_lwork = 2 * np0 * nb + nq0 * nb + nb * nb$
 $nb = \text{desca}(mb\_)$
 $np0 = \text{numroc}(n, nb, 0, 0, NPROW)$
 $nq0 = \text{numroc}(n, nb, 0, 0, NPCOL)$
 $ictxt = \text{desca}(ctxt\_)$
 $anb = \text{pjlaenv}(ictxt, 3, 'p?hettrd', 'L', 0, 0, 0, 0)$
 $sqnpc = \text{sqrt}(\text{dble}(NPROW * NPCOL))$
 $nps = \max(\text{numroc}(n, 1, 0, 0, sqnpc), 2 * anb)$
numroc is a ScaLAPACK tool functions;
pjlaenv is a ScaLAPACK environmental inquiry function
MYROW, *MYCOL*, *NPROW* and *NPCOL* can be determined by calling the subroutine *blacs_gridinfo*.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each

of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by `pxerbla`.

rwork

(local)
 REAL for `pchegvx`
 DOUBLE PRECISION for `pzhgvx`.
 Workspace array, DIMENSION (*lrwork*).

lrwork

(local) INTEGER. The dimension of the array *rwork*.
 See below for definitions of variables used to define *lrwork*.
 If no eigenvectors are requested (*jobz* = 'N'), then

$$lrwork \geq 5*nn+4*n$$

If eigenvectors are requested (*jobz* = 'V'), then the amount of workspace required to guarantee that all eigenvectors are computed is:

$$lrwork \geq 4*n + \max(5*nn, np0*mq0) + \text{iceil}(neig, \text{NPROW}*NPCOL) * nn$$

The computed eigenvectors may not be orthogonal if the minimal workspace is supplied and *orfac* is too small. If you want to guarantee orthogonality (at the cost of potentially poor performance) you should add the following value to *lrwork*:

$$(clustersize-1)*n,$$

where *clustersize* is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:

$$\{w(k), \dots, w(k+clustersize-1) \mid w(j+1) \leq w(j) + orfac*2*norm(A)\}$$

Variable definitions:

neig = number of eigenvectors requested;

nb = *desca*(*mb_*) = *desca*(*nb_*) = *descz*(*mb_*) = *descz*(*nb_*);

nn = max(*n*, *nb*, 2);

desca(*rsrc_*) = *desca*(*nb_*) = *descz*(*rsrc_*) = *descz*(*csrc_*) = 0;

np0 = numroc(*nn*, *nb*, 0, 0, NPROW);

mq0 = numroc(max(*neig*, *nb*, 2), *nb*, 0, 0, NPCOL);

iceil(*x*, *y*) is a ScaLAPACK function returning ceiling(*x*/*y*).

When *lwork* is too small:

If *lwork* is too small to guarantee orthogonality, *p?hegvx* attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues.

If *lwork* is too small to compute all the eigenvectors requested, no computation is performed and *info*= -25 is returned. Note that when *range*='V', *p?hegvx* does not know how many eigenvectors are requested until the eigenvalues are computed. Therefore, when *range*='V' and as long as *lwork* is large enough to allow *p?hegvx* to compute the eigenvalues, *p?hegvx* will compute the eigenvalues and as many eigenvectors as it can.

Relationship between workspace, orthogonality & performance:

If *clustersize* > $n/\sqrt{\text{NPROW}*\text{NPCOL}}$, then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. In the limit (that is, *clustersize* = *n*-1) *p?stein* will perform no better than *?stein* on 1 processor.

For *clustersize* = $n/\sqrt{\text{NPROW}*\text{NPCOL}}$ reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.

For *clustersize* > $n/\sqrt{\text{NPROW}*\text{NPCOL}}$ execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by *pxerbla*.

iwork

(local) INTEGER. Workspace array.

liwork

(local) INTEGER, dimension of *iwork*.

$liwork \geq 6*nnp$

Where: $nnp = \max(n, \text{NPROW}*\text{NPCOL} + 1, 4)$

If $liwork = -1$, then $liwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a	On exit, if $jobz = 'V'$, then if $info = 0$, $sub(A)$ contains the distributed matrix Z of eigenvectors. The eigenvectors are normalized as follows: If $ibtype = 1$ or 2, then $Z^H * sub(B) * Z = i$; If $ibtype = 3$, then $Z^H * inv(sub(B)) * Z = i$. If $jobz = 'N'$, then on exit the upper triangle (if $uplo='U'$) or the lower triangle (if $uplo='L'$) of $sub(A)$, including the diagonal, is destroyed.
b	On exit, if $info \leq n$, the part of $sub(B)$ containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization $sub(B) = U^H * U$, or $sub(B) = L * L^H$.
m	(global) INTEGER. The total number of eigenvalues found, $0 \leq m \leq n$.
nz	(global) INTEGER. Total number of eigenvectors computed. $0 < nz < m$. The number of columns of z that are filled. If $jobz \neq 'V'$, nz is not referenced. If $jobz = 'V'$, $nz = m$ unless the user supplies insufficient space and <code>p?hegvx</code> is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in z (m. <i>le.</i> $descz(n_)$) and sufficient workspace to compute them. (See $lwork$ below.) The routine <code>p?hegvx</code> is always able to detect insufficient space without computation unless $range = 'V'$.
w	(global) REAL for <code>pchegvx</code> DOUBLE PRECISION for <code>pzhegvx</code>.

	Array, DIMENSION (n). On normal exit, the first m entries contain the selected eigenvalues in ascending order.
<i>z</i>	(local). COMPLEX for pchegvx DOUBLE COMPLEX for pzhegvx. global dimension (n, n), local dimension (lld_z, $LOCc(jz+n-1)$). If <i>jobz</i> = 'V', then on normal exit the first m columns of <i>z</i> contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of <i>z</i> contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in <i>ifail</i>. If <i>jobz</i> = 'N', then <i>z</i> is not referenced.
<i>work</i>	On exit, <i>work</i> (1) returns the optimal amount of workspace.
<i>rwork</i>	On exit, <i>rwork</i>(1) contains the amount of workspace required for optimal efficiency If <i>jobz</i>='N' <i>rwork</i>(1) = optimal amount of workspace required to compute eigenvalues efficiently If <i>jobz</i>='V' <i>rwork</i>(1) = optimal amount of workspace required to compute eigenvalues and eigenvectors efficiently with no guarantee on orthogonality. If <i>range</i>='V', it is assumed that all eigenvectors may be required when computing optimal workspace.
<i>ifail</i>	(global) INTEGER. Array, DIMENSION (n). <i>ifail</i> provides additional information when <i>info.ne.0</i> If (<i>mod</i>(<i>info</i>/16,2).ne.0), then <i>ifail</i>(1) indicates the order of the smallest minor which is not positive definite. If (<i>mod</i>(<i>info</i>,2).ne.0) on exit, then <i>ifail</i>(1) contains the indices of the eigenvectors that failed to converge. If neither of the above error conditions are held, and <i>jobz</i> = 'V', then the first m elements of <i>ifail</i> are set to zero.
<i>iclustr</i>	(global) INTEGER. Array, DIMENSION (2*NPROW*NPCOL). This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to

insufficient workspace (see *lwork*, *orfac* and *info*).
 Eigenvectors corresponding to clusters of eigenvalues indexed *iclustr*(2*i-1) to *iclustr*(2*i), could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal.

iclustr() is a zero terminated array.

(*iclustr*(2*k) .ne.0 .and. *iclustr*(2*k+1) .eq.0) if and only if *k* is the number of clusters.

iclustr is not referenced if *jobz* = 'N'.

gap

(global)

REAL for pchegvx

DOUBLE PRECISION for pzhegvx.

Array, DIMENSION (NPROW*NPCOL).

This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the *i*-th cluster may be as high as $(C*n)/gap(i)$, where *C* is a small constant.

info

(global) INTEGER.

If *info* = 0, the execution is successful.

If *info* < 0: the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(i*100+j), if the *i*-th argument is a scalar and had an illegal value, then *info* = -i.

If *info* > 0:

If (mod(*info*,2) .ne.0), then one or more eigenvectors failed to converge. Their indices are stored in *ifail*.

If (mod(*info*,2,2) .ne.0), then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace. The indices of the clusters are stored in the array *iclustr*.

If (mod(*info*/4,2) .ne.0), then space limit prevented p?sygvx from computing all of the eigenvectors between *vl* and *vu*. The number of eigenvectors computed is returned in *nz*.

If $(\text{mod}(\text{info}/8, 2) \neq 0)$, then `p?stebz` failed to compute eigenvalues.

If $(\text{mod}(\text{info}/16, 2) \neq 0)$, then B was not positive definite. `ifail(1)` indicates the order of the smallest minor which is not positive definite.

ScaLAPACK Auxiliary and Utility Routines

7

This chapter describes the Intel® Math Kernel Library implementation of ScaLAPACK [Auxiliary Routines](#) and [Utility Functions and Routines](#). The library includes routines for both real and complex data.



NOTE. ScaLAPACK routines are provided only with Intel® MKL versions for Linux* and Windows* OSs.

Routine naming conventions, mathematical notation, and matrix storage schemes used for ScaLAPACK auxiliary and utility routines are the same as described in previous chapters. Some routines and functions may have combined character codes, such as `sc` or `dz`. For example, the routine `pscsum1` uses a complex input array and returns a real value.

Auxiliary Routines

Table 7-1 ScaLAPACK Auxiliary Routines

Routine Name	Data Types	Description
<code>p?lacgv</code>	<code>c, z</code>	Conjugates a complex vector.
<code>p?max1</code>	<code>c, z</code>	Finds the index of the element whose real part has maximum absolute value (similar to the Level 1 PBLAS <code>p?amax</code> , but using the absolute value to the real part).
<code>?combamax1</code>	<code>c, z</code>	Finds the element with maximum real part absolute value and its corresponding global index.
<code>p?sum1</code>	<code>sc, dz</code>	Forms the 1-norm of a complex vector similar to Level 1 PBLAS <code>p?asum</code> , but using the true absolute value.
<code>p?dbtrsv</code>	<code>s, d, c, z</code>	Computes an <i>LU</i> factorization of a general tridiagonal matrix with no pivoting. The routine is called by <code>p?dbtrs</code> .
<code>p?dttrsv</code>	<code>s, d, c, z</code>	Computes an <i>LU</i> factorization of a general band matrix, using partial pivoting with row interchanges. The routine is called by <code>p?dttrs</code> .

Routine Name	Data Types	Description
p?gebd2	s, d, c, z	Reduces a general rectangular matrix to real bidiagonal form by an orthogonal/unitary transformation (unblocked algorithm).
p?gehd2	s, d, c, z	Reduces a general matrix to upper Hessenberg form by an orthogonal/unitary similarity transformation (unblocked algorithm).
p?gelq2	s, d, c, z	Computes an LQ factorization of a general rectangular matrix (unblocked algorithm).
p?geql2	s, d, c, z	Computes a QL factorization of a general rectangular matrix (unblocked algorithm).
p?geqr2	s, d, c, z	Computes a QR factorization of a general rectangular matrix (unblocked algorithm).
p?gerq2	s, d, c, z	Computes an RQ factorization of a general rectangular matrix (unblocked algorithm).
p?getf2	s, d, c, z	Computes an LU factorization of a general matrix, using partial pivoting with row interchanges (local blocked algorithm).
p?labrd	s, d, c, z	Reduces the first nb rows and columns of a general rectangular matrix A to real bidiagonal form by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of A .
p?lacon	s, d, c, z	Estimates the 1-norm of a square matrix, using the reverse communication for evaluating matrix-vector products.
p?laconsb	s, d	Looks for two consecutive small subdiagonal elements.
p?lACP2	s, d, c, z	Copies all or part of a distributed matrix to another distributed matrix.
p?lACP3	s, d	Copies from a global parallel array into a local replicated array or vice versa.
p?lACPY	s, d, c, z	Copies all or part of one two-dimensional array to another.

Routine Name	Data Types	Description
<code>p?laevswp</code>	<code>s, d, c, z</code>	Moves the eigenvectors from where they are computed to ScaLAPACK standard block cyclic array.
<code>p?lahrd</code>	<code>s, d, c, z</code>	Reduces the first nb columns of a general rectangular matrix A so that elements below the k^{th} subdiagonal are zero, by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of A .
<code>p?laiect</code>	<code>s, d, c, z</code>	Exploits IEEE arithmetic to accelerate the computations of eigenvalues. (C interface function).
<code>p?lange</code>	<code>s, d, c, z</code>	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a general rectangular matrix.
<code>p?lanhs</code>	<code>s, d, c, z</code>	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of an upper Hessenberg matrix.
<code>p?lansy, p?lanhe</code>	<code>s, d, c, z/c, z</code>	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a real symmetric or complex Hermitian matrix.
<code>p?lantr</code>	<code>s, d, c, z</code>	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a triangular matrix.
<code>p?lapiv</code>	<code>s, d, c, z</code>	Applies a permutation matrix to a general distributed matrix, resulting in row or column pivoting.
<code>p?laqge</code>	<code>s, d, c, z</code>	Scales a general rectangular matrix, using row and column scaling factors computed by <code>p?geequ</code> .
<code>p?laqsy</code>	<code>s, d, c, z</code>	Scales a symmetric/Hermitian matrix, using scaling factors computed by <code>p?poequ</code> .
<code>p?laredld</code>	<code>s, d</code>	Redistributes an array assuming that the input array <code>bycol</code> is distributed across rows and that all process columns contain the same copy of <code>bycol</code> .

Routine Name	Data Types	Description
p?lared2d	<i>s, d</i>	Redistributes an array assuming that the input array <i>byrow</i> is distributed across columns and that all process rows contain the same copy of <i>byrow</i> .
p?larf	<i>s, d, c, z</i>	Applies an elementary reflector to a general rectangular matrix.
p?larfb	<i>s, d, c, z</i>	Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.
p?larfc	<i>c, z</i>	Applies the conjugate transpose of an elementary reflector to a general matrix.
p?larfg	<i>s, d, c, z</i>	Generates an elementary reflector (Householder matrix).
p?larft	<i>s, d, c, z</i>	Forms the triangular vector T of a block reflector $H=I-VTV^H$
p?larz	<i>s, d, c, z</i>	Applies an elementary reflector as returned by p?tzrzf to a general matrix.
p?larzb	<i>s, d, c, z</i>	Applies a block reflector or its transpose/conjugate-transpose as returned by p?tzrzf to a general matrix.
p?larzc	<i>c, z</i>	Applies (multiplies by) the conjugate transpose of an elementary reflector as returned by p?tzrzf to a general matrix.
p?larzt	<i>s, d, c, z</i>	Forms the triangular factor T of a block reflector $H=I-VTV^H$ as returned by p?tzrzf .
p?lascl	<i>s, d, c, z</i>	Multiplies a general rectangular matrix by a real scalar defined as C_{to}/C_{from} .
p?laset	<i>s, d, c, z</i>	Initializes the off-diagonal elements of a matrix to α and the diagonal elements to β .
p?lasmsub	<i>s, d</i>	Looks for a small subdiagonal element from the bottom of the matrix that it can safely set to zero.
p?lassq	<i>s, d, c, z</i>	Updates a sum of squares represented in scaled form.

Routine Name	Data Types	Description
p?laswp	s, d, c, z	Performs a series of row interchanges on a general rectangular matrix.
p?latra	s, d, c, z	Computes the trace of a general square distributed matrix.
p?latrd	s, d, c, z	Reduces the first nb rows and columns of a symmetric/Hermitian matrix A to real tridiagonal form by an orthogonal/unitary similarity transformation.
p?latrz	s, d, c, z	Reduces an upper trapezoidal matrix to upper triangular form by means of orthogonal/unitary transformations.
p?lauu2	s, d, c, z	Computes the product UU^H or $L^H L$, where U and L are upper or lower triangular matrices (local unblocked algorithm).
p?lauum	s, d, c, z	Computes the product UU^H or $L^H L$, where U and L are upper or lower triangular matrices.
p?lawil	s, d	Forms the Wilkinson transform.
p?org2l/p?ung2l	s, d, c, z	Generates all or part of the orthogonal/unitary matrix Q from a QL factorization determined by p?geqlf (unblocked algorithm).
p?org2r/p?ung2r	s, d, c, z	Generates all or part of the orthogonal/unitary matrix Q from a QR factorization determined by p?geqrf (unblocked algorithm).
p?orgl2/p?ungl2	s, d, c, z	Generates all or part of the orthogonal/unitary matrix Q from an LQ factorization determined by p?gelqf (unblocked algorithm).
p?orgr2/p?ungr2	s, d, c, z	Generates all or part of the orthogonal/unitary matrix Q from an RQ factorization determined by p?gerqf (unblocked algorithm).
p?orm2l/p?unm2l	s, d, c, z	Multiplies a general matrix by the orthogonal/unitary matrix from a QL factorization determined by p?geqlf (unblocked algorithm).

Routine Name	Data Types	Description
p?orm2r/p?unm2r	<i>s, d, c, z</i>	Multiplies a general matrix by the orthogonal/unitary matrix from a <i>QR</i> factorization determined by p?geqrf (unblocked algorithm).
p?orml2/p?unml2	<i>s, d, c, z</i>	Multiplies a general matrix by the orthogonal/unitary matrix from an <i>LQ</i> factorization determined by p?gelqf (unblocked algorithm).
p?ormr2/p?unmr2	<i>s, d, c, z</i>	Multiplies a general matrix by the orthogonal/unitary matrix from an <i>RQ</i> factorization determined by p?gerqf (unblocked algorithm).
p?pbtrsv	<i>s, d, c, z</i>	Solves a single triangular linear system via frontsolve or backsolve where the triangular matrix is a factor of a banded matrix computed by p?pbtrf .
p?pttrsv	<i>s, d, c, z</i>	Solves a single triangular linear system via frontsolve or backsolve where the triangular matrix is a factor of a tridiagonal matrix computed by p?pttrf .
p?potf2	<i>s, d, c, z</i>	Computes the Cholesky factorization of a symmetric/Hermitian positive definite matrix (local unblocked algorithm).
p?rscl	<i>s, d, cs, zd</i>	Multiplies a vector by the reciprocal of a real scalar.
p?sygs2/p?hegs2	<i>s, d, c, z</i>	Reduces a symmetric/Hermitian definite generalized eigenproblem to standard form, using the factorization results obtained from p?potrf (local unblocked algorithm).
p?sytd2/p?hetd2	<i>s, d, c, z</i>	Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation (local unblocked algorithm).
p?trti2	<i>s, d, c, z</i>	Computes the inverse of a triangular matrix (local unblocked algorithm).
?lamsh	<i>s, d</i>	Sends multiple shifts through a small (single node) matrix to maximize the number of bulges that can be sent through.
?laref	<i>s, d</i>	Applies Householder reflectors to matrices on either their rows or columns.

Routine Name	Data Types	Description
<code>?lasorte</code>	<code>s, d</code>	Sorts eigenpairs by real and complex data types.
<code>?lasrt2</code>	<code>s, d</code>	Sorts numbers in increasing or decreasing order.
<code>?stein2</code>	<code>s, d</code>	Computes the eigenvectors corresponding to specified eigenvalues of a real symmetric tridiagonal matrix, using inverse iteration.
<code>?dbtff2</code>	<code>s, d, c, z</code>	Computes an LU factorization of a general band matrix with no pivoting (local unblocked algorithm).
<code>?dbtrf</code>	<code>s, d, c, z</code>	Computes an LU factorization of a general band matrix with no pivoting (local blocked algorithm).
<code>?dttrf</code>	<code>s, d, c, z</code>	Computes an LU factorization of a general tridiagonal matrix with no pivoting (local blocked algorithm).
<code>?dttrsv</code>	<code>s, d, c, z</code>	Solves a general tridiagonal system of linear equations using the LU factorization computed by <code>?dttrf</code> .
<code>?pttrsv</code>	<code>s, d, c, z</code>	Solves a symmetric (Hermitian) positive-definite tridiagonal system of linear equations, using the LDL^H factorization computed by <code>?pttrf</code> .
<code>?steqr2</code>	<code>s, d</code>	Computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method.

p?lacgv

Conjugates a complex vector.

Syntax

```
call pclacgv(n, x, ix, jx, descx, incx)
```

```
call pzlacgv(n, x, ix, jx, descx, incx)
```

Description

The routine conjugates a complex vector `sub(x)` of length n , where `sub(x)` denotes $X(ix, jx:jx+n-1)$ if `incx = m_x`, and $X(ix:ix+n-1, jx)$ if `incx = 1`.

Input Parameters

<i>n</i>	(global) INTEGER. The length of the distributed vector <code>sub(x)</code> .
<i>x</i>	(local). COMPLEX for <code>pclacgv</code> COMPLEX*16 for <code>pzlacgv</code> . Pointer into the local memory to an array of DIMENSION (<i>lld_x</i> , *). On entry the vector to be conjugated $x(i) = X(ix + (jx-1)*m_x + (i-1)*incx)$, $1 \leq i \leq n$.
<i>ix</i>	(global) INTEGER. The row index in the global array <i>x</i> indicating the first row of <code>sub(x)</code> .
<i>jx</i>	(global) INTEGER. The column index in the global array <i>x</i> indicating the first column of <code>sub(x)</code> .
<i>descx</i>	(global and local) INTEGER. Array, DIMENSION (<i>dlen_</i>). The array descriptor for the distributed matrix <i>x</i> .
<i>incx</i>	(global) INTEGER. The global increment for the elements of <i>x</i> . Only two values of <i>incx</i> are supported in this version, namely 1 and <i>m_x</i> . <i>incx</i> must not be zero.

Output Parameters

<i>x</i>	(local). On exit, the conjugated vector.

p?max1

Finds the index of the element whose real part has maximum absolute value (similar to the Level 1 PBLAS `p?amax`, but using the absolute value to the real part).

Syntax

```
call pmax1(n, amax, indx, x, ix, jx, descx, incx)
call pzmax1(n, amax, indx, x, ix, jx, descx, incx)
```

Description

This routine computes the global index of the maximum element in absolute value of a distributed vector $\text{sub}(x)$. The global index is returned in indx and the value is returned in amax , where $\text{sub}(x)$ denotes $X(\text{ix}:\text{ix}+n-1, \text{jx})$ if $\text{incx} = 1$, $X(\text{ix}, \text{jx}:\text{jx}+n-1)$ if $\text{incx} = m_x$.

Input Parameters

n	(global) pointer to INTEGER. The number of components of the distributed vector $\text{sub}(x)$. $n \geq 0$.
x	(local) COMPLEX for p _c max1. COMPLEX*16 for p _z max1 Array containing the local pieces of a distributed matrix of dimension of at least $((\text{jx}-1)*m_x + \text{ix} + (n-1)*\text{abs}(\text{incx}))$. This array contains the entries of the distributed vector $\text{sub}(x)$.
ix	(global) INTEGER. The row index in the global array x indicating the first row of $\text{sub}(x)$.
jx	(global) INTEGER. The column index in the global array x indicating the first column of $\text{sub}(x)$.
descx	(global and local) INTEGER. Array, DIMENSION (dlen_x) . The array descriptor for the distributed matrix x .
incx	(global) INTEGER. The global increment for the elements of x . Only two values of incx are supported in this version, namely 1 and m_x . incx must not be zero.

Output Parameters

amax	(global output) pointer to REAL. The absolute value of the largest entry of the distributed vector $\text{sub}(x)$ only in the scope of $\text{sub}(x)$.
indx	(global output) pointer to INTEGER. The global index of the element of the distributed vector $\text{sub}(x)$ whose real part has maximum absolute value.

?combamax1

Finds the element with maximum real part absolute value and its corresponding global index.

Syntax

```
call ccombamax1(v1, v2)
```

```
call zcombamax1(v1, v2)
```

Description

This routine finds the element having maximum real part absolute value as well as its corresponding global index.

Input Parameters

<i>v1</i>	(local) COMPLEX for ccombamax1 COMPLEX*16 for zcombamax1 Array, DIMENSION 2. The first maximum absolute value element and its global index. <i>v1</i> (1)=amax, <i>v1</i> (2)=indx.
<i>v2</i>	(local) COMPLEX for ccombamax1 COMPLEX*16 for zcombamax1 Array, DIMENSION 2. The second maximum absolute value element and its global index. <i>v2</i> (1)=amax, <i>v2</i> (2)=indx.

Output Parameters

<i>v1</i>	(local). The first maximum absolute value element and its global index. <i>v1</i> (1)=amax, <i>v1</i> (2)=indx.

p?sum1

Forms the 1-norm of a complex vector similar to Level 1 PBLAS p?asum, but using the true absolute value.

Syntax

```
call pscsum1(n, asum, x, ix, jx, descx, incx)
```

```
call pdzsum1(n, asum, x, ix, jx, descx, incx)
```

Description

This routine returns the sum of absolute values of a complex distributed vector $\text{sub}(x)$ in *asum*, where $\text{sub}(x)$ denotes $X(ix:ix+n-1, jx:jx)$, if $incx = 1$, $X(ix:ix, jx:jx+n-1)$, if $incx = m_x$.

Based on p?asum from the Level 1 PBLAS. The change is to use the 'genuine' absolute value.

Input Parameters

<i>n</i>	(global) INTEGER. The number of components of the distributed vector $\text{sub}(x)$. $n \geq 0$.
<i>x</i>	(local) COMPLEX for pscsum1 COMPLEX*16 for pdzsum1. Array containing the local pieces of a distributed matrix of dimension of at least $((jx-1)*m_x+ix+(n-1)*abs(incx))$. This array contains the entries of the distributed vector $\text{sub}(x)$.
<i>ix</i>	(global) INTEGER. The row index in the global array <i>x</i> indicating the first row of $\text{sub}(x)$.
<i>jx</i>	(global) INTEGER. The column index in the global array <i>x</i> indicating the first column of $\text{sub}(x)$.
<i>descx</i>	(global and local) INTEGER. Array, DIMENSION 8. The array descriptor for the distributed matrix <i>x</i> .
<i>incx</i>	(global) INTEGER. The global increment for the elements of <i>x</i> . Only two values of <i>incx</i> are supported in this version, namely 1 and m_x .

Output Parameters

<code>asum</code>	(local) Pointer to <code>REAL</code> . The sum of absolute values of the distributed vector <code>sub(x)</code> only in its scope.

p?dbtrsv

Computes an LU factorization of a general triangular matrix with no pivoting. The routine is called by p?dbtrsv.

Syntax

```
call psdbtrsv(uplo, trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af,
laf, work, lwork, info)

call pddbtrsv(uplo, trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af,
laf, work, lwork, info)

call pcdbtrsv(uplo, trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af,
laf, work, lwork, info)

call pzdbtrsv(uplo, trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af,
laf, work, lwork, info)
```

Description

This routines solves a banded triangular system of linear equations

$$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs) \text{ or}$$

$$A(1:n, ja:ja+n-1)^T * X = B(ib:ib+n-1, 1:nrhs) \text{ (for real flavors); } A(1:n, ja:ja+n-1)^H * X = B(ib:ib+n-1, 1:nrhs) \text{ (for complex flavors),}$$

where $A(1:n, ja:ja+n-1)$ is a banded triangular matrix factor produced by the Gaussian elimination code PD@ (dom_pre) BTRF and is stored in $A(1:n, ja:ja+n-1)$ and `af`. The matrix stored in $A(1:n, ja:ja+n-1)$ is either upper or lower triangular according to `uplo`, and the choice of solving $A(1:n, ja:ja+n-1)$ or $A(1:n, ja:ja+n-1)^T$ is dictated by the user by the parameter `trans`.

Routine `p?dbtrf` must be called first.

Input Parameters

<i>uplo</i>	(global) CHARACTER. If <i>uplo</i> ='U', the upper triangle of $A(1:n, ja:ja+n-1)$ is stored, if <i>uplo</i> = 'L', the lower triangle of $A(1:n, ja:ja+n-1)$ is stored.
<i>trans</i>	(global) CHARACTER. If <i>trans</i> = 'N', solve with $A(1:n, ja:ja+n-1)$, if <i>trans</i> = 'C', solve with conjugate transpose $A(1:n, ja:ja+n-1)$.
<i>n</i>	(global) INTEGER. The order of the distributed submatrix A_i ($n \geq 0$).
<i>bwl</i>	(global) INTEGER. Number of subdiagonals. $0 \leq bwl \leq n-1$.
<i>bwu</i>	(global) INTEGER. Number of subdiagonals. $0 \leq bwu \leq n-1$.
<i>nrhs</i>	(global) INTEGER. The number of right-hand sides; the number of columns of the distributed submatrix B ($nrhs \geq 0$).
<i>a</i>	(local). REAL for psdbtrsv DOUBLE PRECISION for pddbtrsv COMPLEX for pcdbrsv COMPLEX*16 for pzdbtrsv. Pointer into the local memory to an array with first DIMENSION $lld\_a \geq (bwl+bwu+1)$ (stored in <i>desca</i>). On entry, this array contains the local pieces of the n -by- n unsymmetric banded distributed Cholesky factor L or $L^T * A(1:n, ja:ja+n-1)$. This local portion is stored in the packed banded format used in LAPACK. See the <i>Application Notes</i> below and the ScaLAPACK manual for more detail on the format of distributed matrices.
<i>ja</i>	(global) INTEGER. The index in the global array <i>a</i> that points to the start of the matrix to be operated on (which may be either all of A or a submatrix of A).
<i>desca</i>	(global and local) INTEGER array of DIMENSION (<i>dlen_</i>).

	if 1d type (<i>dtype_a</i> = 501 or 502), <i>dlen</i> ≥ 7; if 2d type (<i>dtype_a</i> = 1), <i>dlen</i> ≥ 9. The array descriptor for the distributed matrix <i>A</i>. Contains information of mapping of <i>A</i> to memory.
<i>b</i>	(local) REAL for psdbtrsv DOUBLE PRECISION for pddbtrsv COMPLEX for pcdbrsv COMPLEX*16 for pzdbtrsv. Pointer into the local memory to an array of local lead DIMENSION <i>lld_b</i> ≥ <i>nb</i>. On entry, this array contains the local pieces of the right-hand sides <i>B(ib:ib+n-1, 1:nrhs)</i>.
<i>ib</i>	(global) INTEGER. The row index in the global array <i>b</i> that points to the first row of the matrix to be operated on (which may be either all of <i>b</i> or a submatrix of <i>B</i>).
<i>descb</i>	(global and local) INTEGER array of DIMENSION (<i>dlen_</i>). if 1d type (<i>dtype_b</i> = 502), <i>dlen</i> ≥ 7; if 2d type (<i>dtype_b</i> = 1), <i>dlen</i> ≥ 9. The array descriptor for the distributed matrix <i>B</i>. Contains information of mapping <i>B</i> to memory.
<i>laf</i>	(local) INTEGER. Size of user-input Auxiliary Filling space <i>af</i>. <i>laf</i> must be ≥ <i>nb</i> * (<i>bwl</i> + <i>bwu</i>) + 6 * max(<i>bwl</i>, <i>bwu</i>) * max(<i>bwl</i>, <i>bwu</i>). If <i>laf</i> is not large enough, an error code is returned and the minimum acceptable size will be returned in <i>af</i>(1).
<i>work</i>	(local). REAL for psdbtrsv DOUBLE PRECISION for pddbtrsv COMPLEX for pcdbrsv COMPLEX*16 for pzdbtrsv. Temporary workspace. This space may be overwritten in between calls to routines. <i>work</i> must be the size given in <i>lwork</i>.
<i>lwork</i>	(local or global) INTEGER.

Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.

$lwork \geq \max(bw1, bwu) * nrhs.$

Output Parameters

<i>a</i>	(local). This local portion is stored in the packed banded format used in LAPACK. Please see the ScaLAPACK manual for more detail on the format of distributed matrices.
<i>b</i>	On exit, this contains the local piece of the solutions distributed matrix <i>x</i> .
<i>af</i>	(local). REAL for psdbtrsv DOUBLE PRECISION for pddbtrsv COMPLEX for pcdbtrsv COMPLEX*16 for pzdbtrsv. Auxiliary Filling Space. Filling is created during the factorization routine <i>p?dbtrf</i> and this is stored in <i>af</i> . If a linear system is to be solved using <i>p?dbtrf</i> after the factorization routine, <i>af</i> must not be altered after the factorization.
<i>work</i>	On exit, <i>work</i> (1) contains the minimal <i>lwork</i> .
<i>info</i>	(local). INTEGER. If <i>info</i> = 0, the execution is successful. < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = - (<i>i</i> *100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

p?dttrsv

Computes an LU factorization of a general band matrix, using partial pivoting with row interchanges. The routine is called by p?dttrsv.

Syntax

```
call psdttrsv(uplo, trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af,
lwf, work, lwork, info)

call pddttrsv(uplo, trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af,
lwf, work, lwork, info)

call pcdttrsv(uplo, trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af,
lwf, work, lwork, info)

call pzdttrsv(uplo, trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af,
lwf, work, lwork, info)
```

Description

This routine solves a tridiagonal triangular system of linear equations

$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs)$ or

$A(1:n, ja:ja+n-1)^T * X = B(ib:ib+n-1, 1:nrhs)$ for real flavors; $A(1:n, ja:ja+n-1)^H * X = B(ib:ib+n-1, 1:nrhs)$ for complex flavors,

where $A(1:n, ja:ja+n-1)$ is a tridiagonal matrix factor produced by the Gaussian elimination code PS@ (dom_pre) TTRF and is stored in $A(1:n, ja:ja+n-1)$ and af .

The matrix stored in $A(1:n, ja:ja+n-1)$ is either upper or lower triangular according to $uplo$, and the choice of solving $A(1:n, ja:ja+n-1)$ or $A(1:n, ja:ja+n-1)^T$ is dictated by the user by the parameter $trans$.

Routine p?dttrf must be called first.

Input Parameters

uplo (global) CHARACTER.
 If $uplo='U'$, the upper triangle of $A(1:n, ja:ja+n-1)$ is stored,
 if $uplo = 'L'$, the lower triangle of $A(1:n, ja:ja+n-1)$ is stored.

trans (global) CHARACTER.
 If *trans* = 'N', solve with $A(1:n, ja:ja+n-1)$,
 if *trans* = 'C', solve with conjugate transpose $A(1:n, ja:ja+n-1)$.

n (global) INTEGER. The order of the distributed submatrix A ; ($n \geq 0$).

nrhs (global) INTEGER. The number of right-hand sides; the number of columns of the distributed submatrix $B(ib:ib+n-1, 1:nrhs)$. ($nrhs \geq 0$).

dl (local).
 REAL for psdttrsv
 DOUBLE PRECISION for pddttrsv
 COMPLEX for pcdttrsv
 COMPLEX*16 for pzdttrsv.
 Pointer to local part of global vector storing the lower diagonal of the matrix.
 Globally, *dl*(1) is not referenced, and *dl* must be aligned with *d*.
 Must be of size $\geq \text{desca}(nb\_)$.

d (local).
 REAL for psdttrsv
 DOUBLE PRECISION for pddttrsv
 COMPLEX for pcdttrsv
 COMPLEX*16 for pzdttrsv.
 Pointer to local part of global vector storing the main diagonal of the matrix.

du (local).
 REAL for psdttrsv
 DOUBLE PRECISION for pddttrsv
 COMPLEX for pcdttrsv
 COMPLEX*16 for pzdttrsv.
 Pointer to local part of global vector storing the upper diagonal of the matrix.
 Globally, *du*(*n*) is not referenced, and *du* must be aligned with *d*.

<i>ja</i>	(global) INTEGER. The index in the global array <i>a</i> that points to the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	(global and local). INTEGER array of DIMENSION (<i>dlen_</i>). if 1d type (<i>dtype_a</i> = 501 or 502), <i>dlen</i> ≥ 7; if 2d type (<i>dtype_a</i> = 1), <i>dlen</i> ≥ 9. The array descriptor for the distributed matrix <i>A</i> . Contains information of mapping of <i>A</i> to memory.
<i>b</i>	(local) REAL for psdttrsv DOUBLE PRECISION for pddttrsv COMPLEX for pcdttrsv COMPLEX*16 for pzdttrsv. Pointer into the local memory to an array of local lead DIMENSION <i>lld_b</i> ≥ <i>nb</i> . On entry, this array contains the local pieces of the right-hand sides <i>B</i> (<i>ib</i> : <i>ib</i> + <i>n</i> -1, 1 : <i>nrhs</i>).
<i>ib</i>	(global). INTEGER. The row index in the global array <i>b</i> that points to the first row of the matrix to be operated on (which may be either all of <i>b</i> or a submatrix of <i>B</i>).
<i>descb</i>	(global and local). INTEGER array of DIMENSION (<i>dlen_</i>). if 1d type (<i>dtype_b</i> = 502), <i>dlen</i> ≥ 7; if 2d type (<i>dtype_b</i> = 1), <i>dlen</i> ≥ 9. The array descriptor for the distributed matrix <i>B</i> . Contains information of mapping <i>B</i> to memory.
<i>laf</i>	(local). INTEGER. Size of user-input Auxiliary Filling space <i>af</i> . <i>laf</i> must be ≥ 2*(<i>nb</i> +2). If <i>laf</i> is not large enough, an error code is returned and the minimum acceptable size will be returned in <i>af</i> (1).
<i>work</i>	(local). REAL for psdttrsv DOUBLE PRECISION for pddttrsv COMPLEX for pcdttrsv

COMPLEX*16 for pzdttrsv.
 Temporary workspace. This space may be overwritten in
 between calls to routines.
work must be the size given in *lwork*.
lwork (local or global). INTEGER.
 Size of user-input workspace *work*. If *lwork* is too small,
 the minimal acceptable size will be returned in *work*(1) and
 an error code is returned.
 $lwork \geq 10 * npcol + 4 * nrhs$.

Output Parameters

dl (local).
 On exit, this array contains information containing the
 factors of the matrix.

d On exit, this array contains information containing the
 factors of the matrix. Must be of size $\geq desca(nb\_)$.

b On exit, this contains the local piece of the solutions
 distributed matrix X.

af (local).
 REAL for psdttrsv
 DOUBLE PRECISION for pddttrsv
 COMPLEX for pcdttrsv
 COMPLEX*16 for pzdttrsv.
 Auxiliary Filling Space. Filling is created during the
 factorization routine p?dttrf and this is stored in *af*. If a
 linear system is to be solved using p?dttrs after the
 factorization routine, *af* must not be altered after the
 factorization.

work On exit, *work*(1) contains the minimal *lwork*.

info (local). INTEGER.
 If *info*=0, the execution is successful.
 if *info*< 0: If the *i*-th argument is an array and the *j*-entry
 had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th
 argument is a scalar and had an illegal value, then *info* =
 -*i*.

p?gebd2

Reduces a general rectangular matrix to real bidiagonal form by an orthogonal/unitary transformation (unblocked algorithm).

Syntax

```
call psgebd2(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
call pdgebd2(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
call pcgebd2(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
call pzgebd2(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
```

Description

This routine reduces a real/complex general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ to upper or lower bidiagonal form B by an orthogonal/unitary transformation:
 $Q^* \text{sub}(A) P = B$.

If $m \geq n$, B is the upper bidiagonal; if $m < n$, B is the lower bidiagonal.

Input Parameters

m	(global) INTEGER. The number of rows of the distributed submatrix $\text{sub}(A)$. ($m \geq 0$).
n	(global) INTEGER. The order of the distributed submatrix $\text{sub}(A)$. ($n \geq 0$).
a	(local). REAL for psgebd2 DOUBLE PRECISION for pdgebd2 COMPLEX for pcgebd2 COMPLEX*16 for pzgebd2. Pointer into the local memory to an array of DIMENSION(lld_a , $LOCc(ja+n-1)$). On entry, this array contains the local pieces of the general distributed matrix $\text{sub}(A)$.

<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, DIMENSION (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>work</i>	(local). REAL for psgebd2 DOUBLE PRECISION for pdgebd2 COMPLEX for pcgebd2 COMPLEX*16 for pzgebd2. This is a workspace array of DIMENSION (<i>lwork</i>).
<i>lwork</i>	(local or global) INTEGER. The dimension of the array <i>work</i> . <i>lwork</i> is local input and must be at least $lwork \geq \max(m_{pa0}, n_{qa0})$, where $nb = mb\_a = nb\_a$, $iroffa = \text{mod}(ia-1, nb)$, $iarow = \text{indxg2p}(ia, nb, myrow, rsrc\_a, nprow)$, $iacol = \text{indxg2p}(ja, nb, mycol, csrc\_a, npcol)$, $mpa0 = \text{numroc}(m+iroffa, nb, myrow, iarow, nprow)$, $nqa0 = \text{numroc}(n+icoffa, nb, mycol, iacol, npcol)$. <i>indxg2p</i> and <i>numroc</i> are ScaLAPACK tool functions; <i>myrow</i> , <i>mycol</i> , <i>nprow</i> , and <i>npcol</i> can be determined by calling the subroutine <i>blacs_gridinfo</i> . If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by <i>pxerbla</i> .

Output Parameters

<i>a</i>	(local). On exit, if $m \geq n$, the diagonal and the first superdiagonal of $\text{sub}(A)$ are overwritten with the upper bidiagonal matrix <i>B</i> ; the elements below the diagonal, with the array <i>tauq</i> ,

represent the orthogonal/unitary matrix Q as a product of elementary reflectors, and the elements above the first superdiagonal, with the array *taup*, represent the orthogonal matrix P as a product of elementary reflectors. If $m < n$, the diagonal and the first subdiagonal are overwritten with the lower bidiagonal matrix B ; the elements below the first subdiagonal, with the array *tauq*, represent the orthogonal/unitary matrix Q as a product of elementary reflectors, and the elements above the diagonal, with the array *taup*, represent the orthogonal matrix P as a product of elementary reflectors. See *Applications Notes* below.

d

(local)

REAL for psgebd2

DOUBLE PRECISION for pdgebd2

COMPLEX for pcgebd2

COMPLEX*16 for pzgebd2.

Array, DIMENSION $LOCc(ja+\min(m,n)-1)$ if $m \geq n$;
 $LOCr(ia+\min(m,n)-1)$ otherwise. The distributed diagonal elements of the bidiagonal matrix B : $d(i) = a(i,i)$. *d* is tied to the distributed matrix A .

e

(local)

REAL for psgebd2

DOUBLE PRECISION for pdgebd2

COMPLEX for pcgebd2

COMPLEX*16 for pzgebd2.

Array, DIMENSION $LOCc(ja+\min(m,n)-1)$ if $m \geq n$;
 $LOCr(ia+\min(m,n)-2)$ otherwise. The distributed diagonal elements of the bidiagonal matrix B :

if $m \geq n$, $e(i) = a(i, i+1)$ for $i = 1, 2, \dots, n-1$;

if $m < n$, $e(i) = a(i+1, i)$ for $i = 1, 2, \dots, m-1$.

e is tied to the distributed matrix A .

tauq

(local).

REAL for psgebd2

DOUBLE PRECISION for pdgebd2

COMPLEX for pcgebd2

COMPLEX*16 for pzgebd2.

taup Array, DIMENSION $LOCc(ja+\min(m,n)-1)$. The scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Q . *tauq* is tied to the distributed matrix A .

(local).
 REAL for psgebd2
 DOUBLE PRECISION for pdgebd2
 COMPLEX for pcgebd2
 COMPLEX*16 for pzgebd2.

work Array, DIMENSION $LOCr(ia+\min(m,n)-1)$. The scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix P . *taup* is tied to the distributed matrix A .

info On exit, *work*(1) returns the minimal and optimal *lwork*.
 (local)
 INTEGER.
 If *info* = 0, the execution is successful.
 if *info* < 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100+*j*),
 if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrices Q and P are represented as products of elementary reflectors:

If $m \geq n$,

$$Q = H(1)*H(2)*\dots*H(n), \text{ and } P = G(1)*G(2)*\dots*G(n-1)$$

Each $H(i)$ and $G(i)$ has the form:

$$H(i) = I - \tau_{uq} v v', \text{ and } G(i) = I - \tau_{up} u u',$$

where τ_{uq} and τ_{up} are real/complex scalars, and v and u are real/complex vectors. $v(1:i-1) = 0$, $v(i) = 1$, and $v(i+1:m)$ is stored on exit in

$$A(ia+i-ia+m-1, a+i-1);$$

$u(1:i) = 0$, $u(i+1) = 1$, and $u(i+2:n)$ is stored on exit in $A(ia+i-1, ja+i+1:ja+n-1)$;

τ_{uq} is stored in $TAUQ(ja+i-1)$ and τ_{up} in $TAUP(ia+i-1)$.

If $m < n$,

$v(1:i) = 0$, $v(i+1) = 1$, and $v(i+2:m)$ is stored on exit in $A(ia+i+1:ia+m-1, ja+i-1)$;

$u(1:i-1) = 0$, $u(i) = 1$, and $u(i+1:n)$ is stored on exit in $A(ia+i-1, ja+i:ja+n-1)$;

τ_{auq} is stored in $TAUQ(ja+i-1)$ and τ_{aup} in $TAUP(ia+i-1)$.

The contents of $\text{sub}(A)$ on exit are illustrated by the following examples:

$m = 6$ and $n = 5$ ($m > n$):

$$\begin{bmatrix} d & e & u1 & u1 & u1 \\ v1 & d & e & u2 & u2 \\ v1 & v2 & d & e & u3 \\ v1 & v2 & v3 & d & e \\ v1 & v2 & v3 & v4 & d \\ v1 & v2 & v3 & v4 & v5 \end{bmatrix}$$

$m = 5$ and $n = 6$ ($m < n$):

$$\begin{bmatrix} d & u1 & u1 & u1 & u1 & u1 \\ e & d & u2 & u2 & u2 & u2 \\ v1 & e & d & u3 & u3 & u3 \\ v1 & v2 & e & d & u4 & u4 \\ v1 & v2 & v3 & e & d & u5 \end{bmatrix}$$

where d and e denote diagonal and off-diagonal elements of B , v_i denotes an element of the vector defining $H(i)$, and u_i an element of the vector defining $G(i)$.

p?gehd2

Reduces a general matrix to upper Hessenberg form by an orthogonal/unitary similarity transformation (unblocked algorithm).

Syntax

call psgehd2($n, ilo, ihi, a, ia, ja, desca, \tau, work, lwork, info$)

call pdgehd2($n, ilo, ihi, a, ia, ja, desca, \tau, work, lwork, info$)

call pcgehd2($n, ilo, ihi, a, ia, ja, desca, \tau, work, lwork, info$)

call pzgehd2($n, ilo, ihi, a, ia, ja, desca, \tau, work, lwork, info$)

Description

This routine reduces a real/complex general distributed matrix $\text{sub}(A)$ to upper Hessenberg form H by an orthogonal/unitary similarity transformation: $Q' * \text{sub}(A) * Q = H$, where $\text{sub}(A) = A(ia+n-1:ia+n-1, ja+n-1:ja+n-1)$.

Input Parameters

<i>n</i>	(global) INTEGER. The order of the distributed submatrix <i>A</i> . ($n \geq 0$).
<i>ilo, ihi</i>	(global) INTEGER. It is assumed that $\text{sub}(A)$ is already upper triangular in rows $ia:ia+ilo-2$ and $ia+ihi:ia+n-1$ and columns $ja:ja+jlo-2$ and $ja+jhi:ja+n-1$. See <i>Application Notes</i> for further information. If $n \geq 0$, $1 \leq ilo \leq ihi \leq n$; otherwise set $ilo = 1$, $ihi = n$.
<i>a</i>	(local). REAL for psgehd2 DOUBLE PRECISION for pdgehd2 COMPLEX for pcgehd2 COMPLEX*16 for pzgehd2. Pointer into the local memory to an array of $\text{DIMENSION}(lld\_a, LOCC(ja+n-1))$. On entry, this array contains the local pieces of the n -by- n general distributed matrix $\text{sub}(A)$ to be reduced.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>A</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, $\text{DIMENSION}(dlen\_)$. The array descriptor for the distributed matrix <i>A</i> .
<i>work</i>	(local). REAL for psgehd2 DOUBLE PRECISION for pdgehd2 COMPLEX for pcgehd2 COMPLEX*16 for pzgehd2. This is a workspace array of $\text{DIMENSION}(lwork)$.
<i>lwork</i>	(local or global). INTEGER. The dimension of the array <i>work</i> .

lwork is local input and must be at least $lwork \geq nb + \max(npa0, nb)$, where $nb = mb\_a = nb\_a$, $iroffa = \text{mod}(ia-1, nb)$, $iarow = \text{indxg2p}(ia, nb, myrow, rsrc\_a, nprow)$, $npa0 = \text{numroc}(ihi+iroffa, nb, myrow, iarow, nprow)$.

indxg2p and *numroc* are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine *blacs_gridinfo*.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by *pserbla*.

Output Parameters

<i>a</i>	(local). On exit, the upper triangle and the first subdiagonal of sub(<i>A</i>) are overwritten with the upper Hessenberg matrix <i>H</i> , and the elements below the first subdiagonal, with the array <i>tau</i> , represent the orthogonal/unitary matrix <i>Q</i> as a product of elementary reflectors. (see <i>Application Notes</i> below).
<i>tau</i>	(local). REAL for <i>psgehd2</i> DOUBLE PRECISION for <i>pdgehd2</i> COMPLEX for <i>pcgehd2</i> COMPLEX*16 for <i>pzgehd2</i> . Array, DIMENSION <i>LOCc</i> (<i>ja+n-2</i>) The scalar factors of the elementary reflectors (see <i>Application Notes</i> below). Elements <i>ja:ja+ilo-2</i> and <i>ja+ihi:ja+n-2</i> of <i>tau</i> are set to zero. <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>work</i>	On exit, <i>work</i> (1) returns the minimal and optimal <i>lwork</i> .
<i>info</i>	(local).INTEGER. If <i>info</i> = 0, the execution is successful. if <i>info</i> < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = - (<i>i</i> *100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

Application Notes

The matrix Q is represented as a product of $(ihi-ilo)$ elementary reflectors

$$Q = H(ilo)*H(ilo+1)*\dots*H(ihi-1).$$

Each $H(i)$ has the form

$$H(i) = I - \tau v v',$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i)=0$, $v(i+1)=1$ and $v(ihi+1:n)=0$; $v(i+2:ihi)$ is stored on exit in $A(ia+ilo+i:ia+ihi-1, ia+ilo+i-2)$, and τ in $\tau(ja+ilo+i-2)$.

The contents of $A(ia:ia+n-1, ja:ja+n-1)$ are illustrated by the following example, with $n = 7$, $ilo = 2$ and $ihi = 6$:

on entry	on exit
$\begin{bmatrix} a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \\ & & & & & & a \end{bmatrix}$	$\begin{bmatrix} a & a & h & h & h & h & a \\ & a & h & h & h & h & a \\ & h & h & h & h & h & h \\ & v2 & h & h & h & h & h \\ & v2 & v3 & h & h & h & h \\ & v2 & v3 & v4 & h & h & h \\ & & & & & & a \end{bmatrix}$

where a denotes an element of the original matrix $\text{sub}(A)$, h denotes a modified element of the upper Hessenberg matrix H , and v_i denotes an element of the vector defining $H(ja+ilo+i-2)$.

p?gelq2

Computes an LQ factorization of a general rectangular matrix (unblocked algorithm).

Syntax

```
call psgelq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgelq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgelq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgelq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Description

This routine computes an LQ factorization of a real/complex distributed m -by- n matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) = L^*Q$.

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$. ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$. ($n \geq 0$).
<i>a</i>	(local). REAL for psgelq2 DOUBLE PRECISION for pdgelq2 COMPLEX for pcgelq2 COMPLEX*16 for pzgelq2. Pointer into the local memory to an array of <code>DIMENSION(lld_a, LOcc(ja+n-1))</code> . On entry, this array contains the local pieces of the m -by- n distributed matrix $\text{sub}(A)$ which is to be factored.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, <code>DIMENSION(dlen_)</code> . The array descriptor for the distributed matrix <i>A</i> .
<i>work</i>	(local). REAL for psgelq2 DOUBLE PRECISION for pdgelq2 COMPLEX for pcgelq2 COMPLEX*16 for pzgelq2. This is a workspace array of <code>DIMENSION(lwork)</code> .
<i>lwork</i>	(local or global) INTEGER. The dimension of the array <i>work</i> .

lwork is local input and must be at least $lwork \geq nq0 + \max(1, mp0)$,

where $irow = \text{mod}(ia-1, mb\_a)$, $icoff = \text{mod}(ja-1, nb\_a)$,

$iarow = \text{indxg2p}(ia, mb\_a, myrow, rsrc\_a, nprow)$,
 $iacol = \text{indxg2p}(ja, nb\_a, mycol, csrc\_a, npcot)$,
 $mp0 = \text{numroc}(m+irow, mb\_a, myrow, iarow, nprow)$,
 $nq0 = \text{numroc}(n+icoff, nb\_a, mycol, iacol, npcot)$,
 indxg2p and numroc are ScaLAPACK tool functions; $myrow$, $mycol$, $nprow$, and $npcot$ can be determined by calling the subroutine `blacs_gridinfo`.

If $lwork = -1$, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a

(local).

On exit, the elements on and below the diagonal of sub(*A*) contain the m by $\min(m, n)$ lower trapezoidal matrix L (L is lower triangular if $m \leq n$); the elements above the diagonal, with the array *tau*, represent the orthogonal/unitary matrix Q as a product of elementary reflectors (see *Application Notes* below).

tau

(local).

REAL for `psgelq2`

DOUBLE PRECISION for `pdgelq2`

COMPLEX for `pcgelq2`

COMPLEX*16 for `pzgelq2`.

Array, DIMENSION $LOCr(ia+\min(m, n)-1)$. This array contains the scalar factors of the elementary reflectors. *tau* is tied to the distributed matrix *A*.

work

On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local).INTEGER. If *info* = 0, the execution is successful. if *info* < 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$Q = H(ia+k-1) * H(ia+k-2) * \dots * H(ia)$ for real flavors, $Q = (H(ia+k-1))^H * (H(ia+k-2))^H \dots * (H(ia))^H$ for complex flavors,

where $k = \min(m, n)$.

Each $H(i)$ has the form

$H(i) = I - \tau v v'$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:n)$ (for real flavors) or $\text{conjg}(v(i+1:n))$ (for complex flavors) is stored on exit in $A(ia+i-1, ja+i:ja+n-1)$, and τ in $TAU(ia+i-1)$.

p?geql2

Computes a QL factorization of a general rectangular matrix (unblocked algorithm).

Syntax

```
call psgeql2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgeql2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgeql2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgeql2(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine computes a QL factorization of a real/complex distributed m -by- n matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) = Q * L$.

Input Parameters

m (global) INTEGER.

The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$. ($m \geq 0$).

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$. ($n \geq 0$).

a (local).
REAL for psgeql2
DOUBLE PRECISION for pdgeql2
COMPLEX for pcgeql2
COMPLEX*16 for pzgeql2.
Pointer into the local memory to an array of DIMENSION (*lld_a*, *LOCc* (*ja*+*n*-1)).
On entry, this array contains the local pieces of the *m*-by-*n* distributed matrix $\text{sub}(A)$ which is to be factored.

ia, ja (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

work (local).
REAL for psgeql2
DOUBLE PRECISION for pdgeql2
COMPLEX for pcgeql2
COMPLEX*16 for pzgeql2.
This is a workspace array of DIMENSION (*lwork*).

lwork (local or global) INTEGER.
The dimension of the array *work*.
lwork is local input and must be at least $lwork \geq mp0 + \max(1, nq0)$,
where $iroff = \text{mod}(ia-1, mb\_a)$, $icoff = \text{mod}(ja-1, nb\_a)$,
 $iarow = \text{indxg2p}(ia, mb\_a, myrow, rsrc\_a, nprow)$,
 $iacol = \text{indxg2p}(ja, nb\_a, mycol, csrc\_a, npc0l)$,
 $mp0 = \text{numroc}(m+iroff, mb\_a, myrow, iarow, nprow)$,
 $nq0 = \text{numroc}(n+icoff, nb\_a, mycol, iacol, npc0l)$,

`indxg2p` and `numroc` are ScaLAPACK tool functions; `myrow`, `mycol`, `nprow`, and `npcol` can be determined by calling the subroutine `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>a</code>	(local). On exit, if $m \geq n$, the lower triangle of the distributed submatrix $A(ia+m-n:ia+m-1, ja:ja+n-1)$ contains the n-by-n lower triangular matrix L; if $m \leq n$, the elements on and below the $(n-m)$-th superdiagonal contain the m-by-n lower trapezoidal matrix L; the remaining elements, with the array <code>tau</code>, represent the orthogonal/ unitary matrix Q as a product of elementary reflectors (see <i>Application Notes</i> below).
<code>tau</code>	(local). REAL for <code>psgeql2</code> DOUBLE PRECISION for <code>pdgeql2</code> COMPLEX for <code>pcgeql2</code> COMPLEX*16 for <code>pzgeql2</code>. Array, DIMENSION <code>LOCc(ja+n-1)</code>. This array contains the scalar factors of the elementary reflectors. <code>tau</code> is tied to the distributed matrix A.
<code>work</code>	On exit, <code>work(1)</code> returns the minimal and optimal <code>lwork</code> .
<code>info</code>	(local). INTEGER. If <code>info = 0</code>, the execution is successful. if <code>info < 0</code>: If the i-th argument is an array and the j-entry had an illegal value, then <code>info = - (i*100+j)</code>, if the i-th argument is a scalar and had an illegal value, then <code>info = -i</code>.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(ja+k-1) * \dots * H(ja+1) * H(ja), \text{ where } k = \min(m, n).$$

Each $H(i)$ has the form

$$H(i) = I - \tau v v'$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(m-k+i+1:m) = 0$ and $v(m-k+i) = 1$; $v(1:m-k+i-1)$ is stored on exit in $A(ia:ia+m-k+i-2, ja+n-k+i-1)$, and τ in $TAU(ja+n-k+i-1)$.

p?geqr2

Computes a QR factorization of a general rectangular matrix (unblocked algorithm).

Syntax

```
call psgeqr2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgeqr2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgeqr2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgeqr2(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Description

This routine computes a QR factorization of a real/complex distributed m -by- n matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) = Q * R$.

Input Parameters

m	(global). INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$. ($m \geq 0$).
n	(global). INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$. ($n \geq 0$).
a	(local). REAL for psgeqr2

DOUBLE PRECISION for pdgeqr2
 COMPLEX for pcgeqr2
 COMPLEX*16 for pzgeqr2.
 Pointer into the local memory to an array of DIMENSION
 (*lld_a*, *LOCc* (*ja+n-1*)).
 On entry, this array contains the local pieces of the *m*-by-*n*
 distributed matrix sub(*A*) which is to be factored.

ia, *ja* (global) INTEGER. The row and column indices in the global
 array *a* indicating the first row and the first column of the
 submatrix *A*, respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The
 array descriptor for the distributed matrix *A*.

work (local).
 REAL for psgeqr2
 DOUBLE PRECISION for pdgeqr2
 COMPLEX for pcgeqr2
 COMPLEX*16 for pzgeqr2.
 This is a workspace array of DIMENSION (*lwork*).

lwork (local or global). INTEGER.
 The dimension of the array *work*.
lwork is local input and must be at least $lwork \geq mp0 + \max(1, nq0)$,
 where $iroff = \text{mod}(ia-1, mb\_a)$, $icoff = \text{mod}(ja-1, nb\_a)$,
 $iarow = \text{indxg2p}(ia, mb\_a, myrow, rsrc\_a, nprow)$,
 $iacol = \text{indxg2p}(ja, nb\_a, mycol, csrc\_a, npcot)$,
 $mp0 = \text{numroc}(m+iroff, mb\_a, myrow, iarow, nprow)$,
 $nq0 = \text{numroc}(n+icoff, nb\_a, mycol, iacol, npcot)$.
indxg2p and *numroc* are ScaLAPACK tool functions; *myrow*,
mycol, *nprow*, and *npcol* can be determined by calling the
 subroutine *blacs_gridinfo*.
 If *lwork* = -1, then *lwork* is global input and a workspace
 query is assumed; the routine only calculates the minimum
 and optimal size for all work arrays. Each of these values
 is returned in the first entry of the corresponding work array,
 and no error message is issued by *pxerbla*.

Output Parameters

<i>a</i>	(local). On exit, the elements on and above the diagonal of $\text{sub}(A)$ contain the $\min(m,n)$ by n upper trapezoidal matrix R (R is upper triangular if $m \geq n$); the elements below the diagonal, with the array <i>tau</i> , represent the orthogonal/unitary matrix Q as a product of elementary reflectors (see <i>Application Notes</i> below).
<i>tau</i>	(local). REAL for psgeqr2 DOUBLE PRECISION for pdgeqr2 COMPLEX for pcgeqr2 COMPLEX*16 for pzgeqr2. Array, DIMENSION $\text{LOCc}(ja+\min(m,n)-1)$. This array contains the scalar factors of the elementary reflectors. <i>tau</i> is tied to the distributed matrix A .
<i>work</i>	On exit, <i>work</i> (1) returns the minimal and optimal <i>lwork</i> .
<i>info</i>	(local). INTEGER. If <i>info</i> = 0, the execution is successful. if <i>info</i> < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = - (<i>i</i> *100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(ja) * H(ja+1) * \dots * H(ja+k-1), \text{ where } k = \min(m,n).$$

Each $H(i)$ has the form

$$H(j) = I - \tau * v * v',$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:m)$ is stored on exit in $A(ia+i:ia+m-1, ja+i-1)$, and τ in $TAU(ja+i-1)$.

p?gerq2

Computes an RQ factorization of a general rectangular matrix (unblocked algorithm).

Syntax

```
call psgerq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgerq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgerq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgerq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Description

This routine computes an RQ factorization of a real/complex distributed m -by- n matrix $\text{sub}(A)$
 $= A(ia:ia+m-1, ja:ja+n-1) = R*Q$.

Input Parameters

m	(global). INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$. ($m \geq 0$).
n	(global). INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$. ($n \geq 0$).
a	(local). REAL for psgerq2 DOUBLE PRECISION for pdgerq2 COMPLEX for pcgerq2 COMPLEX*16 for pzgerq2. Pointer into the local memory to an array of DIMENSION ($lld\_a, LOCC(ja+n-1)$). On entry, this array contains the local pieces of the m -by- n distributed matrix $\text{sub}(A)$ which is to be factored.
ia, ja	(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix A , respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix A.

work (local).
 REAL for *psgerq2*
 DOUBLE PRECISION for *pdgerq2*
 COMPLEX for *pcgerq2*
 COMPLEX*16 for *pzgerq2*.
 This is a workspace array of DIMENSION (*lwork*).

lwork (local or global). INTEGER.
 The dimension of the array *work*.
lwork is local input and must be at least $lwork \geq nq0 + \max(1, mp0)$, where
 $iroff = \text{mod}(ia-1, mb\_a)$, $icoff = \text{mod}(ja-1, nb\_a)$,
 $iarow = \text{indxg2p}(ia, mb\_a, myrow, rsrc\_a, nprow)$,
 $iacol = \text{indxg2p}(ja, nb\_a, mycol, csrc\_a, npc0l)$,
 $mp0 = \text{numroc}(m+iroff, mb\_a, myrow, iarow, nprow)$,
 $nq0 = \text{numroc}(n+icoff, nb\_a, mycol, iacol, npc0l)$,
indxg2p and *numroc* are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npc0l* can be determined by calling the subroutine *blacs_gridinfo*.
 If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by *pxerbla*.

Output Parameters

a (local).
 On exit,
 if $m \leq n$, the upper triangle of $A(ia+m-n:ia+m-1, ja:ja+n-1)$ contains the m -by- m upper triangular matrix R ;

if $m \geq n$, the elements on and above the $(m-n)$ -th subdiagonal contain the m -by- n upper trapezoidal matrix R ; the remaining elements, with the array τ , represent the orthogonal/unitary matrix Q as a product of elementary reflectors (see *Application Notes* below).

τ

(local).

REAL for psgqrq2

DOUBLE PRECISION for pdgqrq2

COMPLEX for pcgqrq2

COMPLEX*16 for pzgqrq2.

Array, DIMENSION $LOCr(ia+m-1)$. This array contains the scalar factors of the elementary reflectors. τ is tied to the distributed matrix A .

$work$

On exit, $work(1)$ returns the minimal and optimal $lwork$.

$info$

(local). INTEGER.

If $info = 0$, the execution is successful.

if $info < 0$: If the i -th argument is an array and the j -entry had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$Q = H(ia) * H(ia+1) * \dots * H(ia+k-1)$ for real flavors,

$Q = (H(ia))^H * (H(ia+1))^H * \dots * (H(ia+k-1))^H$ for complex flavors,

where $k = \min(m, n)$.

Each $H(i)$ has the form

$H(i) = I - \tau * v * v'$,

where τ is a real/complex scalar, and v is a real/complex vector with $v(n-k+i+1:n) = 0$ and $v(n-k+i) = 1$; $v(1:n-k+i-1)$ for real flavors or $\text{conjg}(v(1:n-k+i-1))$ for complex flavors is stored on exit in $A(ia+m-k+i-1, ja:ja+n-k+i-2)$, and τ in $TAU(ia+m-k+i-1)$.

p?getf2

Computes an LU factorization of a general matrix, using partial pivoting with row interchanges (local blocked algorithm).

Syntax

```
call psgetf2(m, n, a, ia, ja, desca, ipiv, info)
call pdgetf2(m, n, a, ia, ja, desca, ipiv, info)
call pcgetf2(m, n, a, ia, ja, desca, ipiv, info)
call pzgetf2(m, n, a, ia, ja, desca, ipiv, info)
```

Description

This routine computes an LU factorization of a general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ using partial pivoting with row interchanges.

The factorization has the form $\text{sub}(A) = P * L * U$, where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$), and U is upper triangular (upper trapezoidal if $m < n$). This is the right-looking Parallel Level 2 BLAS version of the algorithm.

Input Parameters

m	(global). INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$. ($m \geq 0$).
n	(global). INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$. ($nb\_a - \text{mod}(ja-1, nb\_a) \geq n \geq 0$).
a	(local). REAL for psgetf2 DOUBLE PRECISION for pdgetf2 COMPLEX for pcgetf2 COMPLEX*16 for pzgetf2. Pointer into the local memory to an array of DIMENSION(lld_a , $LOCc(ja+n-1)$).

On entry, this array contains the local pieces of the m -by- n distributed matrix $\text{sub}(A)$.

ia, ja (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

Output Parameters

ipiv (local). INTEGER.
Array, DIMENSION ($\text{LOCr}(m\_a) + mb\_a$). This array contains the pivoting information. $ipiv(i) \rightarrow$ The global row that local row *i* was swapped with. This array is tied to the distributed matrix *A*.

info (local). INTEGER.
If *info* = 0: successful exit.
If *info* < 0:

- if the *i*-th argument is an array and the *j*-entry had an illegal value, then $info = -(i*100+j)$,
- if the *i*-th argument is a scalar and had an illegal value, then $info = -i$.

If *info* > 0: If $info = k$, $u(ia+k-1, ja+k-1)$ is exactly zero. The factorization has been completed, but the factor *u* is exactly singular, and division by zero will occur if it is used to solve a system of equations.

p?labrd

Reduces the first nb rows and columns of a general rectangular matrix A to real bidiagonal form by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of A .

Syntax

```
call pslabrd(m, n, nb, a, ia, ja, desca, d, e, tauq, taup, x, ix, jx, descx,
y, iy, jy, descy, work)
```

```
call pdlabrd(m, n, nb, a, ia, ja, desca, d, e, tauq, taup, x, ix, jx, descx,
y, iy, jy, descy, work)
```

```
call pclabrd(m, n, nb, a, ia, ja, desca, d, e, tauq, taup, x, ix, jx, descx,
y, iy, jy, descy, work)
```

```
call pzlabrd(m, n, nb, a, ia, ja, desca, d, e, tauq, taup, x, ix, jx, descx,
y, iy, jy, descy, work)
```

Description

This routine reduces the first nb rows and columns of a real/complex general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ to upper or lower bidiagonal form by an orthogonal/unitary transformation $Q'^* A * P$, and returns the matrices X and Y necessary to apply the transformation to the unreduced part of $\text{sub}(A)$.

If $m \geq n$, $\text{sub}(A)$ is reduced to upper bidiagonal form; if $m < n$, $\text{sub}(A)$ is reduced to lower bidiagonal form.

This is an auxiliary routine called by [p?gebrd](#).

Input Parameters

m	(global). INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$. ($m \geq 0$).
n	(global). INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$. ($n \geq 0$).

<i>nb</i>	(global) INTEGER. The number of leading rows and columns of sub(<i>A</i>) to be reduced.
<i>a</i>	(local). REAL for pslabrd DOUBLE PRECISION for pdlabrd COMPLEX for pclabrd COMPLEX*16 for pzlabrd. Pointer into the local memory to an array of DIMENSION(<i>lld_a</i> , <i>LOC(ja+n-1)</i>). On entry, this array contains the local pieces of the general distributed matrix sub(<i>A</i>).
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix sub(<i>A</i>), respectively.
<i>desca</i>	(global and local) INTEGER array, DIMENSION(<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the global array <i>x</i> indicating the first row and the first column of the submatrix sub(<i>X</i>), respectively.
<i>descx</i>	(global and local) INTEGER array, DIMENSION(<i>dlen_</i>). The array descriptor for the distributed matrix <i>X</i> .
<i>iy, jy</i>	(global) INTEGER. The row and column indices in the global array <i>y</i> indicating the first row and the first column of the submatrix sub(<i>Y</i>), respectively.
<i>descy</i>	(global and local) INTEGER array, DIMENSION(<i>dlen_</i>). The array descriptor for the distributed matrix <i>Y</i> .
<i>work</i>	(local). REAL for pslabrd DOUBLE PRECISION for pdlabrd COMPLEX for pclabrd COMPLEX*16 for pzlabrd Workspace array, DIMENSION(<i>lwork</i>) $lwork \geq nb\_a + nq$, with $nq = \text{numroc}(n + \text{mod}(ia-1, nb\_y), nb\_y, mycol, iacol, npcol)$

`iacol = indxg2p (ja, nb_a, mycol, csrc_a, npcol)`
`indxg2p` and `numroc` are ScaLAPACK tool functions; `myrow`,
`mycol`, `nprow`, and `npcol` can be determined by calling the
subroutine `blacs_gridinfo`.

Output Parameters

- a** (local)
On exit, the first `nb` rows and columns of the matrix are overwritten; the rest of the distributed matrix `sub(A)` is unchanged.
If $m \geq n$, elements on and below the diagonal in the first `nb` columns, with the array `tauq`, represent the orthogonal/unitary matrix Q as a product of elementary reflectors; and elements above the diagonal in the first `nb` rows, with the array `taup`, represent the orthogonal/unitary matrix P as a product of elementary reflectors.
If $m < n$, elements below the diagonal in the first `nb` columns, with the array `tauq`, represent the orthogonal/unitary matrix Q as a product of elementary reflectors, and elements on and above the diagonal in the first `nb` rows, with the array `taup`, represent the orthogonal/unitary matrix P as a product of elementary reflectors. See *Application Notes* below.
- d** (local).
REAL for `pslabrd`
DOUBLE PRECISION for `pdlabrd`
COMPLEX for `pclabrd`
COMPLEX*16 for `pzlabrd`
Array, DIMENSION `LOCr(ia+min(m,n)-1)` if $m \geq n$;
`LOCc(ja+min(m,n)-1)` otherwise. The distributed diagonal elements of the bidiagonal distributed matrix B :
 $d(i) = A(ia+i-1, ja+i-1)$.
`d` is tied to the distributed matrix A .
- e** (local).
REAL for `pslabrd`
DOUBLE PRECISION for `pdlabrd`
COMPLEX for `pclabrd`

COMPLEX*16 for pzlabrd

Array, DIMENSION $LOCr(ia+\min(m,n)-1)$ if $m \geq n$;
 $LOCc(ja+\min(m,n)-2)$ otherwise. The distributed
off-diagonal elements of the bidiagonal distributed matrix
 B :

if $m \geq n$, $E(i) = A(ia+i-1, ja+i)$ for $i = 1, 2, \dots, n-1$;
if $m < n$, $E(i) = A(ia+i, ja+i-1)$ for $i = 1, 2, \dots, m-1$.
 e is tied to the distributed matrix A .

τ_{auq}, τ_{aup} (local).
REAL for pslabrd
DOUBLE PRECISION for pdlabrd
COMPLEX for pclabrd
COMPLEX*16 for pzlabrd
Array DIMENSION $LOCc(ja+\min(m,n)-1)$ for τ_{auq} ,
DIMENSION $LOCr(ia+\min(m,n)-1)$ for τ_{aup} . The scalar
factors of the elementary reflectors which represent the
orthogonal/unitary matrix Q for τ_{auq} , P for τ_{aup} . τ_{auq} and
 τ_{aup} are tied to the distributed matrix A . See *Application
Notes* below.

x (local)
REAL for pslabrd
DOUBLE PRECISION for pdlabrd
COMPLEX for pclabrd
COMPLEX*16 for pzlabrd
Pointer into the local memory to an array of DIMENSION
 (lld_x, nb) . On exit, the local pieces of the distributed
 m -by- nb matrix $X(ix:ix+m-1, jx:jx+nb-1)$ required to
update the unreduced part of $\text{sub}(A)$.

y (local).
REAL for pslabrd
DOUBLE PRECISION for pdlabrd
COMPLEX for pclabrd
COMPLEX*16 for pzlabrd

Pointer into the local memory to an array of `DIMENSION` (`lld_y, nb`). On exit, the local pieces of the distributed n -by- nb matrix $Y(iy:iy+n-1, jy:jy+nb-1)$ required to update the unreduced part of $\text{sub}(A)$.

Application Notes

The matrices Q and P are represented as products of elementary reflectors:

$$Q = H(1) * H(2) * \dots * H(nb), \text{ and } P = G(1) * G(2) * \dots * G(nb)$$

Each $H(i)$ and $G(i)$ has the form:

$$H(i) = I - \text{tauq} * v * v', \text{ and } G(i) = I - \text{taup} * u * u',$$

where tauq and taup are real/complex scalars, and v and u are real/complex vectors.

If $m \geq n$, $v(1:i-1) = 0$, $v(i) = 1$, and $v(i:m)$ is stored on exit in

$A(ia+i-1:ia+m-1, ja+i-1)$; $u(1:i) = 0$, $u(i+1) = 1$, and $u(i+1:n)$ is stored on exit in $A(ia+i-1, ja+i:ja+n-1)$; tauq is stored in $\text{TAUQ}(ja+i-1)$ and taup in $\text{TAUP}(ia+i-1)$.

If $m < n$, $v(1:i) = 0$, $v(i+1) = 1$, and $v(i+1:m)$ is stored on exit in

$A(ia+i+1:ia+m-1, ja+i-1)$; $u(1:i-1) = 0$, $u(i) = 1$, and $u(i:n)$ is stored on exit in $A(ia+i-1, ja+i:ja+n-1)$; tauq is stored in $\text{TAUQ}(ja+i-1)$ and taup in $\text{TAUP}(ia+i-1)$. The elements of the vectors v and u together form the m -by- nb matrix V and the nb -by- n matrix U' which are necessary, with X and Y , to apply the transformation to the unreduced part of the matrix, using a block update of the form: $\text{sub}(A) := \text{sub}(A) - V * Y' - X * U'$. The contents of $\text{sub}(A)$ on exit are illustrated by the following examples with $nb = 2$:

$m = 6$ and $n = 5 (m > n)$:

$$\begin{bmatrix} 1 & 1 & v1 & u1 & u1 \\ v1 & 1 & 1 & u2 & u2 \\ v1 & v2 & a & a & a \\ v1 & v2 & a & a & a \\ v1 & v2 & a & a & a \\ v1 & v2 & a & a & a \end{bmatrix}$$

$m = 5$ and $n = 6 (m < n)$:

$$\begin{bmatrix} 1 & u1 & u1 & u1 & u1 & u1 \\ 1 & 1 & u2 & u2 & u2 & u2 \\ v1 & 1 & a & a & a & a \\ v1 & v2 & a & a & a & a \\ v1 & v2 & a & a & a & a \end{bmatrix}$$

where a denotes an element of the original matrix which is unchanged, vi denotes an element of the vector defining $H(i)$, and ui an element of the vector defining $G(i)$.

p?lacon

Estimates the 1-norm of a square matrix, using the reverse communication for evaluating matrix-vector products.

Syntax

```
call pslacon(n, v, iv, jv, descv, x, ix, jx, descx, isgn, est, kase)
call pdlacon(n, v, iv, jv, descv, x, ix, jx, descx, isgn, est, kase)
call pclacon(n, v, iv, jv, descv, x, ix, jx, descx, isgn, est, kase)
call pzlacon(n, v, iv, jv, descv, x, ix, jx, descx, isgn, est, kase)
```

Description

This routine estimates the 1-norm of a square, real/unitary distributed matrix *A*. Reverse communication is used for evaluating matrix-vector products. *x* and *v* are aligned with the distributed matrix *A*, this information is implicitly contained within *iv*, *ix*, *descv*, and *descx*.

Input Parameters

<i>n</i>	(global).INTEGER. The length of the distributed vectors <i>v</i> and <i>x</i> . $n \geq 0$.
<i>v</i>	(local). REAL for pslacon DOUBLE PRECISION for pdlacon COMPLEX for pclacon COMPLEX*16 for pzlacon. Pointer into the local memory to an array of DIMENSION <i>LOCr</i> ($n + \text{mod}(iv-1, mb\_v)$). On the final return, $v = a * w$, where $est = \text{norm}(v) / \text{norm}(w)$ (<i>w</i> is not returned).
<i>iv, jv</i>	(global) INTEGER. The row and column indices in the global array <i>v</i> indicating the first row and the first column of the submatrix <i>V</i> , respectively.
<i>descv</i>	(global and local) INTEGER array, DIMENSION (<i>dlen_</i>). The array descriptor for the distributed matrix <i>V</i> .
<i>x</i>	(local). REAL for pslacon

DOUBLE PRECISION for pdlacon
 COMPLEX for pclacon
 COMPLEX*16 for pzlacon.
 Pointer into the local memory to an array of DIMENSION
 $LOCr(n+mod(ix-1, mb_x))$.

ix, jx (global) INTEGER. The row and column indices in the global array *x* indicating the first row and the first column of the submatrix *x*, respectively.

descx (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix X.

isgn (local). INTEGER.
 Array, DIMENSION $LOCr(n+mod(ix-1, mb_x))$. *isgn* is aligned with *x* and *v*.

kase (local). INTEGER.
 On the initial call to p?lacon, *kase* should be 0.

Output Parameters

x (local).
 On an intermediate return, X should be overwritten by A^*X , if *kase*=1, $A' * X$, if *kase*=2, p?lacon must be re-called with all the other parameters unchanged.

est (global). REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors

kase (local)
 INTEGER. On an intermediate return, *kase* is 1 or 2, indicating whether *x* should be overwritten by A^*X , or $A' * X$. On the final return from p?lacon, *kase* is again 0.

p?laconsb

Looks for two consecutive small subdiagonal elements.

Syntax

```
call pslaconsb(a, desca, i, l, m, h44, h33, h43h34, buf, lwork)
call pdlaconsb(a, desca, i, l, m, h44, h33, h43h34, buf, lwork)
```

Description

This routine looks for two consecutive small subdiagonal elements by analyzing the effect of starting a double shift QR iteration given by *h44*, *h33*, and *h43h34* to see if this process makes a subdiagonal negligible.

Input Parameters

<i>a</i>	(global). REAL for pslaconsb DOUBLE PRECISION for pdlaconsb Array, DIMENSION (<i>desca</i> (<i>lld_</i>),*). On entry, the Hessenberg matrix whose tridiagonal part is being scanned. Unchanged on exit.
<i>desca</i>	(global and local) INTEGER. Array of DIMENSION (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>i</i>	(global) INTEGER. The global location of the bottom of the unreduced submatrix of <i>A</i> . Unchanged on exit.
<i>l</i>	(global) INTEGER. The global location of the top of the unreduced submatrix of <i>A</i> . Unchanged on exit.
<i>h44, h33, h43h34</i>	(global). REAL for pslaconsb DOUBLE PRECISION for pdlaconsb These three values are for the double shift QR iteration.
<i>lwork</i>	(global) INTEGER. This must be at least $7 * \text{ceil}(\text{ceil}((i-1)/hbl) / \text{lcm}(nprow, npcol))$. Here <i>lcm</i> is least common multiple and <i>nprowxnpcol</i> is the logical grid size.

Output Parameters

<i>m</i>	(global). On exit, this yields the starting location of the <i>QR</i> double shift. This will satisfy: $1 \leq m \leq i-2$.
<i>buf</i>	(local). REAL for <i>pslaconsb</i> DOUBLE PRECISION for <i>pdlaconsb</i> Array of size <i>lwork</i> .
<i>lwork</i>	(global). On exit, <i>lwork</i> is the size of the work buffer.

p?lacp2

Copies all or part of a distributed matrix to another distributed matrix.

Syntax

```
call pslacp2(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pdlacp2(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pclacp2(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pzlacp2(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
```

Description

This routine copies all or part of a distributed matrix *A* to another distributed matrix *B*. No communication is performed, *p?lacp2* performs a local copy $\text{sub}(A) := \text{sub}(B)$, where $\text{sub}(A)$ denotes $A(ia:ia+m-1, a:ja+n-1)$ and $\text{sub}(B)$ denotes $B(ib:ib+m-1, jb:jb+n-1)$.

p?lacp2 requires that only dimension of the matrix operands is distributed.

Input Parameters

<i>uplo</i>	(global) CHARACTER. Specifies the part of the distributed matrix $\text{sub}(A)$ to be copied: = 'U': Upper triangular part is copied; the strictly lower triangular part of $\text{sub}(A)$ is not referenced; = 'L': Lower triangular part is copied; the strictly upper triangular part of $\text{sub}(A)$ is not referenced.

	Otherwise: all of the matrix $\text{sub}(A)$ is copied.
<i>m</i>	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$. ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$. ($n \geq 0$).
<i>a</i>	(local). REAL for pslacp2 DOUBLE PRECISION for pdlacp2 COMPLEX for pclacp2 COMPLEX*16 for pzlapc2. Pointer into the local memory to an array of <code>DIMENSION (lld_a, LOcc(ja+n-1))</code> . On entry, this array contains the local pieces of the m -by- n distributed matrix $\text{sub}(A)$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>A</i> indicating the first row and the first column of $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array, <code>DIMENSION (dlen_)</code> . The array descriptor for the distributed matrix <i>A</i> .
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the global array <i>B</i> indicating the first row and the first column of $\text{sub}(B)$, respectively.
<i>descb</i>	(global and local) INTEGER array, <code>DIMENSION (dlen_)</code> . The array descriptor for the distributed matrix <i>B</i> .

Output Parameters

<i>b</i>	(local). REAL for pslacp2 DOUBLE PRECISION for pdlacp2 COMPLEX for pclacp2 COMPLEX*16 for pzlapc2.

Pointer into the local memory to an array of `DIMENSION` (`lld_b, LOCC(jb+n-1)`). This array contains on exit the local pieces of the distributed matrix `sub( B )` set as follows:
 if `uplo = 'U'`, $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1)$,
 $1 \leq i \leq j, 1 \leq j \leq n$;
 if `uplo = 'L'`, $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1)$,
 $j \leq i \leq m, 1 \leq j \leq n$;
 otherwise, $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1)$, $1 \leq i \leq m, 1 \leq j \leq n$.

p?lACP3

Copies from a global parallel array into a local replicated array or vice versa.

Syntax

```
call pslACP3(m, i, a, desca, b, ldb, ii, jj, rev)
```

```
call pdlACP3(m, i, a, desca, b, ldb, ii, jj, rev)
```

Description

This is an auxiliary routine that copies from a global parallel array into a local replicated array or vice versa. Note that the entire submatrix that is copied gets placed on one node or more. The receiving node can be specified precisely, or all nodes can receive, or just one row or column of nodes.

Input Parameters

<i>m</i>	(global) INTEGER. <i>m</i> is the order of the square submatrix that is copied. $m \geq 0$. Unchanged on exit.
<i>i</i>	(global) INTEGER. $A(i, i)$ is the global location that the copying starts from. Unchanged on exit.
<i>a</i>	(global). REAL for pslACP3 DOUBLE PRECISION for pdlACP3 Array, DIMENSION (<code>desca(lld_)</code> ,*). On entry, the parallel matrix to be copied into or from.

<i>desca</i>	(global and local) INTEGER array, DIMENSION (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>b</i>	(local). REAL for pslacp3 DOUBLE PRECISION for pdlacp3 Array, DIMENSION (<i>ldb</i> , <i>m</i>). If <i>rev</i> = 0, this is the global portion of the array <i>A</i> (<i>i</i> : <i>i+m-1</i> , <i>i</i> : <i>i+m-1</i>). If <i>rev</i> = 1, this is the unchanged on exit.
<i>ldb</i>	(local) INTEGER. The leading dimension of <i>B</i> .
<i>ii</i>	(global) INTEGER. By using <i>rev</i> 0 and 1, data can be sent out and returned again. If <i>rev</i> = 0, then <i>ii</i> is destination row index for the node(s) receiving the replicated <i>B</i> . If <i>ii</i> ≥ 0, <i>jj</i> ≥ 0, then node (<i>ii</i> , <i>jj</i>) receives the data. If <i>ii</i> = -1, <i>jj</i> ≥ 0, then all rows in column <i>jj</i> receive the data. If <i>ii</i> ≥ 0, <i>jj</i> = -1, then all cols in row <i>ii</i> receive the data. If <i>ii</i> = -1, <i>jj</i> = -1, then all nodes receive the data. If <i>rev</i> != 0, then <i>ii</i> is the source row index for the node(s) sending the replicated <i>B</i> .
<i>jj</i>	(global) INTEGER. Similar description as <i>ii</i> above.
<i>rev</i>	(global) INTEGER. Use <i>rev</i> = 0 to send global <i>A</i> into locally replicated <i>B</i> (on node (<i>ii</i> , <i>jj</i>)). Use <i>rev</i> != 0 to send locally replicated <i>B</i> from node (<i>ii</i> , <i>jj</i>) to its owner (which changes depending on its location in <i>A</i>) into the global <i>A</i> .

Output Parameters

<i>a</i>	(global). On exit, if <i>rev</i> = 1, the copied data. Unchanged on exit if <i>rev</i> = 0.
<i>b</i>	(local). If <i>rev</i> = 1, this is unchanged on exit.

p?lacpy

Copies all or part of one two-dimensional array to another.

Syntax

```
call pslacpy(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pdlacpy(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pclacpy(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pzlacpy(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
```

Description

This routine copies all or part of a distributed matrix A to another distributed matrix B . No communication is performed, `p?lacpy` performs a local copy $\text{sub}(A) := \text{sub}(B)$, where $\text{sub}(A)$ denotes $A(ia:ia+m-1, ja:ja+n-1)$ and $\text{sub}(B)$ denotes $B(ib:ib+m-1, jb:jb+n-1)$.

Input Parameters

<i>uplo</i>	(global). CHARACTER. Specifies the part of the distributed matrix $\text{sub}(A)$ to be copied: = 'U': Upper triangular part; the strictly lower triangular part of $\text{sub}(A)$ is not referenced; = 'L': Lower triangular part; the strictly upper triangular part of $\text{sub}(A)$ is not referenced. Otherwise: all of the matrix $\text{sub}(A)$ is copied.
<i>m</i>	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$. ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$. ($n \geq 0$).
<i>a</i>	(local). REAL for <code>pslacpy</code> DOUBLE PRECISION for <code>pdlacpy</code> COMPLEX for <code>pclacpy</code>

	COMPLEX*16 for pzlapcy. Pointer into the local memory to an array of DIMENSION (lld_a, LOcc(ja+n-1)). On entry, this array contains the local pieces of the distributed matrix sub(A).
ia, ja	(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix sub(A), respectively.
desca	(global and local) INTEGER array, DIMENSION (dlen_). The array descriptor for the distributed matrix A.
ib, jb	(global) INTEGER. The row and column indices in the global array B indicating the first row and the first column of sub(B) respectively.
descb	(global and local) INTEGER array, DIMENSION (dlen_). The array descriptor for the distributed matrix A.

Output Parameters

b	(local). REAL for pslapcy DOUBLE PRECISION for pdlapcy COMPLEX for pclapcy COMPLEX*16 for pzlapcy. Pointer into the local memory to an array of DIMENSION (lld_b, LOcc(jb+n-1)). This array contains on exit the local pieces of the distributed matrix sub(B) set as follows: if uplo = 'U', $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1)$, $1 \leq i \leq j, 1 \leq j \leq n$; if uplo = 'L', $B(ib+i-1, jb+j-1) = A(ia+i-1,$ $ja+j-1), j \leq i \leq m, 1 \leq j \leq n$; otherwise, $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1), 1 \leq i \leq m,$ $1 \leq j \leq n$.

p?laevswp

Moves the eigenvectors from where they are computed to ScaLAPACK standard block cyclic array.

Syntax

```
call pslaevswp(n, zin, ldzi, z, iz, jz, descz, nvs, key, rwork, lrwork)
call pdlaevswp(n, zin, ldzi, z, iz, jz, descz, nvs, key, rwork, lrwork)
call pclaevswp(n, zin, ldzi, z, iz, jz, descz, nvs, key, rwork, lrwork)
call pzlaevswp(n, zin, ldzi, z, iz, jz, descz, nvs, key, rwork, lrwork)
```

Description

This routine moves the eigenvectors (potentially unsorted) from where they are computed, to a ScaLAPACK standard block cyclic array, sorted so that the corresponding eigenvalues are sorted.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

n (global). INTEGER.
The order of the matrix *A*. $n \geq 0$.

zin (local).
REAL for pslaevswp
DOUBLE PRECISION for pdlaevswp
COMPLEX for pclaevswp
COMPLEX*16 for pzlaevswp. Array, DIMENSION (*ldzi*, *nvs*(*iam*)). The eigenvectors on input. Each eigenvector resides entirely in one process. Each process holds a contiguous set of *nvs*(*iam*) eigenvectors. The first eigenvector which the process holds is: sum for $i=[0, iam-1]$ of *nvs*(*i*).

ldzi (local)
INTEGER. The leading dimension of the *zin* array.

<i>iz, jz</i>	(global) INTEGER. The row and column indices in the global array <i>z</i> indicating the first row and the first column of the submatrix <i>z</i> , respectively.
<i>descz</i>	(global and local) INTEGER array, DIMENSION (<i>dlen_</i>). The array descriptor for the distributed matrix <i>Z</i> .
<i>nvs</i>	(global) INTEGER. Array, DIMENSION(<i>nprocs</i> +1) <i>nvs</i> (<i>i</i>) = number of processes number of eigenvectors held by processes [0, <i>i</i> -1] <i>nvs</i> (1) = number of eigen vectors held by [0, 1 -1) = 0 <i>nvs</i> (<i>nprocs</i> +1)= number of eigen vectors held by [0, <i>nprocs</i>)= total number of eigenvectors.
<i>key</i>	(global) INTEGER. Array, DIMENSION (<i>n</i>). Indicates the actual index (after sorting) for each of the eigenvectors.
<i>rwork</i>	(local). REAL for <i>pslaevswp</i> DOUBLE PRECISION for <i>pdlaevswp</i> COMPLEX for <i>pclaevswp</i> COMPLEX*16 for <i>pzlaevswp</i> . Array, DIMENSION (<i>lrwork</i>).
<i>lrwork</i>	(local) INTEGER. Dimension of <i>work</i> .

Output Parameters

<i>z</i>	(local). REAL for <i>pslaevswp</i> DOUBLE PRECISION for <i>pdlaevswp</i> COMPLEX for <i>pclaevswp</i> COMPLEX*16 for <i>pzlaevswp</i> . Array, global DIMENSION (<i>n, n</i>), local DIMENSION (<i>descz</i> (<i>dlen_</i>), <i>nq</i>). The eigenvectors on output. The eigenvectors are distributed in a block cyclic manner in both dimensions, with a block size of <i>nb</i> .

p?lahrd

Reduces the first nb columns of a general rectangular matrix A so that elements below the k -th subdiagonal are zero, by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of A .

Syntax

```
call pslahrd(n, k, nb, a, ia, ja, desca, tau, t, y, iy, jy, descy, work)
call pdlahrd(n, k, nb, a, ia, ja, desca, tau, t, y, iy, jy, descy, work)
call pclahrd(n, k, nb, a, ia, ja, desca, tau, t, y, iy, jy, descy, work)
call pzlahrd(n, k, nb, a, ia, ja, desca, tau, t, y, iy, jy, descy, work)
```

Description

The routines reduces the first nb columns of a real general n -by- $(n-k+1)$ distributed matrix $A(ia:ia+n-1, ja:ja+n-k)$ so that elements below the k -th subdiagonal are zero. The reduction is performed by an orthogonal/unitary similarity transformation $Q' * A * Q$. The routine returns the matrices V and T which determine Q as a block reflector $I - V * T * V'$, and also the matrix $Y = A * V * T$.

This is an auxiliary routine called by `p?gehrd`. In the following comments `sub(A)` denotes $A(ia:ia+n-1, ja:ja+n-1)$.

Input Parameters

n	(global) INTEGER. The order of the distributed submatrix <code>sub(A)</code> . $n \geq 0$.
k	(global) INTEGER. The offset for the reduction. Elements below the k -th subdiagonal in the first nb columns are reduced to zero.
nb	(global) INTEGER. The number of columns to be reduced.
a	(local). REAL for <code>pslahrd</code> DOUBLE PRECISION for <code>pdlahrd</code>

	COMPLEX for pclahrd COMPLEX*16 for pzlahrd. Pointer into the local memory to an array of DIMENSION (<i>lld_a</i> , <i>LOCc</i> (<i>ja+n-k</i>)). On entry, this array contains the local pieces of the <i>n</i> -by- <i>(n-k+1)</i> general distributed matrix <i>A</i> (<i>ia:ia+n-1</i> , <i>ja:ja+n-k</i>).
<i>ia</i> , <i>ja</i>	(global) INTEGER. The row and column indices in the global array <i>A</i> indicating the first row and the first column of the submatrix sub(<i>A</i>), respectively.
<i>desca</i>	(global and local) INTEGER array, DIMENSION (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>iy</i> , <i>jy</i>	(global) INTEGER. The row and column indices in the global array <i>Y</i> indicating the first row and the first column of the submatrix sub(<i>Y</i>), respectively.
<i>descy</i>	(global and local) INTEGER array, DIMENSION (<i>dlen_</i>). The array descriptor for the distributed matrix <i>Y</i> .
<i>work</i>	(local). REAL for pslahrd DOUBLE PRECISION for pdlahrd COMPLEX for pclahrd COMPLEX*16 for pzlahrd. Array, DIMENSION (<i>nb</i>).

Output Parameters

<i>a</i>	(local). On exit, the elements on and above the <i>k</i> -th subdiagonal in the first <i>nb</i> columns are overwritten with the corresponding elements of the reduced distributed matrix; the elements below the <i>k</i> -th subdiagonal, with the array <i>tau</i> , represent the matrix <i>Q</i> as a product of elementary reflectors. The other columns of <i>A</i> (<i>ia:ia+n-1</i> , <i>ja:ja+n-k</i>) are unchanged. (See <i>Application Notes</i> below.)
<i>tau</i>	(local) REAL for pslahrd DOUBLE PRECISION for pdlahrd COMPLEX for pclahrd

COMPLEX*16 for pzlahrd.
 Array, DIMENSION $LOCc(ja+n-2)$. The scalar factors of the elementary reflectors (see *Application Notes* below). τ is tied to the distributed matrix A .

t (local) REAL for pslahrd
 DOUBLE PRECISION for pdlahrd
 COMPLEX for pclahrd
 COMPLEX*16 for pzlahrd.
 Array, DIMENSION (nb_a, nb_a) The upper triangular matrix T .

y (local).
 REAL for pslahrd
 DOUBLE PRECISION for pdlahrd
 COMPLEX for pclahrd
 COMPLEX*16 for pzlahrd.
 Pointer into the local memory to an array of DIMENSION (lld_y, nb_a) . On exit, this array contains the local pieces of the n -by- nb distributed matrix Y . $lld\_y \geq LOCr(ia+n-1)$.

Application Notes

The matrix Q is represented as a product of nb elementary reflectors

$$Q = H(1) * H(2) * \dots * H(nb).$$

Each $H(i)$ has the form

$$H(i) = I - \tau v v',$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i+k-1) = 0$, $v(i+k) = 1$; $v(i+k+1:n)$ is stored on exit in $A(ia+i+k:ia+n-1, ja+i-1)$, and τ in $TAU(ja+i-1)$.

The elements of the vectors v together form the $(n-k+1)$ -by- nb matrix V which is needed, with T and Y , to apply the transformation to the unreduced part of the matrix, using an update of the form: $A(ia:ia+n-1, ja:ja+n-k) := (I - V * T * V') * (A(ia:ia+n-1, ja:ja+n-k) - Y * V')$. The contents of $A(ia:ia+n-1, ja:ja+n-k)$ on exit are illustrated by the following example with $n = 7$, $k = 3$, and $nb = 2$:

$$\begin{bmatrix} a & h & a & a & a \\ a & h & a & a & a \\ a & h & a & a & a \\ h & h & a & a & a \\ v1 & h & a & a & a \\ v1 & v2 & a & a & a \\ v1 & v2 & a & a & a \end{bmatrix}$$

where a denotes an element of the original matrix $A(ia:ia+n-1, ja:ja+n-k)$, h denotes a modified element of the upper Hessenberg matrix H , and vi denotes an element of the vector defining $H(i)$.

p?laiect

Exploits IEEE arithmetic to accelerate the computations of eigenvalues. (C interface function).

Syntax

```
void pslaieict(float *sigma, int *n, float *d, int *count);
void pdlaieictb(float *sigma, int *n, float *d, int *count);
void pdlaieictl(float *sigma, int *n, float *d, int *count);
```

Description

This routine computes the number of negative eigenvalues of $(A - \sigma I)$. This implementation of the Sturm Sequence loop exploits IEEE arithmetic and has no conditionals in the innermost loop. The signbit for real routine `pslaieict` is assumed to be bit 32. Double precision routines `pdlaieictb` and `pdlaieictl` differ in the order of the double precision word storage and, consequently, in the signbit location. For `pdlaieictb`, the double precision word is stored in the big-endian word order and the signbit is assumed to be bit 32. For `pdlaieictl`, the double precision word is stored in the little-endian word order and the signbit is assumed to be bit 64.

Note that all arguments are call-by-reference so that this routine can be directly called from Fortran code.

This is a ScaLAPACK internal subroutine and arguments are not checked for unreasonable values.

Input Parameters

sigma Real for `pslaiect`
DOUBLE PRECISION for `pdlaiectb`/`pdlaiectl`.
The shift. `p?laiect` finds the number of eigenvalues less than equal to *sigma*.

n INTEGER. The order of the tridiagonal matrix *T*. $n \geq 1$.

d Real for `pslaiect`
DOUBLE PRECISION for `pdlaiectb`/`pdlaiectl`.
Array of DIMENSION $(2n - 1)$.
On entry, this array contains the diagonals and the squares of the off-diagonal elements of the tridiagonal matrix *T*. These elements are assumed to be interleaved in memory for better cache performance. The diagonal entries of *T* are in the entries $d(1), d(3), \dots, d(2n-1)$, while the squares of the off-diagonal entries are $d(2), d(4), \dots, d(2n-2)$. To avoid overflow, the matrix must be scaled so that its largest entry is no greater than $overflow^{(1/2)} * underflow^{(1/4)}$ in absolute value, and for greatest accuracy, it should not be much smaller than that.

Output Parameters

n INTEGER. The count of the number of eigenvalues of *T* less than or equal to *sigma*.

p?lange

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a general rectangular matrix.

Syntax

```
val = pslange(norm, m, n, a, ia, ja, desca, work)
val = pdlange(norm, m, n, a, ia, ja, desca, work)
val = pclange(norm, m, n, a, ia, ja, desca, work)
val = pzlange(norm, m, n, a, ia, ja, desca, work)
```

Description

The routine returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$.

Input Parameters

<i>norm</i>	(global) CHARACTER. Specifies what value is returned by the routine: = 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A, it s not a matrix norm. = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum), = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum), = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<i>m</i>	(global). INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$. When $m = 0$, <code>p?lange</code> is set to zero. $m \geq 0$.
<i>n</i>	(global). INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$. When $n = 0$, <code>p?lange</code> is set to zero. $n \geq 0$.
<i>a</i>	(local). Real for <code>pslange</code> DOUBLE PRECISION for <code>pdlange</code> COMPLEX for <code>pclange</code> COMPLEX*16 for <code>pzlange</code>. Pointer into the local memory to an array of DIMENSION (<code>lld_a</code>, <code>LOCc(ja+n-1)</code>) containing the local pieces of the distributed matrix $\text{sub}(A)$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix A.

work (local).
 Real for pslange
 DOUBLE PRECISION for pdlange
 COMPLEX for pclange
 COMPLEX*16 for pzlange.
 Array DIMENSION (*lwork*).
lwork ≥ 0 if *norm* = 'M' or 'm' (not referenced),
*nq*0 if *norm* = '1', 'O' or 'o',
*mp*0 if *norm* = 'I' or 'i',
 0 if *norm* = 'F', 'f', 'E' or 'e' (not referenced),
 where
iroffa = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*),
iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*),
*mp*0 = numroc(*m*+*iroffa*, *mb_a*, *myrow*, *iarow*, *nprow*),
*nq*0 = numroc(*n*+*icoffa*, *nb_a*, *mycol*, *iacol*, *npcol*),
 indxg2p and numroc are ScaLAPACK tool routines; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine `blacs_gridinfo`.

Output Parameters

val The value returned by the routine.

p?lanhs

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of an upper Hessenberg matrix.

Syntax

```
val = pslanhs(norm, n, a, ia, ja, desca, work)
val = pdlanhs(norm, n, a, ia, ja, desca, work)
val = pclanhs(norm, n, a, ia, ja, desca, work)
val = pzlanhs(norm, n, a, ia, ja, desca, work)
```

Description

The routine returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an upper Hessenberg distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$.

Input Parameters

<i>norm</i>	CHARACTER*1. Specifies the value to be returned by the routine: = 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A. = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum), = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum), = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<i>n</i>	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$. When $n = 0$, p?lanhs is set to zero. $n \geq 0$.
<i>a</i>	(local). Real for pslanhs DOUBLE PRECISION for pdlanhs COMPLEX for pclanhs

COMPLEX*16 for pzlanhs.
 Pointer into the local memory to an array of
 DIMENSION(*lld_a*, *LOCc(ja+n-1)*) containing the local
 pieces of the distributed matrix sub(*A*).

ia, ja (global) INTEGER.
 The row and column indices in the global array *A* indicating
 the first row and the first column of the submatrix sub(*A*),
 respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The
 array descriptor for the distributed matrix *A*.

work (local).
 Real for pslanhs
 DOUBLE PRECISION for pdlanhs
 COMPLEX for pclanhs
 COMPLEX*16 for pzlanh.
 Array, DIMENSION (*lwork*).
lwork ≥ 0 if *norm* = 'M' or 'm' (not referenced),
*nq*0 if *norm* = '1', 'O' or 'o',
*mp*0 if *norm* = 'I' or 'i',
 0 if *norm* = 'F', 'f', 'E' or 'e' (not referenced),
 where
iroffa = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*),
iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*),
*mp*0 = numroc(*m*+*iroffa*, *mb_a*, *myrow*, *iarow*, *nprow*),
*nq*0 = numroc(*n*+*icoffa*, *nb_a*, *mycol*, *iacol*, *npcol*),
 indxg2p and numroc are ScaLAPACK tool routines; *myrow*,
mycol, *nprow*, and *npcol* can be determined by calling the
 subroutine *blacs_gridinfo*.

Output Parameters

val The value returned by the fuction.

p?lansy, p?lanhe

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a real symmetric or a complex Hermitian matrix.

Syntax

```
val = pslansy(norm, uplo, n, a, ia, ja, desca, work)
val = pdlansy(norm, uplo, n, a, ia, ja, desca, work)
val = pclansy(norm, uplo, n, a, ia, ja, desca, work)
val = pzlansy(norm, uplo, n, a, ia, ja, desca, work)
val = pclanhe(norm, uplo, n, a, ia, ja, desca, work)
val = pzlanhe(norm, uplo, n, a, ia, ja, desca, work)
```

Description

The routines return the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$.

Input Parameters

<i>norm</i>	(global) CHARACTER. Specifies what value is returned by the routine: = 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A, it s not a matrix norm. = '1' or 'O' or 'o': $val = \text{norml}(A)$, 1-norm of the matrix A (maximum column sum), = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum), = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<i>uplo</i>	(global) CHARACTER. Specifies whether the upper or lower triangular part of the symmetric matrix $\text{sub}(A)$ is to be referenced. = 'U': Upper triangular part of $\text{sub}(A)$ is referenced,

= 'L': Lower triangular part of sub(A) is referenced.

n (global) INTEGER.
The number of columns to be operated on i.e the number of columns of the distributed submatrix sub(A). When $n = 0$, p?lansy is set to zero. $n \geq 0$.

a (local).
Real for pslansy
DOUBLE PRECISION for pdlansy
COMPLEX for pclansy, pplanhe
COMPLEX*16 for pzlanse, pzlanhe.
Pointer into the local memory to an array of DIMENSION (lld_a, LOCC(ja+n-1)) containing the local pieces of the distributed matrix sub(A).
If uplo = 'U', the leading n -by- n upper triangular part of sub(A) contains the upper triangular matrix whose norm is to be computed, and the strictly lower triangular part of this matrix is not referenced. If uplo = 'L', the leading n -by- n lower triangular part of sub(A) contains the lower triangular matrix whose norm is to be computed, and the strictly upper triangular part of sub(A) is not referenced.

ia, ja (global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of the submatrix sub(A), respectively.

desca (global and local) INTEGER array, DIMENSION (dlen_). The array descriptor for the distributed matrix A.

work (local).
Real for pslansy
DOUBLE PRECISION for pdlansy
COMPLEX for pclansy, pplanhe
COMPLEX*16 for pzlanse, pzlanhe.
Array DIMENSION (lwork).
 $lwork \geq 0$ if norm = 'M' or 'm' (not referenced),
 $2*nq0+np0+ldw$ if norm = '1', 'O' or 'o', 'I' or 'i',
where ldw is given by:
if(nprow.ne.npcol) then
ldw = mb_a*ceil(ceil(np0/mb_a)/(lcm/nprow))
else

```
ldw = 0
end if
0 if norm = 'F', 'f', 'E' or 'e' (not referenced),
where lcm is the least common multiple of nprow and
npcol, lcm = ilcm( nprow, npcol ) and ceil denotes
the ceiling operation (iceil).
iroffa = mod(ia-1, mb_a ), icoffa = mod( ja-1, nb_a),
iarow = indxg2p(ia, mb_a, myrow, rsrc_a, nprow),
iacol = indxg2p(ja, nb_a, mycol, csrc_a, npcol),
mp0 = numroc(m+iroffa, mb_a, myrow, iarow, nprow),
nq0 = numroc(n+icoffa, nb_a, mycol, iacol, npcol),
indxg2p and numroc are ScaLAPACK tool functions; myrow,
mycol, nprow, and npcol can be determined by calling the
subroutine blacs_gridinfo.
```

Output Parameters

val The value returned by the routine.

p?lantr

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a triangular matrix.

Syntax

```
val = pslantr(norm, uplo, diag, m, n, a, ia, ja, desca, work)
val = pdlantr(norm, uplo, diag, m, n, a, ia, ja, desca, work)
val = pclantr(norm, uplo, diag, m, n, a, ia, ja, desca, work)
val = pzlantr(norm, uplo, diag, m, n, a, ia, ja, desca, work)
```

Description

The routine returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular distributed matrix *sub*(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1).

Input Parameters

<i>norm</i>	(global) CHARACTER. Specifies what value is returned by the routine: = 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A, it is not a matrix norm. = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum), = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum), = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<i>uplo</i>	(global) CHARACTER. Specifies whether the upper or lower triangular part of the symmetric matrix $\text{sub}(A)$ is to be referenced. = 'U': Upper trapezoidal, = 'L': Lower trapezoidal. Note that $\text{sub}(A)$ is triangular instead of trapezoidal if $m = n$.
<i>diag</i>	(global). CHARACTER. Specifies whether the distributed matrix $\text{sub}(A)$ has unit diagonal. = 'N': Non-unit diagonal. = 'U': Unit diagonal.
<i>m</i>	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$. When $m = 0$, <code>p?lantr</code> is set to zero. $m \geq 0$.
<i>n</i>	(global) INTEGER. The number of columns to be operated on i.e the number of columns of the distributed submatrix $\text{sub}(A)$. When $n = 0$, <code>p?lantr</code> is set to zero. $n \geq 0$.
<i>a</i>	(local). Real for <code>pslantr</code> DOUBLE PRECISION for <code>pdlantr</code> COMPLEX for <code>pclantr</code> COMPLEX*16 for <code>pzlantr</code>.

Pointer into the local memory to an array of
`DIMENSION(11d_a, LOCc(ja+n-1))` containing the local
pieces of the distributed matrix `sub(A)`.

`ia, ja` (global) INTEGER.
The row and column indices in the global array `a` indicating
the first row and the first column of the submatrix `sub(A)`,
respectively.

`desca` (global and local) INTEGER array, `DIMENSION(dlen_)`. The
array descriptor for the distributed matrix `A`.

`work` (local).
Real for `pslantr`
DOUBLE PRECISION for `pdlantr`
COMPLEX for `pclantr`
COMPLEX*16 for `pzlantr`.
Array `DIMENSION(lwork)`.
`lwork ≥ 0` if `norm = 'M' or 'm'` (not referenced),
`nq0` if `norm = '1', 'O' or 'o'`,
`mp0` if `norm = 'I' or 'i'`,
`0` if `norm = 'F', 'f', 'E' or 'e'` (not referenced),
where `lcm` is the least common multiple of `nprow` and `npcol`
`lcm = ilcm(nprow, npcol)` and `ceil` denotes the ceiling
operation (`iceil`).
`iroffa = mod(ia-1, mb_a), icoffa = mod(ja-1,`
`nb_a),`
`iarow = indxg2p(ia, mb_a, myrow, rsrc_a, nprow),`
`iacol = indxg2p(ja, nb_a, mycol, csrc_a, npcol),`
`mp0 = numroc(m+iroffa, mb_a, myrow, iarow,`
`nprow),`
`nq0 = numroc(n+icoffa, nb_a, mycol, iacol,`
`npcol),`
`indxg2p` and `numroc` are ScaLAPACK tool functions; `myrow,`
`mycol, nprow,` and `npcol` can be determined by calling the
subroutine `blacs_gridinfo`.

Output Parameters

`val` The value returned by the routine.

p?lapiv

Applies a permutation matrix to a general distributed matrix, resulting in row or column pivoting.

Syntax

```
call pslapiv(direc, rowcol, pivroc, m, n, a, ia, ja, desca, ipiv, ip, jp,
descip, iwork)

call pdlapiv(direc, rowcol, pivroc, m, n, a, ia, ja, desca, ipiv, ip, jp,
descip, iwork)

call pclapiv(direc, rowcol, pivroc, m, n, a, ia, ja, desca, ipiv, ip, jp,
descip, iwork)

call pzlapiv(direc, rowcol, pivroc, m, n, a, ia, ja, desca, ipiv, ip, jp,
descip, iwork)
```

Description

This routine applies either P (permutation matrix indicated by *ipiv*) or $\text{inv}(P)$ to a general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$, resulting in row or column pivoting. The pivot vector may be distributed across a process row or a column. The pivot vector should be aligned with the distributed matrix A . This routine will transpose the pivot vector, if necessary.

For example, if the row pivots should be applied to the columns of $\text{sub}(A)$, pass *rowcol*='C' and *pivroc*='C'.

Input Parameters

<i>direc</i>	(global) CHARACTER*1. Specifies in which order the permutation is applied: = 'F' (Forward). Applies pivots forward from top of matrix. Computes $P * \text{sub}(A)$. = 'B' (Backward). Applies pivots backward from bottom of matrix. Computes $\text{inv}(P) * \text{sub}(A)$.
<i>rowcol</i>	(global) CHARACTER*1. Specifies if the rows or columns are to be permuted: = 'R' Rows will be permuted, = 'C' Columns will be permuted.

<i>pivroc</i>	(global) CHARACTER*1. Specifies whether <i>ipiv</i> is distributed over a process row or column: = 'R' <i>ipiv</i> is distributed over a process row, = 'C' <i>ipiv</i> is distributed over a process column.
<i>m</i>	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(<i>A</i>). When $m = 0$, <i>p?lapiv</i> is set to zero. $m \geq 0$.
<i>n</i>	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(<i>A</i>). When $n = 0$, <i>p?lapiv</i> is set to zero. $n \geq 0$.
<i>a</i>	(local). Real for <i>pslapiv</i> DOUBLE PRECISION for <i>pdlapiv</i> COMPLEX for <i>pclapiv</i> COMPLEX*16 for <i>pzlapiv</i>. Pointer into the local memory to an array of DIMENSION(<i>lld_a</i>, <i>LOCc(ja+n-1)</i>) containing the local pieces of the distributed matrix sub(<i>A</i>).
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>A</i> indicating the first row and the first column of the submatrix sub(<i>A</i>), respectively.
<i>desca</i>	(global and local) INTEGER array, DIMENSION (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i>.
<i>ipiv</i>	(local). INTEGER. Array, DIMENSION (<i>lipiv</i>) ; when <i>rowcol</i>='R' or 'r': <i>lipiv</i> ≥ <i>LOCr(ia+m-1) + mb_a</i> if <i>pivroc</i>='C' or 'c', <i>lipiv</i> ≥ <i>LOCc(m + mod(jp-1, nb_p))</i> if <i>pivroc</i>='R' or 'r', and, when <i>rowcol</i>='C' or 'c':

$lipiv \geq LOCr(n + \text{mod}(ip-1, mb\_p))$ if $pivroc='C'$ or $'c'$,

$lipiv \geq LOCc(ja+n-1) + nb\_a$ if $pivroc='R'$ or $'r'$.

This array contains the pivoting information. $ipiv(i)$ is the global row (column), local row (column) i was swapped with. When $rowcol='R'$ or $'r'$ and $pivroc='C'$ or $'c'$, or $rowcol='C'$ or $'c'$ and $pivroc='R'$ or $'r'$, the last piece of this array of size mb_a (resp. nb_a) is used as workspace. In those cases, this array is tied to the distributed matrix A .

ip, jp

(global) INTEGER. The row and column indices in the global array P indicating the first row and the first column of the submatrix $\text{sub}(P)$, respectively.

$descip$

(global and local) INTEGER array, DIMENSION ($dlen\_$). The array descriptor for the distributed vector $ipiv$.

$iwork$

(local). INTEGER.

Array, DIMENSION (ldw), where ldw is equal to the workspace necessary for transposition, and the storage of the transposed $ipiv$:

Let lcm be the least common multiple of $nprow$ and $npcol$.

```
If( rowcol.eq.'r' .and. pivroc.eq.'r') then
    If( nprow.eq. npcol) then
        ldw = LOCr( n_p + mod(jp-1, nb_p) ) + nb_p
    else
        ldw = LOCr( n_p + mod(jp-1, nb_p) ) +
        nb_p * ceil( ceil(LOCc(n_p)/nb_p) / (lcm/npcol) )
    end if
else if( rowcol.eq.'c' .and. pivroc.eq.'c') then
    if( nprow.eq.
npcol ) then
        ldw = LOCc( m_p + mod(ip-1, mb_p) ) + mb_p
    else
        ldw = LOCc( m_p + mod(ip-1, mb_p) ) +
        mb_p * ceil(ceil(LOCr(m_p)/mb_p) / (lcm/nprow) )
    end if
else
iwork is not referenced.
end if.
```

Output Parameters

a

(local).

On exit, the local pieces of the permuted distributed submatrix.

p?laqge

Scales a general rectangular matrix, using row and column scaling factors computed by p?geequ.

Syntax

```
call pslaqge(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, equed)
call pdlaqge(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, equed)
call pclaqge(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, equed)
call pzlaqge(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, equed)
```

Description

This routine equilibrates a general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ using the row and scaling factors in the vectors r and c computed by p?geequ.

Input Parameters

m	(global). INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$. ($m \geq 0$).
n	(global). INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$. ($n \geq 0$).
a	(local). REAL for pslaqge DOUBLE PRECISION for pdlaqge COMPLEX for pclaqge COMPLEX*16 for pzlaqge. Pointer into the local memory to an array of DIMENSION(lld_a , $LOCc(ja+n-1)$). On entry, this array contains the distributed matrix $\text{sub}(A)$.
ia, ja	(global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.

<i>desca</i>	(global and local) INTEGER array, DIMENSION (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>r</i>	(local). REAL for pslaqge DOUBLE PRECISION for pdlaqge COMPLEX for pclaqge COMPLEX*16 for pzlaqge. Array, DIMENSION <i>LOCr(m_a)</i>. The row scale factors for sub(<i>A</i>). <i>r</i> is aligned with the distributed matrix <i>A</i>, and replicated across every process column. <i>r</i> is tied to the distributed matrix <i>A</i>.
<i>c</i>	(local). REAL for pslaqge DOUBLE PRECISION for pdlaqge COMPLEX for pclaqge COMPLEX*16 for pzlaqge. Array, DIMENSION <i>LOCc(n_a)</i>. The row scale factors for sub(<i>A</i>). <i>c</i> is aligned with the distributed matrix <i>A</i>, and replicated across every process column. <i>c</i> is tied to the distributed matrix <i>A</i>.
<i>rowcnd</i>	(local). REAL for pslaqge DOUBLE PRECISION for pdlaqge COMPLEX for pclaqge COMPLEX*16 for pzlaqge. The global ratio of the smallest $r(i)$ to the largest $r(i)$, $ia \leq i \leq ia+m-1$.
<i>colcnd</i>	(local). REAL for pslaqge DOUBLE PRECISION for pdlaqge COMPLEX for pclaqge COMPLEX*16 for pzlaqge. The global ratio of the smallest $c(i)$ to the largest $c(i)$, $ia \leq i \leq ia+n-1$.
<i>amax</i>	(global). REAL for pslaqge DOUBLE PRECISION for pdlaqge

COMPLEX for pclaqqe
 COMPLEX*16 for pzlaqqe.
 Absolute value of largest distributed submatrix entry.

Output Parameters

a (local).
 On exit, the equilibrated distributed matrix. See *equed* for the form of the equilibrated distributed submatrix.

equed (global) CHARACTER.
 Specifies the form of equilibration that was done.
 = 'N': No equilibration
 = 'R': Row equilibration, that is, $\text{sub}(A)$ has been pre-multiplied by $\text{diag}(r(ia:ia+m-1))$,
 = 'C': column equilibration, that is, $\text{sub}(A)$ has been post-multiplied by $\text{diag}(c(ja:ja+n-1))$,
 = 'B': Both row and column equilibration, that is, $\text{sub}(A)$ has been replaced by $\text{diag}(r(ia:ia+m-1)) * \text{sub}(A) * \text{diag}(c(ja:ja+n-1))$.

p?laqsy

Scales a symmetric/Hermitian matrix, using scaling factors computed by p?poequ.

Syntax

```
call pslaqsy(uplo, n, a, ia, ja, desca, sr, sc, scond, amax, equed)
call pdlaqsy(uplo, n, a, ia, ja, desca, sr, sc, scond, amax, equed)
call pclaqsy(uplo, n, a, ia, ja, desca, sr, sc, scond, amax, equed)
call pzlaqsy(uplo, n, a, ia, ja, desca, sr, sc, scond, amax, equed)
```

Description

This routine equilibrates a symmetric distributed matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ using the scaling factors in the vectors *sr* and *sc*. The scaling factors are computed by p?poequ.

Input Parameters

<i>uplo</i>	(global) CHARACTER. Specifies the upper or lower triangular part of the symmetric distributed matrix $\text{sub}(A)$ is to be referenced: = 'U': Upper triangular part; = 'L': Lower triangular part.
<i>n</i>	(global) INTEGER. The order of the distributed submatrix $\text{sub}(A)$. $n \geq 0$.
<i>a</i>	(local). REAL for pslaqsy DOUBLE PRECISION for pdlaqsy COMPLEX for pclaqsy COMPLEX*16 for pzlaqsy. Pointer into the local memory to an array of DIMENSION (<i>lld_a</i>, <i>LOCc(ja+n-1)</i>). On entry, this array contains the local pieces of the distributed matrix $\text{sub}(A)$. On entry, the local pieces of the distributed symmetric matrix $\text{sub}(A)$. If <i>uplo</i> = 'U', the leading <i>n</i>-by-<i>n</i> upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and the strictly lower triangular part of $\text{sub}(A)$ is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i>-by-<i>n</i> lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the matrix, and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>A</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array, DIMENSION (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i>.
<i>sr</i>	(local) REAL for pslaqsy DOUBLE PRECISION for pdlaqsy COMPLEX for pclaqsy COMPLEX*16 for pzlaqsy.

	Array, DIMENSION $LOCr(m\_a)$. The scale factors for $A(ia:ia+m-1, ja:ja+n-1)$. sr is aligned with the distributed matrix A, and replicated across every process column. sr is tied to the distributed matrix A.
<i>sc</i>	(local) REAL for pslaqsy DOUBLE PRECISION for pdlaqsy COMPLEX for pclaqsy COMPLEX*16 for pzlaqsy. Array, DIMENSION $LOCc(m\_a)$. The scale factors for $A(ia:ia+m-1, ja:ja+n-1)$. sr is aligned with the distributed matrix A, and replicated across every process column. sr is tied to the distributed matrix A.
<i>scond</i>	(global). REAL for pslaqsy DOUBLE PRECISION for pdlaqsy COMPLEX for pclaqsy COMPLEX*16 for pzlaqsy. Ratio of the smallest $sr(i)$ (respectively $sc(j)$) to the largest $sr(i)$ (respectively $sc(j)$), with $ia \leq i \leq ia+n-1$ and $ja \leq j \leq ja+n-1$.
<i>amax</i>	(global). REAL for pslaqsy DOUBLE PRECISION for pdlaqsy COMPLEX for pclaqsy COMPLEX*16 for pzlaqsy. Absolute value of largest distributed submatrix entry.

Output Parameters

<i>a</i>	On exit, if <i>equed</i> = 'Y', the equilibrated matrix: $\text{diag}(sr(ia:ia+n-1)) * \text{sub}(A) * \text{diag}(sc(ja:ja+n-1))$.
<i>equed</i>	(global) CHARACTER*1. Specifies whether or not equilibration was done. = 'N': No equilibration.

```
= 'Y': Equilibration was done, that is, sub(A) has been
replaced by:
diag(sr(ia:ia+n-1))* sub(A) *
diag(sc(ja:ja+n-1)).
```

p?lared1d

Redistributes an array assuming that the input array, `bycol`, is distributed across rows and that all process columns contain the same copy of `bycol`.

Syntax

```
call pslared1d(n, ia, ja, desc, bycol, byall, work, lwork)
call pdlared1d(n, ia, ja, desc, bycol, byall, work, lwork)
```

Description

This routine redistributes a 1D array. It assumes that the input array `bycol` is distributed across rows and that all process column contain the same copy of `bycol`. The output array `byall` is identical on all processes and contains the entire array.

Input Parameters

`np` = Number of local rows in `bycol()`

`n` (global). INTEGER.
The number of elements to be redistributed. $n \geq 0$.

`ia, ja` (global) INTEGER. `ia, ja` must be equal to 1.

`desc` (global and local) INTEGER array, DIMENSION 8. A 2D array descriptor, which describes `bycol`.

`bycol` (local).
REAL for `pslared1d`
DOUBLE PRECISION for `pdlared1d`
COMPLEX for `pclared1d`
COMPLEX*16 for `pzlared1d`.
Distributed block cyclic array global DIMENSION (n), local DIMENSION np . `bycol` is distributed across the process rows. All process columns are assumed to contain the same value.

work (local).
 REAL for pslared1d
 DOUBLE PRECISION for pdlared1d
 COMPLEX for pclared1d
 COMPLEX*16 for pzlarred1d.
 DIMENSION (*lwork*). Used to hold the buffers sent from one process to another.

lwork (local)
 INTEGER. The size of the *work* array. $lwork \geq \text{numroc}(n, \text{desc}(\text{nb\_}), 0, 0, \text{npcol})$.

Output Parameters

byall (global). REAL for pslared1d
 DOUBLE PRECISION for pdlared1d
 COMPLEX for pclared1d
 COMPLEX*16 for pzlarred1d.
 Global DIMENSION (*n*), local DIMENSION (*n*). *byall* is exactly duplicated on all processes. It contains the same values as *bycol*, but it is replicated across all processes rather than being distributed.

p?lared2d

*Redistributes an array assuming that the input array *byrow* is distributed across columns and that all process rows contain the same copy of *byrow*.*

Syntax

```
call pslared2d(n, ia, ja, desc, byrow, byall, work, lwork)
call pdlared2d(n, ia, ja, desc, byrow, byall, work, lwork)
```

Description

This routine redistributes a 1D array. It assumes that the input array *byrow* is distributed across columns and that all process rows contain the same copy of *byrow*. The output array *byall* will be identical on all processes and will contain the entire array.

Input Parameters

np = Number of local rows in *byrow*()

<i>n</i>	(global) INTEGER. The number of elements to be redistributed. $n \geq 0$.
<i>ia, ja</i>	(global) INTEGER. <i>ia, ja</i> must be equal to 1.
<i>desc</i>	(global and local) INTEGER array, DIMENSION (<i>dlen_</i>). A 2D array descriptor, which describes <i>byrow</i> .
<i>byrow</i>	(local). REAL for <i>pslared2d</i> DOUBLE PRECISION for <i>pdlared2d</i> COMPLEX for <i>pclared2d</i> COMPLEX*16 for <i>pzlared2d</i> . Distributed block cyclic array global DIMENSION (<i>n</i>), local DIMENSION <i>np</i> . <i>bycol</i> is distributed across the process columns. All process rows are assumed to contain the same value.
<i>work</i>	(local). REAL for <i>pslared2d</i> DOUBLE PRECISION for <i>pdlared2d</i> COMPLEX for <i>pclared2d</i> COMPLEX*16 for <i>pzlared2d</i> . DIMENSION (<i>lwork</i>). Used to hold the buffers sent from one process to another.
<i>lwork</i>	(local).INTEGER. The size of the <i>work</i> array. $lwork \geq \text{numroc}(n, \text{desc}(nb\_), 0, 0, npcol)$.

Output Parameters

<i>byall</i>	(global). REAL for <i>pslared2d</i> DOUBLE PRECISION for <i>pdlared2d</i> COMPLEX for <i>pclared2d</i> COMPLEX*16 for <i>pzlared2d</i> . Global DIMENSION(<i>n</i>), local DIMENSION (<i>n</i>). <i>byall</i> is exactly duplicated on all processes. It contains the same values as <i>bycol</i> , but it is replicated across all processes rather than being distributed.

p?larf

Applies an elementary reflector to a general rectangular matrix.

Syntax

```
call pslarf(side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pdlarf(side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pclarf(side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pzlarf(side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
```

Description

This routine applies a real/complex elementary reflector Q (or Q^T) to a real/complex m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$, from either the left or the right. Q is represented in the form

$$Q = I - \tau v v',$$

where τ is a real/complex scalar and v is a real/complex vector.

If $\tau = 0$, then Q is taken to be the unit matrix.

Input Parameters

<i>side</i>	(global). CHARACTER. = 'L': form $Q * \text{sub}(C)$, = 'R': form $\text{sub}(C) * Q$, $Q = Q^T$.
<i>m</i>	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$. ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$. ($n \geq 0$).
<i>v</i>	(local). REAL for pslarf DOUBLE PRECISION for pdlarf COMPLEX for pclarf

	COMPLEX*16 for pzlarf. Pointer into the local memory to an array of DIMENSION (lld_v,*) containing the local pieces of the distributed vectors <i>v</i> representing the Householder transformation <i>Q</i>, <i>v</i>(iv:iv+m-1, jv) if <i>side</i> = 'L' and <i>incv</i> = 1, <i>v</i>(iv, jv:jv+m-1) if <i>side</i> = 'L' and <i>incv</i> = m_v, <i>v</i>(iv:iv+n-1, jv) if <i>side</i> = 'R' and <i>incv</i> = 1, <i>v</i>(iv, jv:jv+n-1) if <i>side</i> = 'R' and <i>incv</i> = m_v. The vector <i>v</i> is the representation of <i>Q</i>. <i>v</i> is not used if <i>tau</i> = 0.
<i>iv, jv</i>	(global) INTEGER. The row and column indices in the global array <i>v</i> indicating the first row and the first column of the submatrix sub(<i>v</i>), respectively.
<i>descv</i>	(global and local) INTEGER array, DIMENSION (dlen_). The array descriptor for the distributed matrix <i>V</i> .
<i>incv</i>	(global) INTEGER. The global increment for the elements of <i>v</i> . Only two values of <i>incv</i> are supported in this version, namely 1 and m_v. <i>incv</i> must not be zero.
<i>tau</i>	(local). REAL for pslarf DOUBLE PRECISION for pdlarf COMPLEX for pclarf COMPLEX*16 for pzlarf. Array, DIMENSION LOCc(jv) if <i>incv</i> = 1, and LOCc(iv) otherwise. This array contains the Householder scalars related to the Householder vectors. <i>tau</i> is tied to the distributed matrix <i>v</i> .
<i>c</i>	(local). REAL for pslarf DOUBLE PRECISION for pdlarf COMPLEX for pclarf COMPLEX*16 for pzlarf. Pointer into the local memory to an array of DIMENSION (lld_c, LOCc(jc+n-1)), containing the local pieces of sub(<i>c</i>).
<i>ic, jc</i>	(global) INTEGER.

The row and column indices in the global array *c* indicating the first row and the first column of the submatrix sub(*c*), respectively.

desc (global and local) INTEGER array, DIMENSION (*dlen*₁). The array descriptor for the distributed matrix *C*.

work (local).

```

REAL for pslarf
DOUBLE PRECISION for pdlarf
COMPLEX for pclarf
COMPLEX*16 for pzlarf.
Array, DIMENSION (lwork).
If incv = 1,
    if side = 'L',
        if ivcol = iccol,
            lwork ≥ nqc0
        else
            lwork ≥ mpc0 + max( 1, nqc0 )
        end if
    else if side = 'R' ,
        lwork ≥ nqc0 + max( max( 1, mpc0), numroc(numroc( n+
            icoffc,nb_v,0,0,npcol),nb_v,0,0,lcmq ) )
    end if
else if incv = m_v,
    if side = 'L',
        lwork ≥ mpc0 + max( max( 1, nqc0 ), numroc(
            numroc(m+iroffc,mb_v,0,0,nprow),mb_v,0,0, lcmp )
        )
    else if side = 'R',
        if ivrow = icrow,
            lwork ≥ mpc0
        else
            lwork ≥ nqc0 + max( 1, mpc0 )
        end if
    end if
end if,

```

where lcm is the least common multiple of $nprow$ and $npcol$ and $lcm = ilcm(nprow, npcol)$, $lcmp = lcm/nprow$, $lcmq = lcm/npcol$,
 $iroffc = mod(ic-1, mb\_c)$, $icoffc = mod(jc-1, nb\_c)$,
 $icrow = indxg2p(ic, mb\_c, myrow, rsrc\_c, nprow)$,
 $iccol = indxg2p(jc, nb\_c, mycol, csrc\_c, npcol)$,
 $mpc0 = numroc(m+iroffc, mb\_c, myrow, icrow, nprow)$,
 $nqc0 = numroc(n+icoffc, nb\_c, mycol, iccol, npcol)$,
 $ilcm$, $indxg2p$, and $numroc$ are ScaLAPACK tool functions;
 $myrow$, $mycol$, $nprow$, and $npcol$ can be determined by calling the subroutine `blacs_gridinfo`.

Output Parameters

c

(local).

On exit, $sub(c)$ is overwritten by the $Q * sub(C)$ if $side = 'L'$,
or $sub(c) * Q$ if $side = 'R'$.

p?larfb

Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.

Syntax

```
call pslarfb(side, trans, direct, storev, m, n, k, v, iv, jv, descv, t, c,
            ic, jc, descc, work)

call pdlarfb(side, trans, direct, storev, m, n, k, v, iv, jv, descv, t, c,
            ic, jc, descc, work)

call pclarfb(side, trans, direct, storev, m, n, k, v, iv, jv, descv, t, c,
            ic, jc, descc, work)

call pzlarfb(side, trans, direct, storev, m, n, k, v, iv, jv, descv, t, c,
            ic, jc, descc, work)
```

Description

This routine applies a real/complex block reflector Q or its transpose Q^T /conjugate transpose Q^H to a real/complex distributed m -by- n matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ from the left or the right.

Input Parameters

<i>side</i>	(global).CHARACTER. if <i>side</i> = 'L': apply Q or Q^T for real flavors (Q^H for complex flavors) from the Left; if <i>side</i> = 'R': apply Q or Q^T for real flavors (Q^H for complex flavors) from the Right.
<i>trans</i>	(global).CHARACTER. if <i>trans</i> = 'N': no transpose, apply Q ; for real flavors, if <i>trans</i> ='T': transpose, apply Q^T for complex flavors, if <i>trans</i> = 'C': conjugate transpose, apply Q^H ;
<i>direct</i>	(global) CHARACTER. Indicates how Q is formed from a product of elementary reflectors. if <i>direct</i> = 'F': $Q = H(1)*H(2)*\dots*H(k)$ (Forward) if <i>direct</i> = 'B': $Q = H(k)*\dots*H(2)*H(1)$ (Backward)

<i>storev</i>	(global) CHARACTER. Indicates how the vectors that define the elementary reflectors are stored: if <i>storev</i> = 'C': Columnwise if <i>storev</i> = 'R': Rowwise.
<i>m</i>	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(C)$. ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(C)$. ($n \geq 0$).
<i>k</i>	(global) INTEGER. The order of the matrix T.
<i>v</i>	(local). REAL for pslarfb DOUBLE PRECISION for pdlarfb COMPLEX for pclarfb COMPLEX*16 for pzlarfb. Pointer into the local memory to an array of DIMENSION (<i>lld_v</i>, <i>LOCc</i>(<i>lv</i>+<i>k</i>-1)) if <i>storev</i> = 'C', (<i>lld_v</i>, <i>LOCc</i>(<i>lv</i>+<i>m</i>-1)) if <i>storev</i> = 'R' and <i>side</i> = 'L', (<i>lld_v</i>, <i>LOCc</i>(<i>lv</i>+<i>n</i>-1)) if <i>storev</i> = 'R' and <i>side</i> = 'R'. It contains the local pieces of the distributed vectors <i>v</i> representing the Householder transformation. if <i>storev</i> = 'C' and <i>side</i> = 'L', <i>lld_v</i> $\geq$ <i>max</i>(1,<i>LOCr</i>(<i>iv</i>+<i>m</i>-1)); if <i>storev</i> = 'C' and <i>side</i> = 'R', <i>lld_v</i> $\geq$ <i>max</i>(1,<i>LOCr</i>(<i>iv</i>+<i>n</i>-1)); if <i>storev</i> = 'R', <i>lld_v</i> $\geq$ <i>LOCr</i>(<i>lv</i>+<i>k</i>-1).
<i>iv, lv</i>	(global) INTEGER. The row and column indices in the global array <i>v</i> indicating the first row and the first column of the submatrix $\text{sub}(v)$, respectively.

descv (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *v*.

c (local).
 REAL for pslarfb
 DOUBLE PRECISION for pdlarfb
 COMPLEX for pclarfb
 COMPLEX*16 for pzlarfb.
 Pointer into the local memory to an array of
 DIMENSION(*lld_c*, *LOCc(jc+n-1)*), containing the local pieces of sub(*c*).

ic, jc (global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix sub(*c*), respectively.

descc (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *c*.

work (local).
 REAL for pslarfb
 DOUBLE PRECISION for pdlarfb
 COMPLEX for pclarfb
 COMPLEX*16 for pzlarfb.
 Workspace array, DIMENSION (*lwork*).
 If *storev* = 'C',
 if *side* = 'L',
 $lwork \geq (nqc0 + mpc0) * k$
 else if *side* = 'R',
 $lwork \geq (nqc0 + \max(npv0 + \text{numroc}(\text{numroc}(n + icoffc, nb\_v, 0, 0, npc0), nb\_v, 0, 0, lcmq), mpc0)) * k$
 end if
 else if *storev* = 'R',
 if *side* = 'L',
 $lwork \geq (mpc0 + \max(mqv0 + \text{numroc}(\text{numroc}(m + iroffc, mb\_v, 0, 0, nprow), mb\_v, 0, 0, lcmp), nqc0)) * k$
 else if *side* = 'R',
 $lwork \geq (mpc0 + nqc0) * k$

```

end if
end if,
where
lcmq = lcm / npcol with lcm = iclm( nprow, npcol ),
iroffv = mod( iv-1, mb_v ), icoffv = mod( jv-1, nb_v ),
ivrow = indxg2p( iv, mb_v, myrow, rsrc_v, nprow ),
ivcol = indxg2p( jv, nb_v, mycol, csrc_v, npcol ),
MqV0 = numroc( m+icoffv, nb_v, mycol, ivcol, npcol ),
NpV0 = numroc( n+iroffv, mb_v, myrow, ivrow, nprow ),
iroffc = mod( ic-1, mb_c ), icoffc = mod( jc-1, nb_c ),
icrow = indxg2p( ic, mb_c, myrow, rsrc_c, nprow ),
iccol = indxg2p( jc, nb_c, mycol, csrc_c, npcol ),
MpC0 = numroc( m+iroffc, mb_c, myrow, icrow, nprow ),
NpC0 = numroc( n+icoffc, mb_c, myrow, icrow, nprow ),
NqC0 = numroc( n+icoffc, nb_c, mycol, iccol, npcol ),
ilcm, indxg2p, and numroc are ScaLAPACK tool functions;
myrow, mycol, nprow, and npcol can be determined by
calling the subroutine blacs_gridinfo.

```

Output Parameters

<i>t</i>	(local). REAL for pslarfb DOUBLE PRECISION for pdlarfb COMPLEX for pclarfb COMPLEX*16 for pzlarfb. Array, DIMENSION(<i>mb_v</i>, <i>mb_v</i>) if <i>storev</i> = 'R', and (<i>nb_v</i>, <i>nb_v</i>) if <i>storev</i> = 'C'. The triangular matrix <i>t</i> is the representation of the block reflector.
<i>c</i>	(local). On exit, sub(<i>c</i>) is overwritten by the $Q \cdot \text{sub}(C)$, or $Q' \cdot \text{sub}(C)$, or $\text{sub}(C) \cdot Q$, or $\text{sub}(C) \cdot Q'$. Q' is transpose (conjugate transpose) of Q.

p?larfc

Applies the conjugate transpose of an elementary reflector to a general matrix.

Syntax

```
call pclarfc(side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pzlarfc(side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
```

Description

This routine applies a complex elementary reflector Q^H to a complex m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$, from either the left or the right. Q is represented in the form

$$Q = I - \tau v v^H,$$

where τ is a complex scalar and v is a complex vector.

If $\tau = 0$, then Q is taken to be the unit matrix.

Input Parameters

<i>side</i>	(global) CHARACTER. if <i>side</i> = 'L': form $Q^H * \text{sub}(C)$; if <i>side</i> = 'R': form $\text{sub}(C) * Q^H$.
<i>m</i>	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(C)$. ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(C)$. ($n \geq 0$).
<i>v</i>	(local). COMPLEX for pclarfc COMPLEX*16 for pzlarfc. Pointer into the local memory to an array of DIMENSION (<i>lld_v</i> ,*) containing the local pieces of the distributed vectors v representing the Householder transformation Q ,

$v(iv:iv+m-1, jv)$ if $side = 'L'$ and $incv = 1$,
 $v(iv, jv:jv+m-1)$ if $side = 'L'$ and $incv = m\_v$,
 $v(iv:iv+n-1, jv)$ if $side = 'R'$ and $incv = 1$,
 $v(iv, jv:jv+n-1)$ if $side = 'R'$ and $incv = m\_v$.
The vector v is the representation of Q . v is not used if $tau = 0$.

iv, jv	(global) INTEGER. The row and column indices in the global array v indicating the first row and the first column of the submatrix $sub(v)$, respectively.
$descv$	(global and local) INTEGER array, DIMENSION ($dlen\_$). The array descriptor for the distributed matrix v .
$incv$	(global) INTEGER. The global increment for the elements of v . Only two values of $incv$ are supported in this version, namely 1 and m_v . $incv$ must not be zero.
tau	(local) COMPLEX for pclarfc COMPLEX*16 for pzlarfc. Array, DIMENSION $LOCc(jv)$ if $incv = 1$, and $LOCr(iv)$ otherwise. This array contains the Householder scalars related to the Householder vectors. tau is tied to the distributed matrix v .
c	(local). COMPLEX for pclarfc COMPLEX*16 for pzlarfc. Pointer into the local memory to an array of DIMENSION ($lld\_c, LOCc(jc+n-1)$), containing the local pieces of $sub(c)$.
ic, jc	(global) INTEGER. The row and column indices in the global array c indicating the first row and the first column of the submatrix $sub(c)$, respectively.
$desc$	(global and local) INTEGER array, DIMENSION ($dlen\_$). The array descriptor for the distributed matrix c .
$work$	(local).

```

COMPLEX for polarfc
COMPLEX*16 for pzlarfc.
Workspace array, DIMENSION (lwork).
If incv = 1,
  if side = 'L' ,
    if ivcol = iccol,
      lwork ≥ nqc0
    else
      lwork ≥ mpc0 + max( 1, nqc0 )
    end if
  else if side = 'R' ,
    lwork ≥ nqc0 + max( max( 1, mpc0 ), numroc( numroc(
      n+icoffc,nb_v,0,0,npcol ), nb_v,0,0,lcmq ) )
    end if
  else if incv = m_v,
    if side = 'L' ,
      lwork ≥ mpc0 + max( max( 1, nqc0 ), numroc( numroc(
        m+iroffc,mb_v,0,0,nprow ),mb_v,0,0,lcmp ) )
    else if side = 'R' ,
      if ivrow = icrow,
        lwork ≥ mpc0
      else
        lwork ≥ nqc0 + max( 1, mpc0 )
      end if
    end if
  end if,
where lcm is the least common multiple of nprow and npcol
and lcm = ilcm(nprow, npcol),
lcmp = lcm/nprow, lcmq = lcm/npcol,
iroffc = mod(ic-1, mb_c), icoffc = mod(jc-1,
nb_c),
icrow = indxg2p(ic, mb_c, myrow, rsrc_c, nprow),
iccol = indxg2p(jc, nb_c, mycol, csrc_c, npcol),
mpc0 = numroc(m+iroffc, mb_c, myrow, icrow,
nprow),
nqc0 = numroc(n+icoffc, nb_c, mycol, iccol,
npcol),

```

`ilcm`, `indxg2p`, and `numroc` are ScaLAPACK tool functions; `myrow`, `mycol`, `nrow`, and `ncol` can be determined by calling the subroutine `blacs_gridinfo`.

Output Parameters

`c` (local).
On exit, `sub(c)` is overwritten by the $Q^H \cdot \text{sub}(C)$ if `side = 'L'`, or `sub(C) * Q^H` if `side = 'R'`.

p?larfg

Generates an elementary reflector (Householder matrix).

Syntax

```
call pslarfg(n, alpha, iax, jax, x, ix, jx, descx, incx, tau)
call pdlarfg(n, alpha, iax, jax, x, ix, jx, descx, incx, tau)
call pclarfg(n, alpha, iax, jax, x, ix, jx, descx, incx, tau)
call pzlarfg(n, alpha, iax, jax, x, ix, jx, descx, incx, tau)
```

Description

This routine generates a real/complex elementary reflector H of order n , such that

$$H^* \text{sub}(X) = H^* (x(iax, jax)) = (\alpha), \quad H'^* H = I,$$

$$\begin{pmatrix} x \\ 0 \end{pmatrix}$$

where α is a scalar (a real scalar - for complex flavors), and `sub(X)` is an $(n-1)$ -element real/complex distributed vector $X(ix:ix+n-2, jx)$ if `incx = 1` and $X(ix, jx:jx+n-2)$ if `incx = descx(m_)`. H is represented in the form

$$H = I - \tau \begin{pmatrix} 1 \\ v \end{pmatrix} \begin{pmatrix} 1 & v' \end{pmatrix},$$

where τ is a real/complex scalar and v is a real/complex $(n-1)$ -element vector. Note that H is not Hermitian.

If the elements of `sub(x)` are all zero (and $X(iax, jax)$ is real for complex flavors), then $\tau = 0$ and H is taken to be the unit matrix.

Otherwise $1 \leq \text{real}(\tau) \leq 2$ and $\text{abs}(\tau-1) \leq 1$.

Input Parameters

n (global) INTEGER.
The global order of the elementary reflector. $n \geq 0$.

iax, jax (global) INTEGER.
The global row and column indices in *x* of $X(iax, jax)$.

x (local).
Real for pslarfg
DOUBLE PRECISION for pdlarfg
COMPLEX for pclarfg
COMPLEX*16 for pzlarfg.
Pointer into the local memory to an array of DIMENSION (*lld_x*, *). This array contains the local pieces of the distributed vector sub(*x*). Before entry, the incremented array sub(*x*) must contain vector *x*.

ix, jx (global) INTEGER.
The row and column indices in the global array *x* indicating the first row and the first column of sub(*x*), respectively.

descx (global and local) INTEGER.
Array of DIMENSION (*dlen_*). The array descriptor for the distributed matrix *x*.

incx (global) INTEGER.
The global increment for the elements of *x*. Only two values of *incx* are supported in this version, namely 1 and *m_x*. *incx* must not be zero.

Output Parameters

alpha (local)
REAL for pslafg
DOUBLE PRECISION for pdlafg
COMPLEX for pclafg
COMPLEX*16 for pzlafg.
On exit, *alpha* is computed in the process scope having the vector sub(*x*).

x (local).
On exit, it is overwritten with the vector *v*.

tau (local).
Real for `pslarfg`
DOUBLE PRECISION for `pdlarfg`
COMPLEX for `pclarfg`
COMPLEX*16 for `pzlarfg`.
Array, DIMENSION *LOCc(jx)* if *incx* = 1, and *LOCr(ix)* otherwise. This array contains the Householder scalars related to the Householder vectors.
tau is tied to the distributed matrix *X*.

p?larft

Forms the triangular vector *T* of a block reflector

$$H = I - V * T * V^H.$$

Syntax

```
call pslarft(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
call pdlarft(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
call pclarft(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
call pzlarft(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
```

Description

This routine forms the triangular factor *T* of a real/complex block reflector *H* of order *n*, which is defined as a product of *k* elementary reflectors.

If *direct* = 'F', $H = H(1) * H(2) \dots * H(k)$, and *T* is upper triangular;

If *direct* = 'B', $H = H(k) * \dots * H(2) * H(1)$, and *T* is lower triangular.

If *storev* = 'C', the vector which defines the elementary reflector $H(i)$ is stored in the *i*-th column of the distributed matrix *V*, and

$$H = I - V * T * V'$$

If *storev* = 'R', the vector which defines the elementary reflector $H(i)$ is stored in the *i*-th row of the distributed matrix *V*, and

$$H = I - V' * T * V.$$

Input Parameters

<i>direct</i>	(global) CHARACTER*1. Specifies the order in which the elementary reflectors are multiplied to form the block reflector: if <i>direct</i> = 'F': $H = H(1) * H(2) * \dots * H(k)$ (forward) if <i>direct</i> = 'B': $H = H(k) * \dots * H(2) * H(1)$ (backward).
<i>storev</i>	(global) CHARACTER*1. Specifies how the vectors that define the elementary reflectors are stored (See <i>Application Notes</i> below): if <i>storev</i> = 'C': columnwise; if <i>storev</i> = 'R': rowwise.
<i>n</i>	(global) INTEGER. The order of the block reflector H. $n \geq 0$.
<i>k</i>	(global) INTEGER. The order of the triangular factor T, is equal to the number of elementary reflectors. $1 \leq k \leq mb\_v$ ($= nb\_v$).
<i>v</i>	REAL for pslarft DOUBLE PRECISION for pdlarft COMPLEX for pclarft COMPLEX*16 for pzlarft. Pointer into the local memory to an array of local DIMENSION ($LOCr(iv+n-1), LOCc(jv+k-1)$) if <i>storev</i> = 'C', and ($LOCr(iv+k-1), LOCc(jv+n-1)$) if <i>storev</i> = 'R'. The distributed matrix v contains the Householder vectors. (See <i>Application Notes</i> below).
<i>iv, jv</i>	(global) INTEGER. The row and column indices in the global array v indicating the first row and the first column of the submatrix $sub(v)$, respectively.
<i>descv</i>	(global and local) INTEGER array, DIMENSION ($dlen\_$). The array descriptor for the distributed matrix v.
<i>tau</i>	(local) REAL for pslarft DOUBLE PRECISION for pdlarft

COMPLEX for pclarft
 COMPLEX*16 for pzlarft.
Array, DIMENSIONLOCr($iv+k-1$) if $incv = m\_v$, and
 LOCc($jv+k-1$) otherwise. This array contains the
 Householder scalars related to the Householder vectors.
 tau is tied to the distributed matrix v .

work (local).
 REAL for psclarft
 DOUBLE PRECISION for pdclarft
 COMPLEX for pclarft
 COMPLEX*16 for pzlarft.
Workspace array, DIMENSION ($k*(k-1)/2$).

Output Parameters

v REAL for psclarft
 DOUBLE PRECISION for pdclarft
 COMPLEX for pclarft
 COMPLEX*16 for pzlarft.

t (local)
 REAL for psclarft
 DOUBLE PRECISION for pdclarft
 COMPLEX for pclarft
 COMPLEX*16 for pzlarft.
Array, DIMENSION (nb_v, nb_v) if $storev = 'C'$, and
 (mb_v, mb_v) otherwise. It contains the k -by- k triangular
 factor of the block reflector associated with v . If $direct =$
 'F', t is upper triangular;
 if $direct = 'B'$, t is lower triangular.

Application Notes

The shape of the matrix v and the storage of the vectors that define the $H(i)$ is best illustrated by the following example with $n = 5$ and $k = 3$. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

direct = 'F' and *storev* = 'C' :

$$V(i_v : i_v + n - 1, \\ j_v : j_v + k - 1) = \begin{bmatrix} 1 & & & \\ v_1 & 1 & & \\ v_1 & v_2 & 1 & \\ v_1 & v_2 & v_3 & \\ v_1 & v_2 & v_3 & \end{bmatrix}$$

direct = 'F' and *storev* = 'R' :

$$V(i_v : i_v + k - 1, \\ j_v : j_v + n - 1) = \begin{bmatrix} 1 & v_1 & v_1 & v_1 & v_1 \\ & 1 & v_2 & v_2 & v_2 \\ & & 1 & v_3 & v_3 \end{bmatrix}$$

direct = 'B' and *storev* = 'C' :

$$V(i_v : i_v + n - 1, \\ j_v : j_v + k - 1) = \begin{bmatrix} v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \\ 1 & v_2 & v_3 \\ & 1 & v_3 \\ & & 1 \end{bmatrix}$$

direct = 'B' and *storev* = 'R' :

$$V(i_v : i_v + k - 1, \\ j_v : j_v + n - 1) = \begin{bmatrix} v_1 & v_1 & 1 \\ v_2 & v_2 & v_2 & 1 \\ v_3 & v_3 & v_3 & v_3 & 1 \end{bmatrix}$$

p?larz

Applies an elementary reflector as returned by p?tzzrf to a general matrix.

Syntax

```
call pslarz(side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc, descc,
work)
```

```
call pdlarz(side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc, descc,
work)
```

```
call pclarz(side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc, descc,
work)
```

```
call pzlarz(side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc, descc,
work)
```

Description

This routine applies a real/complex elementary reflector Q (or Q^T) to a real/complex m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$, from either the left or the right. Q is represented in the form

$$Q = I - \tau v v',$$

where τ is a real/complex scalar and v is a real/complex vector.

If $\tau = 0$, then Q is taken to be the unit matrix.

Q is a product of k elementary reflectors as returned by [p?tzzrf](#).

Input Parameters

<i>side</i>	(global) CHARACTER. if <i>side</i> = 'L': form $Q * \text{sub}(C)$, if <i>side</i> = 'R': form $\text{sub}(C) * Q$, $Q = Q^T$ (for real flavors).
<i>m</i>	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(C)$. ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(C)$. ($n \geq 0$).
<i>l</i>	(global). INTEGER. The columns of the distributed submatrix $\text{sub}(A)$ containing the meaningful part of the Householder reflectors. If <i>side</i> = 'L', $m \geq l \geq 0$, if <i>side</i> = 'R', $n \geq l \geq 0$.
<i>v</i>	(local). REAL for pslarz DOUBLE PRECISION for pdlarz COMPLEX for pclarz COMPLEX*16 for pzlarz. Pointer into the local memory to an array of DIMENSION (<i>lld_v</i> ,*) containing the local pieces of the distributed vectors v representing the Householder transformation Q ,

	$v(iv:iv+l-1, jv)$ if $side = 'L'$ and $incv = 1$, $v(iv, jv:jv+l-1)$ if $side = 'L'$ and $incv = m\_v$, $v(iv:iv+l-1, jv)$ if $side = 'R'$ and $incv = 1$, $v(iv, jv:jv+l-1)$ if $side = 'R'$ and $incv = m\_v$. The vector v in the representation of Q. v is not used if $tau = 0$.
iv, jv	(global) INTEGER. The row and column indices in the global array v indicating the first row and the first column of the submatrix $sub(v)$, respectively.
$descv$	(global and local) INTEGER array, DIMENSION ($dlen\_$). The array descriptor for the distributed matrix v .
$incv$	(global) INTEGER. The global increment for the elements of v . Only two values of $incv$ are supported in this version, namely 1 and m_v . $incv$ must not be zero.
tau	(local) REAL for pslarz DOUBLE PRECISION for pdlarz COMPLEX for pclarz COMPLEX*16 for pzlarz. Array, DIMENSION $LOCc(jv)$ if $incv = 1$, and $LOCr(iv)$ otherwise. This array contains the Householder scalars related to the Householder vectors. tau is tied to the distributed matrix v .
c	(local). REAL for pslarz DOUBLE PRECISION for pdlarz COMPLEX for pclarz COMPLEX*16 for pzlarz. Pointer into the local memory to an array of DIMENSION ($lld\_c, LOCc(jc+n-1)$), containing the local pieces of $sub(c)$.
ic, jc	(global) INTEGER. The row and column indices in the global array c indicating the first row and the first column of the submatrix $sub(c)$, respectively.

desc (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *c*.

work (local).
 REAL for pslarz
 DOUBLE PRECISION for pdlarz
 COMPLEX for pclarz
 COMPLEX*16 for pzlarz.
 Array, DIMENSION (*lwork*)
 If *incv* = 1,
 if *side* = 'L' ,
 if *ivcol* = *iccol*,
 lwork ≥ *NqC0*
 else
 lwork ≥ *MpC0* + max(1, *NqC0*)
 end if
 else if *side* = 'R' ,
 lwork ≥ *NqC0* + max(max(1, *MpC0*),
 numroc(numroc(*n+icoffc*,*nb_v*,0,0,*npcol*),*nb_v*,0,0,*lcmq*))
 end if
 else if *incv* = *m_v*,
 if *side* = 'L' ,
 lwork ≥ *MpC0* + max(max(1, *NqC0*),
 numroc(numroc(*m+iroffc*,*mb_v*,0,0,*nprow*),*mb_v*,0,0,*lcmp*))
 else if *side* = 'R' ,
 if *ivrow* = *icrow*,
 lwork ≥ *MpC0*
 else
 lwork ≥ *NqC0* + max(1, *MpC0*)
 end if
 end if
 end if.
 Here *lcm* is the least common multiple of *nprow* and *npcol* and
lcm = ilcm(*nprow*, *npcol*), *lcmp* = *lcm* / *nprow*,
lcmq = *lcm* / *npcol*,
iroffc = mod(*ic*-1, *mb_c*), *icoffc* = mod(*jc*-1, *nb_c*),
icrow = indxg2p(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*),

`iccol = indxg2p( jc, nb_c, mycol, csrc_c, npcol ),`
`mpc0 = numroc( m+iroffc, mb_c, myrow, icrow, nprow ),`
`nqc0 = numroc( n+icoffc, nb_c, mycol, iccol, npcol ),`
`ilcm, indxg2p, and numroc` are ScaLAPACK tool functions;
`myrow, mycol, nprow, and npcol` can be determined by
calling the subroutine `blacs_gridinfo`.

Output Parameters

`c` (local).
On exit, `sub(c)` is overwritten by the $Q \cdot \text{sub}(C)$ if `side = 'L'`, or `sub(C) * Q` if `side = 'R'`.

p?larzb

Applies a block reflector or its transpose/conjugate-transpose as returned by p?tzzrf to a general matrix.

Syntax

`call pslarzb(side, trans, direct, storev, m, n, k, l, v, iv, jv, descv, t, c, ic, jc, descc, work)`

`call pdlarzb(side, trans, direct, storev, m, n, k, l, v, iv, jv, descv, t, c, ic, jc, descc, work)`

`call pclarzb(side, trans, direct, storev, m, n, k, l, v, iv, jv, descv, t, c, ic, jc, descc, work)`

`call pzlarzb(side, trans, direct, storev, m, n, k, l, v, iv, jv, descv, t, c, ic, jc, descc, work)`

Description

This routine applies a real/complex block reflector Q or its transpose Q^T (conjugate transpose Q^H for complex flavors) to a real/complex distributed m -by- n matrix `sub(C) = C(ic:ic+m-1, jc:jc+n-1)` from the left or the right.

Q is a product of k elementary reflectors as returned by `p?tzzrf`.

Currently, only `storev = 'R'` and `direct = 'B'` are supported.

Input Parameters

<i>side</i>	(global) CHARACTER. if <i>side</i> = 'L': apply Q or Q^T (Q^H for complex flavors) from the Left; if <i>side</i> = 'R': apply Q or Q^T (Q^H for complex flavors) from the Right.
<i>trans</i>	(global) CHARACTER. if <i>trans</i> = 'N': No transpose, apply Q; If <i>trans</i>='T': Transpose, apply Q^T (real flavors); If <i>trans</i>='C': Conjugate transpose, apply Q^H (complex flavors).
<i>direct</i>	(global) CHARACTER. Indicates how H is formed from a product of elementary reflectors. if <i>direct</i> = 'F': $H = H(1) * H(2) * \dots * H(k)$ - forward (not supported) ; if <i>direct</i> = 'B': $H = H(k) * \dots * H(2) * H(1)$ - backward.
<i>storev</i>	(global) CHARACTER. Indicates how the vectors that define the elementary reflectors are stored: if <i>storev</i> = 'C': columnwise (not supported). if <i>storev</i> = 'R': rowwise.
<i>m</i>	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(<i>C</i>). ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(<i>C</i>). ($n \geq 0$).
<i>k</i>	(global) INTEGER. The order of the matrix T. (= the number of elementary reflectors whose product defines the block reflector).
<i>l</i>	(global) INTEGER. The columns of the distributed submatrix sub(<i>A</i>) containing the meaningful part of the Householder reflectors.

	If $side = 'L', m \geq l \geq 0,$ if $side = 'R', n \geq l \geq 0.$
v	(local). REAL for pslarzb DOUBLE PRECISION for pdlarzb COMPLEX for pclarzb COMPLEX*16 for pzlarzb. Pointer into the local memory to an array of DIMENSION($lld\_v, LOCc(jv+m-1)$) if $side = 'L',$ ($lld\_v, LOCc(jv+m-1)$) if $side = 'R'.$ It contains the local pieces of the distributed vectors v representing the Householder transformation as returned by p?tzzrf. $lld\_v \geq LOCr(iv+k-1).$
iv, jv	(global) INTEGER. The row and column indices in the global array v indicating the first row and the first column of the submatrix sub(v), respectively.
$descv$	(global and local) INTEGER array, DIMENSION ($dlen\_$). The array descriptor for the distributed matrix v.
t	(local) REAL for pslarzb DOUBLE PRECISION for pdlarzb COMPLEX for pclarzb COMPLEX*16 for pzlarzb. Array, DIMENSION mb_v by $mb\_v.$ The lower triangular matrix T in the representation of the block reflector.
c	(local). REAL for pslarfb DOUBLE PRECISION for pdlarfb COMPLEX for pclarfb COMPLEX*16 for pzlarfb. Pointer into the local memory to an array of DIMENSION($lld\_c, LOCc(jc+n-1)$). On entry, the m-by-n distributed matrix sub(c).

ic, jc (global) INTEGER.
The row and column indices in the global array *c* indicating the first row and the first column of the submatrix sub(*c*), respectively.

descc (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *c*.

work (local).
REAL for pslarzb
DOUBLE PRECISION for pdlarzb
COMPLEX for pclarzb
COMPLEX*16 for pzlarzb.
Array, DIMENSION (*lwork*).
If storev = 'C' ,
 if side = 'L' ,
 $lwork \geq (nqc0 + mpc0) * k$
 else if side = 'R' ,
 $lwork \geq (nqc0 + \max(npv0 + \text{numroc}(\text{numroc}(n+icoffc, nb\_v, 0, 0, npc0),$
 $nb\_v, 0, 0, lcmq), mpc0)) * k$
 end if
else if storev = 'R' ,
 if side = 'L' ,
 $lwork \geq (mpc0 + \max(mqv0 + \text{numroc}(\text{numroc}(m+iroffc, mb\_v, 0, 0, nprow),$
 $mb\_v, 0, 0, lcmp), nqc0)) * k$
 else if side = 'R' ,
 $lwork \geq (mpc0 + nqc0) * k$
 end if
end if.

Here $lcmq = lcm / npc0$ with $lcm = iclm(nprow, npc0)$,

```

iroffv = mod(iv-1, mb_v), icoffv = mod( jv-1,
nb_v),
ivrow = indxg2p(iv, mb_v, myrow, rsrc_v, nprow),
ivcol = indxg2p(jv, nb_v, mycol, csrc_v, npcol),
mqv0 = numroc(m+icoffv, nb_v, mycol, ivcol,
npcol),
npv0 = numroc(n+iroffv, mb_v, myrow, ivrow,
nprow),
iroffc = mod(ic-1, mb_c ), icoffc= mod( jc-1,
nb_c),
icrow= indxg2p(ic, mb_c, myrow, rsrc_c, nprow),
iccol= indxg2p(jc, nb_c, mycol, csrc_c, npcol),
mpc0 = numroc(m+iroffc, mb_c, myrow, icrow,
nprow),
npc0 = numroc(n+icoffc, mb_c, myrow, icrow,
nprow),
nqc0 = numroc(n+icoffc, nb_c, mycol, iccol,
npcol),
ilcm, indxg2p, and numroc are ScaLAPACK tool functions;
myrow, mycol, nprow, and npcol can be determined by
calling the subroutine blacs_gridinfo.

```

Output Parameters

c

(local).

On exit, $\text{sub}(c)$ is overwritten by the $Q \cdot \text{sub}(C)$, or $Q' \cdot \text{sub}(C)$, or $\text{sub}(C) \cdot Q$, or $\text{sub}(C) \cdot Q'$, where Q' is the transpose (conjugate transpose) of Q .

p?larzc

Applies (multiplies by) the conjugate transpose of an elementary reflector as returned by p?tzzrf to a general matrix.

Syntax

```
call pclarzc(side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc, descc,
work)

call pzlarzc(side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc, descc,
work)
```

Description

This routine applies a complex elementary reflector Q^H to a complex m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$, from either the left or the right. Q is represented in the form

$$Q = I - \tau v v^H,$$

where τ is a complex scalar and v is a complex vector.

If $\tau = 0$, then Q is taken to be the unit matrix.

Q is a product of k elementary reflectors as returned by p?tzzrf.

Input Parameters

<i>side</i>	(global) CHARACTER. if <i>side</i> = 'L': form $Q^H * \text{sub}(C)$; if <i>side</i> = 'R': form $\text{sub}(C) * Q^H$.
<i>m</i>	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(C)$. ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(C)$. ($n \geq 0$).
<i>l</i>	(global) INTEGER.

	The columns of the distributed submatrix $\text{sub}(\underline{A})$ containing the meaningful part of the Householder reflectors. If $\text{side} = \text{'L'}, m \geq l \geq 0$, if $\text{side} = \text{'R'}, n \geq l \geq 0$.
v	(local). COMPLEX for pclarzc COMPLEX*16 for pzlarzc. Pointer into the local memory to an array of DIMENSION ($lld_v, *$) containing the local pieces of the distributed vectors v representing the Householder transformation Q, $v(iv:iv+l-1, jv)$ if $\text{side} = \text{'L'}$ and $incv = 1$, $v(iv, jv:jv+l-1)$ if $\text{side} = \text{'L'}$ and $incv = m_v$, $v(iv:iv+l-1, jv)$ if $\text{side} = \text{'R'}$ and $incv = 1$, $v(iv, jv:jv+l-1)$ if $\text{side} = \text{'R'}$ and $incv = m_v$. The vector v in the representation of Q. v is not used if $\tau = 0$.
iv, jv	(global) INTEGER. The row and column indices in the global array v indicating the first row and the first column of the submatrix $\text{sub}(v)$, respectively.
$descv$	(global and local) INTEGER array, DIMENSION ($dlen\_v$). The array descriptor for the distributed matrix v.
$incv$	(global). INTEGER. The global increment for the elements of v. Only two values of $incv$ are supported in this version, namely 1 and m_v. $incv$ must not be zero.
τ	(local) COMPLEX for pclarzc COMPLEX*16 for pzlarzc. Array, DIMENSION $LOCc(jv)$ if $incv = 1$, and $LOCr(iv)$ otherwise. This array contains the Householder scalars related to the Householder vectors. τ is tied to the distributed matrix v.
c	(local). COMPLEX for pclarzc COMPLEX*16 for pzlarzc.

	Pointer into the local memory to an array of <code>DIMENSION(<i>lld_c</i>, <i>LOCc(jc+n-1)</i>)</code>, containing the local pieces of <code>sub(c)</code>.
<i>ic, jc</i>	(global) <code>INTEGER</code>. The row and column indices in the global array <i>c</i> indicating the first row and the first column of the submatrix <code>sub(c)</code>, respectively.
<i>desc</i>	(global and local) <code>INTEGER</code> array, <code>DIMENSION(<i>dlen_</i>)</code>. The array descriptor for the distributed matrix <i>c</i>.

work

```
(local).
If incv = 1,
    if side = 'L' ,
        if ivcol = iccol,
            lwork ≥ nqc0
        else
            lwork ≥ mpc0 + max(1, nqc0)
        end if
    else if side = 'R' ,
        lwork ≥ nqc0 + max(max(1, mpc0),
numroc(numroc(n+icoffc, nb_v, 0, 0, npcol),
            nb_v, 0, 0, lcmq))
        end if
    else if incv = m_v,
        if side = 'L' ,
            lwork ≥ mpc0 + max(max(1, nqc0),
numroc(numroc(m+iroffc, mb_v, 0, 0, nprow),
            mb_v, 0, 0, lcmp))
        else if side = 'R',
            if ivrow = icrow,
                lwork ≥ mpc0
            else
                lwork ≥ nqc0 + max(1, mpc0)
            end if
        end if
    end if
end if
```

Here *lcm* is the least common multiple of *nprow* and *npcol*;

```
lcm = ilcm(nprow, npc0l), lcmp = lcm/nprow, lcmq =
lcm/npcol,
iroffc = mod(ic-1, mb_c), icoffc = mod(jc-1,
nb_c),
icrow = indxg2p(ic, mb_c, myrow, rsrc_c, nprow),
iccol = indxg2p(jc, nb_c, mycol, csrc_c, npc0l),
mpc0 = numroc(m+iroffc, mb_c, myrow, icrow,
nprow),
nqc0 = numroc(n+icoffc, nb_c, mycol, iccol,
npc0l),
ilcm, indxg2p, and numroc are ScaLAPACK tool functions;
myrow, mycol, nprow, and npc0l can be determined by
calling the subroutine blacs_gridinfo.
```

Output Parameters

c (local).
On exit, *sub(c)* is overwritten by the $Q^H * \text{sub}(C)$ if *side* = 'L', or *sub(C) * Q^H* if *side* = 'R'.

p?larzt

Forms the triangular factor *T* of a block reflector
 $H = I - V * T * V^H$ as returned by p?tzzrf.

Syntax

```
call pslarzt(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
call pdlarzt(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
call pclarzt(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
call pzlarzt(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
```

Description

This routine forms the triangular factor *T* of a real/complex block reflector *H* of order greater than *n*, which is defined as a product of *k* elementary reflectors as returned by p?tzzrf.

If *direct* = 'F', $H = H(1) * H(2) * \dots * H(k)$, and *T* is upper triangular;

If *direct* = 'B', $H = H(k) * \dots * H(2) * H(1)$, and *T* is lower triangular.

If $storev = 'C'$, the vector which defines the elementary reflector $H(i)$, is stored in the i -th column of the array v , and

$$H = I - v * t * v'.$$

If $storev = 'R'$, the vector, which defines the elementary reflector $H(i)$, is stored in the i -th row of the array v , and

$$H = I - v' * t * v$$

Currently, only $storev = 'R'$ and $direct = 'B'$ are supported.

Input Parameters

<i>direct</i>	(global) CHARACTER. Specifies the order in which the elementary reflectors are multiplied to form the block reflector: if $direct = 'F'$: $H = H(1) * H(2) * \dots * H(k)$ (Forward, not supported) if $direct = 'B'$: $H = H(k) * \dots * H(2) * H(1)$ (Backward).
<i>storev</i>	(global) CHARACTER. Specifies how the vectors which defines the elementary reflectors are stored: if $storev = 'C'$: columnwise (not supported); if $storev = 'R'$: rowwise.
<i>n</i>	(global). INTEGER. The order of the block reflector H . $n \geq 0$.
<i>k</i>	(global). INTEGER. The order of the triangular factor T (= the number of elementary reflectors). $1 \leq k \leq mb\_v (= nb\_v)$.
<i>v</i>	REAL for pslarzt DOUBLE PRECISION for pdlarzt COMPLEX for pclarzt COMPLEX*16 for pzlarzt. Pointer into the local memory to an array of local $DIMENSION(LOCr(iv+k-1), LOCc(jv+n-1))$. The distributed matrix v contains the Householder vectors. See <i>Application Notes</i> below.

<i>iv, jv</i>	(global) INTEGER. The row and column indices in the global array <i>v</i> indicating the first row and the first column of the submatrix <i>sub(v)</i> , respectively.
<i>descv</i>	(global and local) INTEGER array, DIMENSION (<i>dlen_</i>). The array descriptor for the distributed matrix <i>v</i> .
<i>tau</i>	(local) REAL for pslarzt DOUBLE PRECISION for pdlarzt COMPLEX for pclarzt COMPLEX*16 for pzlarzt. Array, DIMENSION <i>LOCr(iv+k-1)</i> if <i>incv = m_v</i> , and <i>LOCc(jv+k-1)</i> otherwise. This array contains the Householder scalars related to the Householder vectors. <i>tau</i> is tied to the distributed matrix <i>v</i> .
<i>work</i>	(local). REAL for pslarzt DOUBLE PRECISION for pdlarzt COMPLEX for pclarzt COMPLEX*16 for pzlarzt. Workspace array, DIMENSION (<i>k*(k-1)/2</i>).

Output Parameters

<i>v</i>	REAL for pslarzt DOUBLE PRECISION for pdlarzt COMPLEX for pclarzt COMPLEX*16 for pzlarzt.
<i>t</i>	(local) REAL for pslarzt DOUBLE PRECISION for pdlarzt COMPLEX for pclarzt COMPLEX*16 for pzlarzt. Array, DIMENSION (<i>mb_v, mb_v</i>). It contains the <i>k</i> -by- <i>k</i> triangular factor of the block reflector associated with <i>v</i> . <i>t</i> is lower triangular.

Application Notes

The shape of the matrix v and the storage of the vectors which define the $H(i)$ is best illustrated by the following example with $n = 5$ and $k = 3$. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

direct='F' and storev='C'

$$v = \begin{bmatrix} v1 & v2 & v3 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \end{bmatrix}$$

$$v = \begin{bmatrix} . & . & . \\ . & . & . \\ 1 & . & . \\ . & 1 & . \\ . & . & 1 \end{bmatrix}$$

direct='F' and storev='R'

$$\begin{bmatrix} \overbrace{v1 \ v1 \ v1 \ v1 \ v1}^v & . & . & . & 1 \\ v2 \ v2 \ v2 \ v2 \ v2 & . & . & . & 1 \\ v3 \ v3 \ v3 \ v3 \ v3 & . & . & . & 1 \end{bmatrix}$$

direct='B' and storev='C'

$$v = \begin{bmatrix} 1 \\ . & 1 \\ . & . & 1 \\ . & . & . \\ . & . & . \end{bmatrix}$$

$$\begin{bmatrix} v1 & v2 & v3 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \end{bmatrix}$$

direct='B' and storev='R':

$$\begin{bmatrix} 1 & . & . & . & \overbrace{.v1 \ v1 \ v1 \ v1 \ v1}^Y \\ . & 1 & . & . & .v2 \ v2 \ v2 \ v2 \ v2 \\ . & . & 1 & . & .v3 \ v3 \ v3 \ v3 \ v3 \end{bmatrix}$$

p?lascl

Multiplies a general rectangular matrix by a real scalar defined as C_{to}/C_{from} .

Syntax

```
call pslascl(type, cfrom, cto, m, n, a, ia, ja, desca, info)
call pdlascl(type, cfrom, cto, m, n, a, ia, ja, desca, info)
call pclascl(type, cfrom, cto, m, n, a, ia, ja, desca, info)
call pzlascl(type, cfrom, cto, m, n, a, ia, ja, desca, info)
```

Description

This routine multiplies the m -by- n real/complex distributed matrix $\text{sub}(A)$ denoting $A(ia:ia+m-1, ja:ja+n-1)$ by the real/complex scalar $cto/cfrom$. This is done without over/underflow as long as the final result $cto*A(i, j)/cfrom$ does not over/underflow. *type* specifies that $\text{sub}(A)$ may be full, upper triangular, lower triangular or upper Hessenberg.

Input Parameters

<i>type</i>	(global) CHARACTER. <i>type</i> indicates the storage type of the input distributed matrix. if <i>type</i> = 'G': $\text{sub}(A)$ is a full matrix, if <i>type</i> = 'L': $\text{sub}(A)$ is a lower triangular matrix, if <i>type</i> = 'U': $\text{sub}(A)$ is an upper triangular matrix, if <i>type</i> = 'H': $\text{sub}(A)$ is an upper Hessenberg matrix.
<i>cfrom, cto</i>	(global)

	REAL for pslascl/pclascl DOUBLE PRECISION for pdlascl/pzlascl. The distributed matrix $\text{sub}(A)$ is multiplied by $cto/cfrom$. $A(i,j)$ is computed without over/underflow if the final result $cto * A(i,j) / cfrom$ can be represented without over/underflow. $cfrom$ must be nonzero.
<i>m</i>	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$. ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$. ($n \geq 0$).
<i>a</i>	(local input/local output) REAL for pslascl DOUBLE PRECISION for pdlascl COMPLEX for pcascl COMPLEX*16 for pzlascl. Pointer into the local memory to an array of DIMENSION(<i>lld_a</i>, <i>LOCc(ja+n-1)</i>). This array contains the local pieces of the distributed matrix $\text{sub}(A)$.
<i>ia, ja</i>	(global) INTEGER. The column and row indices in the global array <i>A</i> indicating the first row and column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER . Array of DIMENSION (<i>dlen_</i>).The array descriptor for the distributed matrix <i>A</i>.

Output Parameters

<i>a</i>	(local). On exit, this array contains the local pieces of the distributed matrix multiplied by $cto/cfrom$.
<i>info</i>	(local) INTEGER.

if *info* = 0: the execution is successful.
 if *info* < 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = $-(i*100+j)$,
 if the *i*-th argument is a scalar and had an illegal value, then *info* = $-i$.

p?laset

Initializes the offdiagonal elements of a matrix to alpha and the diagonal elements to beta.

Syntax

```
call pslaset(uplo, m, n, alpha, beta, a, ia, ja, desca)
call pdlaset(uplo, m, n, alpha, beta, a, ia, ja, desca)
call pclaset(uplo, m, n, alpha, beta, a, ia, ja, desca)
call pzlaset(uplo, m, n, alpha, beta, a, ia, ja, desca)
```

Description

This routine initializes an *m*-by-*n* distributed matrix sub(*A*) denoting *A*(*ia:ia+m-1*, *ja:ja+n-1*) to *beta* on the diagonal and *alpha* on the offdiagonals.

Input Parameters

<i>uplo</i>	(global) CHARACTER. Specifies the part of the distributed matrix sub(<i>A</i>) to be set: if <i>uplo</i> = 'U': upper triangular part; the strictly lower triangular part of sub(<i>A</i>) is not changed; if <i>uplo</i> = 'L': lower triangular part; the strictly upper triangular part of sub(<i>A</i>) is not changed. Otherwise: all of the matrix sub(<i>A</i>) is set.
<i>m</i>	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(<i>A</i>). ($m \geq 0$).
<i>n</i>	(global) INTEGER.

The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$.
 $(n \geq 0)$.

alpha (global).
 REAL for pslaset
 DOUBLE PRECISION for pdlaset
 COMPLEX for pclaset
 COMPLEX*16 for pzlaset.
 The constant to which the offdiagonal elements are to be set.

beta (global).
 REAL for pslaset
 DOUBLE PRECISION for pdlaset
 COMPLEX for pclaset
 COMPLEX*16 for pzlaset.
 The constant to which the diagonal elements are to be set.

Output Parameters

a (local).
 REAL for pslaset
 DOUBLE PRECISION for pdlaset
 COMPLEX for pclaset
 COMPLEX*16 for pzlaset.
 Pointer into the local memory to an array of
 DIMENSION(*lld_a*, *LOCc(ja+n-1)*).
 This array contains the local pieces of the distributed matrix $\text{sub}(A)$ to be set. On exit, the leading m -by- n submatrix $\text{sub}(A)$ is set as follows:

if *uplo* = 'U', $A(ia+i-1, ja+j-1) = \alpha, 1 \leq i \leq j-1,$
 $1 \leq j \leq n,$

if *uplo* = 'L', $A(ia+i-1, ja+j-1) = \alpha, j+1 \leq i \leq m,$
 $1 \leq j \leq n,$

otherwise, $A(ia+i-1, ja+j-1) = \alpha, 1 \leq i \leq m, 1 \leq j \leq n,$
 $ia+i.ne.ja+j,$ and, for all *uplo*, $A(ia+i-1, ja+i-1) =$
 $\beta, 1 \leq i \leq \min(m, n).$

<i>ia, ja</i>	(global) INTEGER. The column and row indices in the global array <i>A</i> indicating the first row and column of the submatrix sub(<i>A</i>), respectively.
<i>desca</i>	(global and local) INTEGER . Array of DIMENSION (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .

p?lasmsub

Looks for a small subdiagonal element from the bottom of the matrix that it can safely set to zero.

Syntax

```
call pslasmsub(a, desca, i, l, k, smlnum, buf, lwork)
call pdlasmsub(a, desca, i, l, k, smlnum, buf, lwork)
```

Description

This routine looks for a small subdiagonal element from the bottom of the matrix that it can safely set to zero. This routine does a global maximum and must be called by all processes.

Input Parameters

<i>a</i>	(global) REAL for pslasmsub DOUBLE PRECISION for pdlasmsub Array, DIMENSION (<i>desca(lld_)</i> , *). On entry, the Hessenberg matrix whose tridiagonal part is being scanned. Unchanged on exit.
<i>desca</i>	(global and local) INTEGER. Array of DIMENSION (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>i</i>	(global) INTEGER. The global location of the bottom of the unreduced submatrix of <i>A</i> . Unchanged on exit.
<i>l</i>	(global) INTEGER.

The global location of the top of the unreduced submatrix of A .
 Unchanged on exit.

smlnum (global)
 REAL for pslasmsub
 DOUBLE PRECISION for pdlasmsub
 On entry, a “small number” for the given matrix. Unchanged on exit.

lwork (global) INTEGER.
 On exit, *lwork* is the size of the work buffer.
 This must be at least $2 * \text{ceil}(\text{ceil}((i-1)/hbl) / \text{lcm}(nprow, npcot))$. Here *lcm* is least common multiple, and $nprow \times npcot$ is the logical grid size.

Output Parameters

k (global) INTEGER.
 On exit, this yields the bottom portion of the unreduced submatrix. This will satisfy: $1 \leq m \leq i-1$.

buf (local).
 REAL for pslasmsub
 DOUBLE PRECISION for pdlasmsub
 Array of size *lwork*.

p?lassq

Updates a sum of squares represented in scaled form.

Syntax

```
call pslasq(n, x, ix, jx, descx, incx, scale, sumsq)
call pdlasq(n, x, ix, jx, descx, incx, scale, sumsq)
call pclasq(n, x, ix, jx, descx, incx, scale, sumsq)
call pzlasq(n, x, ix, jx, descx, incx, scale, sumsq)
```

Description

This routine returns the values *scl* and *sumsq* such that

$$scl^2 * sumsq = x(1)^2 + \dots + x(n)^2 + scale^2 * sumsq,$$

where $x(i) = \text{sub}(X) = X(ix + (jx-1)*descx(m_) + (i - 1)*incx)$ for
plassq/pdllassq ,

and $x(i) = \text{sub}(X) = \text{abs}(X(ix + (jx-1)*descx(m_) + (i - 1)*incx))$ for
pclassq/pzlassq.

For real routines plassq/pdllassq the value of *sumsq* is assumed to be non-negative and *scl* returns the value

$$scl = \max(scale, \text{abs}(x(i))).$$

For complex routines pclassq/pzlassq the value of *sumsq* is assumed to be at least unity and the value of *ssq* will then satisfy

$$1.0 \leq ssq \leq sumsq + 2n$$

Value *scale* is assumed to be non-negative and *scl* returns the value

$$scl = \max_i \left(scale, \text{abs}(\text{real}(x(i))), \text{abs}(\text{aimag}(x(i))) \right)$$

For all routines p?lassq values *scale* and *sumsq* must be supplied in *scale* and *sumsq* respectively, and *scale* and *sumsq* are overwritten by *scl* and *ssq* respectively.

All routines p?lassq make only one pass through the vector $\text{sub}(x)$.

Input Parameters

<i>n</i>	(global) INTEGER. The length of the distributed vector $\text{sub}(x)$.
<i>x</i>	REAL for plassq DOUBLE PRECISION for pdllassq COMPLEX for pclassq COMPLEX*16 for pzlassq. The vector for which a scaled sum of squares is computed: $x(ix + (jx-1)*m_x + (i - 1)*incx)$, $1 \leq i \leq n$.
<i>ix</i>	(global) INTEGER.

	The row index in the global array x indicating the first row of $\text{sub}(x)$.
jx	(global) INTEGER. The column index in the global array x indicating the first column of $\text{sub}(x)$.
$descx$	(global and local) INTEGER array of DIMENSION ($dlen\_$). The array descriptor for the distributed matrix x .
$incx$	(global) INTEGER. The global increment for the elements of x . Only two values of $incx$ are supported in this version, namely 1 and m_x . The argument $incx$ must not equal zero.
$scale$	(local). REAL for psllassq/pclassq DOUBLE PRECISION for pdlassq/pzlassq. On entry, the value $scale$ in the equation above.
$sumsq$	(local) REAL for psllassq/pclassq DOUBLE PRECISION for pdlassq/pzlassq. On entry, the value $sumsq$ in the equation above.

Output Parameters

$scale$	(local). On exit, $scale$ is overwritten with scl , the scaling factor for the sum of squares.
$sumsq$	(local). On exit, $sumsq$ is overwritten with the value $smsq$, the basic sum of squares from which scl has been factored out.

p?laswp

Performs a series of row interchanges on a general rectangular matrix.

Syntax

```
call pslaswp(direc, rowcol, n, a, ia, ja, desca, k1, k2, ipiv)
call pdlaswp(direc, rowcol, n, a, ia, ja, desca, k1, k2, ipiv)
call pclaswp(direc, rowcol, n, a, ia, ja, desca, k1, k2, ipiv)
call pzlaswp(direc, rowcol, n, a, ia, ja, desca, k1, k2, ipiv)
```

Description

This routine performs a series of row or column interchanges on the distributed matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$. One interchange is initiated for each of rows or columns $k1$ through $k2$ of $\text{sub}(A)$. This routine assumes that the pivoting information has already been broadcast along the process row or column. Also note that this routine will only work for $k1-k2$ being in the same *mb* (or *nb*) block. If you want to pivot a full matrix, use [p?lapiv](#).

Input Parameters

<i>direc</i>	(global) CHARACTER. Specifies in which order the permutation is applied: = 'F' - forward, = 'B' - backward.
<i>rowcol</i>	(global) CHARACTER. Specifies if the rows or columns are permuted: = 'R' - rows, = 'C' - columns.
<i>n</i>	(global) INTEGER. If <i>rowcol</i> ='R', the length of the rows of the distributed matrix $A(*, ja:ja+n-1)$ to be permuted; If <i>rowcol</i> ='C', the length of the columns of the distributed matrix $A(ia:ia+n-1, *)$ to be permuted;
<i>a</i>	(local) REAL for pslaswp DOUBLE PRECISION for pdlaswp COMPLEX for pclaswp

	COMPLEX*16 for pzlaswp. Pointer into the local memory to an array of DIMENSION (lld_a, *). On entry, this array contains the local pieces of the distributed matrix to which the row/columns interchanges will be applied.
<i>ia</i>	(global) INTEGER. The row index in the global array <i>A</i> indicating the first row of sub(<i>A</i>).
<i>ja</i>	(global) INTEGER. The column index in the global array <i>A</i> indicating the first column of sub(<i>A</i>).
<i>desca</i>	(global and local) INTEGER array of DIMENSION (dlen_). The array descriptor for the distributed matrix <i>A</i> .
<i>k1</i>	(global) INTEGER. The first element of <i>ipiv</i> for which a row or column interchange will be done.
<i>k2</i>	(global) INTEGER. The last element of <i>ipiv</i> for which a row or column interchange will be done.
<i>ipiv</i>	(local) INTEGER. Array, DIMENSION LOCr(m_a)+mb_a for row pivoting and LOCr(n_a)+nb_a for column pivoting. This array is tied to the matrix <i>A</i> , <i>ipiv</i> (<i>k</i>)=1 implies rows (or columns) <i>k</i> and 1 are to be interchanged.

Output Parameters

<i>A</i>	(local) REAL for pslaswp DOUBLE PRECISION for pdlaswp COMPLEX for pclaswp COMPLEX*16 for pzlaswp. On exit, the permuted distributed matrix.

p?latra

Computes the trace of a general square distributed matrix.

Syntax

```
val = pslatra(n, a, ia, ja, desca)
val = pdlatra(n, a, ia, ja, desca)
val = pclatra(n, a, ia, ja, desca)
val = pzlatra(n, a, ia, ja, desca)
```

Description

This function computes the trace of an n -by- n distributed matrix $\text{sub}(A)$ denoting $A(ia:ia+n-1, ja:ja+n-1)$. The result is left on every process of the grid.

Input Parameters

n	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix $\text{sub}(A)$. $n \geq 0$.
a	(local). Real for pslatra DOUBLE PRECISION for pdlatra COMPLEX for pclatra COMPLEX*16 for pzlatra. Pointer into the local memory to an array of <code>DIMENSION(lld_a, LOCc(ja+n-1))</code> containing the local pieces of the distributed matrix, the trace of which is to be computed.
ia, ja	(global) INTEGER. The row and column indices respectively in the global array A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
$desca$	(global and local) INTEGER array of <code>DIMENSION(dlen_)</code> . The array descriptor for the distributed matrix A .

Output Parameters

`val` The value returned by the fuction.

p?latrd

Reduces the first nb rows and columns of a symmetric/Hermitian matrix A to real tridiagonal form by an orthogonal/unitary similarity transformation.

Syntax

```
call pslatrd(uplo, n, nb, a, ia, ja, desca, d, e, tau, w, iw, jw, descw, work)
call pdlatrd(uplo, n, nb, a, ia, ja, desca, d, e, tau, w, iw, jw, descw, work)
call pclatrd(uplo, n, nb, a, ia, ja, desca, d, e, tau, w, iw, jw, descw, work)
call pzlatrd(uplo, n, nb, a, ia, ja, desca, d, e, tau, w, iw, jw, descw, work)
```

Description

This routine reduces nb rows and columns of a real symmetric or complex Hermitian matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ to symmetric/complex tridiagonal form by an orthogonal/unitary similarity transformation $Q'^* \text{sub}(A) Q$, and returns the matrices V and W , which are needed to apply the transformation to the unreduced part of $\text{sub}(A)$.

If $uplo = U$, p?latrd reduces the last nb rows and columns of a matrix, of which the upper triangle is supplied;

if $uplo = L$, p?latrd reduces the first nb rows and columns of a matrix, of which the lower triangle is supplied.

This is an auxiliary routine called by p?sytrd/p?hetrd.

Input Parameters

`uplo` (global) CHARACTER.
Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix $\text{sub}(A)$ is stored:
= 'U': Upper triangular
= 'L': Lower triangular.

`n` (global) INTEGER.

	The number of rows and columns to be operated on, that is, the order of the distributed submatrix $\text{sub}(A)$. $n \geq 0$.
<i>nb</i>	(global) INTEGER. The number of rows and columns to be reduced.
<i>a</i>	REAL for <code>pslatrd</code> DOUBLE PRECISION for <code>pdlatrd</code> COMPLEX for <code>pclatrd</code> COMPLEX*16 for <code>pzlatrd</code> . Pointer into the local memory to an array of <code>DIMENSION (lld_a, LOcc(ja+n-1))</code> . On entry, this array contains the local pieces of the symmetric/Hermitian distributed matrix $\text{sub}(A)$. If <i>uplo</i> = U, the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If <i>uplo</i> = L, the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.
<i>ia</i>	(global) INTEGER. The row index in the global array <i>A</i> indicating the first row of $\text{sub}(A)$.
<i>ja</i>	(global) INTEGER. The column index in the global array <i>A</i> indicating the first column of $\text{sub}(A)$.
<i>desca</i>	(global and local) INTEGER array of <code>DIMENSION (dlen_)</code> . The array descriptor for the distributed matrix <i>A</i> .
<i>iw</i>	(global) INTEGER. The row index in the global array <i>W</i> indicating the first row of $\text{sub}(W)$.
<i>jw</i>	(global) INTEGER. The column index in the global array <i>W</i> indicating the first column of $\text{sub}(W)$.
<i>descw</i>	(global and local) INTEGER array of <code>DIMENSION (dlen_)</code> . The array descriptor for the distributed matrix <i>W</i> .
<i>work</i>	(local) REAL for <code>pslatrd</code>

DOUBLE PRECISION for pdlatrd
 COMPLEX for pclatrd
 COMPLEX*16 for pzlatrd.
 Workspace array of DIMENSION (nb_a).

Output Parameters

a (local)
 On exit, if $uplo = 'U'$, the last nb columns have been reduced to tridiagonal form, with the diagonal elements overwriting the diagonal elements of $sub(A)$; the elements above the diagonal with the array τ represent the orthogonal/unitary matrix Q as a product of elementary reflectors;
 if $uplo = 'L'$, the first nb columns have been reduced to tridiagonal form, with the diagonal elements overwriting the diagonal elements of $sub(A)$; the elements below the diagonal with the array τ represent the orthogonal/unitary matrix Q as a product of elementary reflectors.

d (local)
 REAL for pslatrd/pclatrd
 DOUBLE PRECISION for pdlatrd/pzlatrd.
 Array, DIMENSION $LOCc(ja+n-1)$.
 The diagonal elements of the tridiagonal matrix T : $d(i) = a(i, i)$. d is tied to the distributed matrix A .

e (local)
 REAL for pslatrd/pclatrd
 DOUBLE PRECISION for pdlatrd/pzlatrd.
 Array, DIMENSION $LOCc(ja+n-1)$ if $uplo = 'U'$, $LOCc(ja+n-2)$ otherwise.
 The off-diagonal elements of the tridiagonal matrix T :
 $e(i) = a(i, i + 1)$ if $uplo = 'U'$,
 $e(i) = a(i + 1, i)$ if $uplo = 'L'$.
 e is tied to the distributed matrix A .

tau (local)
 REAL for pslatrd
 DOUBLE PRECISION for pdlatrd
 COMPLEX for pclatrd

COMPLEX*16 for pzlatrd.
 Array, DIMENSION $LOCc(ja+n-1)$. This array contains the scalar factors τ of the elementary reflectors. τ is tied to the distributed matrix A .

w (local)
 REAL for pslatrd
 DOUBLE PRECISION for pdlatrd
 COMPLEX for pclatrd
 COMPLEX*16 for pzlatrd.
 Pointer into the local memory to an array of DIMENSION (lld_w, nb_w) . This array contains the local pieces of the n -by- nb_w matrix w required to update the unreduced part of $sub(A)$.

Application Notes

If $uplo = 'U'$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(n) * H(n-1) * \dots * H(n-nb+1)$$

Each $H(i)$ has the form

$$H(i) = I - \tau * v * v',$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(i:n) = 0$ and $v(i-1) = 1$; $v(1:i-1)$ is stored on exit in $A(ia:ia+i-1, ja+i)$, and τ in $\tau(ja+i-1)$.

If $uplo = 'L'$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(1) * H(2) * \dots * H(nb)$$

Each $H(i)$ has the form

$$H(i) = I - \tau * v * v',$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i) = 0$ and $v(i+1) = 1$; $v(i+2:n)$ is stored on exit in $A(ia+i+1: ia+n-1, ja+i-1)$, and τ in $\tau(ja+i-1)$.

The elements of the vectors v together form the n -by- nb matrix v which is needed, with w , to apply the transformation to the unreduced part of the matrix, using a symmetric/Hermitian rank- $2k$ update of the form:

$$sub(A) := sub(A) - v w' - w v'.$$

The contents of a on exit are illustrated by the following examples with

$$n = 5 \text{ and } nb = 2:$$

$$\begin{array}{cc}
 \text{if } \text{uplo} = 'U': & \text{if } \text{uplo} = 'L': \\
 \begin{bmatrix} a & a & a & v_4 & v_5 \\ & a & a & v_4 & v_5 \\ & & a & 1 & v_5 \\ & & & d & 1 \\ & & & & d \end{bmatrix} & \begin{bmatrix} d & & & & \\ 1 & d & & & \\ v_1 & 1 & a & & \\ v_1 & v_2 & a & a & \\ v_1 & v_2 & a & a & a \end{bmatrix}
 \end{array}$$

where d denotes a diagonal element of the reduced matrix, a denotes an element of the original matrix that is unchanged, and v_i denotes an element of the vector defining $H(i)$.

p?latrs

Solves a triangular system of equations with the scale factor set to prevent overflow.

Syntax

```
call pslatrs(uplo, trans, diag, normin, n, a, ia, ja, desca, x, ix, jx, descx,
scale, cnorm, work)
```

```
call pdlatrs(uplo, trans, diag, normin, n, a, ia, ja, desca, x, ix, jx, descx,
scale, cnorm, work)
```

```
call pclatrs(uplo, trans, diag, normin, n, a, ia, ja, desca, x, ix, jx, descx,
scale, cnorm, work)
```

```
call pzlatrs(uplo, trans, diag, normin, n, a, ia, ja, desca, x, ix, jx, descx,
scale, cnorm, work)
```

Description

This routine solves a triangular system of equations $Ax = sb$, $A^T x = sb$ or $A^H x = sb$, where s is a scale factor set to prevent overflow. The description of the routine will be extended in the future releases.

Input Parameters

`uplo` CHARACTER*1.

	Specifies whether the matrix A is upper or lower triangular. = 'U': Upper triangular = 'L': Lower triangular
<i>trans</i>	CHARACTER*1. Specifies the operation applied to Ax . = 'N': Solve $Ax = s*b$ (no transpose) = 'T': Solve $A^T x = s*b$ (transpose) = 'C': Solve $A^H x = s*b$ (conjugate transpose), where s - is a scale factor
<i>diag</i>	CHARACTER*1. Specifies whether or not the matrix A is unit triangular. = 'N': Non-unit triangular = 'U': Unit triangular
<i>normin</i>	CHARACTER*1. Specifies whether <i>cnorm</i> has been set or not. = 'Y': <i>cnorm</i> contains the column norms on entry; = 'N': <i>cnorm</i> is not set on entry. On exit, the norms will be computed and stored in <i>cnorm</i> .
<i>n</i>	INTEGER. The order of the matrix A . $n \geq 0$
<i>a</i>	REAL for pslatrs/pclatrs DOUBLE PRECISION for pdlatrs/pzlatrs Array, DIMENSION (<i>lda</i> , <i>n</i>). Contains the triangular matrix A . If <i>uplo</i> = 'U', the leading n -by- n upper triangular part of the array <i>a</i> contains the upper triangular matrix, and the strictly lower triangular part of <i>a</i> is not referenced. If <i>uplo</i> = 'L', the leading n -by- n lower triangular part of the array <i>a</i> contains the lower triangular matrix, and the strictly upper triangular part of <i>a</i> is not referenced. If <i>diag</i> = 'U', the diagonal elements of <i>a</i> are also not referenced and are assumed to be 1.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix A , respectively.

<i>desca</i>	(global and local) INTEGER array, DIMENSION (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>x</i>	REAL for pslatrs/pclatrs DOUBLE PRECISION for pdlatrs/pzlatrs Array, DIMENSION (<i>n</i>). On entry, the right hand side <i>b</i> of the triangular system.
<i>ix</i>	(global) INTEGER. The row index in the global array <i>x</i> indicating the first row of sub(<i>x</i>).
<i>jx</i>	(global) INTEGER. The column index in the global array <i>x</i> indicating the first column of sub(<i>x</i>).
<i>descx</i>	(global and local) INTEGER. Array, DIMENSION (<i>dlen_</i>). The array descriptor for the distributed matrix <i>x</i> .
<i>cnorm</i>	REAL for pslatrs/pclatrs DOUBLE PRECISION for pdlatrs/pzlatrs. Array, DIMENSION (<i>n</i>). If <i>normin</i> = 'Y', <i>cnorm</i> is an input argument and <i>cnorm</i> (<i>j</i>) contains the norm of the off-diagonal part of the <i>j</i> -th column of <i>A</i> . If <i>trans</i> = 'N', <i>cnorm</i> (<i>j</i>) must be greater than or equal to the infinity-norm, and if <i>trans</i> = 'T' or 'C', <i>cnorm</i> (<i>j</i>) must be greater than or equal to the 1-norm.
<i>work</i>	(local). REAL for pslatrs DOUBLE PRECISION for pdlatrs COMPLEX for pclatrs COMPLEX*16 for pzlatrs. Temporary workspace.

Output Parameters

<i>x</i>	On exit, <i>x</i> is overwritten by the solution vector <i>x</i> .
<i>scale</i>	REAL for pslatrs/pclatrs DOUBLE PRECISION for pdlatrs/pzlatrs. Array, DIMENSION (<i>lda</i> , <i>n</i>). The scaling factor <i>s</i> for the triangular system as described above.

If $scale = 0$, the matrix A is singular or badly scaled, and the vector x is an exact or approximate solution to $Ax = 0$.

cnorm

If $normin = 'N'$, *cnorm* is an output argument and *cnorm*(j) returns the 1-norm of the off-diagonal part of the j -th column of A .

p?latrz

Reduces an upper trapezoidal matrix to upper triangular form by means of orthogonal/unitary transformations.

Syntax

```
call pslatz(m, n, l, a, ia, ja, desca, tau, work)
call pdlatrz(m, n, l, a, ia, ja, desca, tau, work)
call pclatz(m, n, l, a, ia, ja, desca, tau, work)
call pzlatrz(m, n, l, a, ia, ja, desca, tau, work)
```

Description

This routine reduces the m -by- n ($m \leq n$) real/complex upper trapezoidal matrix $sub(A) = [A(ia:ia+m-1, ja:ja+m-1) \ A(ia:ia+m-1, ja+n-l:ja+n-1)]$ to upper triangular form by means of orthogonal/unitary transformations.

The upper trapezoidal matrix $sub(A)$ is factored as

$$sub(A) = \begin{pmatrix} R & 0 \end{pmatrix} * Z,$$

where Z is an n -by- n orthogonal/unitary matrix and R is an m -by- m upper triangular matrix.

Input Parameters

m (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix $sub(A)$. $m \geq 0$.

n (global) INTEGER.

	The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$. $n \geq 0$.
<i>l</i>	(global) INTEGER. The number of columns of the distributed submatrix $\text{sub}(A)$ containing the meaningful part of the Householder reflectors. $l > 0$.
<i>a</i>	(local) REAL for pslatz DOUBLE PRECISION for pdlatrz COMPLEX for pclatz COMPLEX*16 for pzlatrz. Pointer into the local memory to an array of $\text{DIMENSION}(lld\_a, LOCc(ja+n-1))$. On entry, the local pieces of the m-by-n distributed matrix $\text{sub}(A)$, which is to be factored.
<i>ia</i>	(global) INTEGER. The row index in the global array <i>A</i> indicating the first row of $\text{sub}(A)$.
<i>ja</i>	(global) INTEGER. The column index in the global array <i>A</i> indicating the first column of $\text{sub}(A)$.
<i>desca</i>	(global and local) INTEGER array of $\text{DIMENSION}(dlen\_)$. The array descriptor for the distributed matrix <i>A</i>.
<i>work</i>	(local) REAL for pslatz DOUBLE PRECISION for pdlatrz COMPLEX for pclatz COMPLEX*16 for pzlatrz. Workspace array, $\text{DIMENSION}(lwork)$. $lwork \geq nq0 + \max(1, mp0)$, where $iroff = \text{mod}(ia-1, mb\_a)$, $icoff = \text{mod}(ja-1, nb\_a)$, $iarow = \text{indxg2p}(ia, mb\_a, myrow, rsrc\_a, nprow)$, $iacol = \text{indxg2p}(ja, nb\_a, mycol, csrc\_a, npcot)$, $mp0 = \text{numroc}(m+iroff, mb\_a, myrow, iarow, nprow)$,

`nq0 = numroc(n+icoff, nb_a, mycol, iacol, npcol)`,
`numroc`, `indxg2p`, and `numroc` are ScaLAPACK tool
functions; `myrow`, `mycol`, `nprow`, and `npcol` can be
determined by calling the subroutine `blacs_gridinfo`.

Output Parameters

<i>a</i>	On exit, the leading m -by- m upper triangular part of <code>sub(A)</code> contains the upper triangular matrix R , and elements $n-l+1$ to n of the first m rows of <code>sub(A)</code> , with the array <i>tau</i> , represent the orthogonal/unitary matrix Z as a product of m elementary reflectors.
<i>tau</i>	(local) REAL for <code>pslatz</code> DOUBLE PRECISION for <code>pdlatz</code> COMPLEX for <code>pclatz</code> COMPLEX*16 for <code>pzlatrz</code> . Array, <code>DIMENSION (LOCr(ja+m-1))</code> . This array contains the scalar factors of the elementary reflectors. <i>tau</i> is tied to the distributed matrix A .

Application Notes

The factorization is obtained by Householder's method. The k -th transformation matrix, $Z(k)$, which is used (or, in case of complex routines, whose conjugate transpose is used) to introduce zeros into the $(m - k + 1)$ -th row of `sub(A)`, is given in the form

$$Z(k) = \begin{bmatrix} I & 0 \\ 0 & T(k) \end{bmatrix},$$

where

$$T(k) = I - \tau u(k) u(k)^H, \quad u(k) = \begin{bmatrix} 1 \\ 0 \\ z(k) \end{bmatrix}$$

τ is a scalar and $z(k)$ is an $(n-m)$ -element vector. τ and $z(k)$ are chosen to annihilate the elements of the k -th row of $\text{sub}(A)$. The scalar τ is returned in the k -th element of τ and the vector $u(k)$ in the k -th row of $\text{sub}(A)$, such that the elements of $z(k)$ are in $a(k, m+1), \dots, a(k, n)$. The elements of R are returned in the upper triangular part of $\text{sub}(A)$.

Z is given by

$$Z = Z(1)Z(2)\dots Z(m).$$

p?lauu2

Computes the product $U*U'$ or $L'*L$, where U and L are upper or lower triangular matrices (local unblocked algorithm).

Syntax

```
call pslauu2(uplo, n, a, ia, ja, desca)
call pdlauu2(uplo, n, a, ia, ja, desca)
call pclauu2(uplo, n, a, ia, ja, desca)
call pzlauu2(uplo, n, a, ia, ja, desca)
```

Description

This routine computes the product $U*U'$ or $L'*L$, where the triangular factor U or L is stored in the upper or lower triangular part of the distributed matrix

$\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$.

If $\text{uplo} = 'U'$ or $'u'$, then the upper triangle of the result is stored, overwriting the factor U in $\text{sub}(A)$.

If $\text{uplo} = 'L'$ or $'l'$, then the lower triangle of the result is stored, overwriting the factor L in $\text{sub}(A)$.

This is the unblocked form of the algorithm, calling [BLAS Level 2 Routines](#). No communication is performed by this routine, the matrix to operate on should be strictly local to one process.

Input Parameters

uplo (global) CHARACTER*1.

	Specifies whether the triangular factor stored in the <i>matrix</i> sub(<i>A</i>) is upper or lower triangular: = <i>U</i> : upper triangular = <i>L</i> : lower triangular.
<i>n</i>	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the triangular factor <i>U</i> or <i>L</i> . $n \geq 0$.
<i>a</i>	(local) REAL for <i>pslauu2</i> DOUBLE PRECISION for <i>pdlauu2</i> COMPLEX for <i>pclauu2</i> COMPLEX*16 for <i>pzlauu2</i> . Pointer into the local memory to an array of DIMENSION (<i>lld_a</i> , <i>LOCc(ja+n-1)</i>). On entry, the local pieces of the triangular factor <i>U</i> or <i>L</i> .
<i>ia</i>	(global) INTEGER. The row index in the global array <i>A</i> indicating the first row of sub(<i>A</i>).
<i>ja</i>	(global) INTEGER. The column index in the global array <i>A</i> indicating the first column of sub(<i>A</i>).
<i>desca</i>	(global and local) INTEGER array of DIMENSION (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .

Output Parameters

<i>a</i>	(local) On exit, if <i>uplo</i> = ' <i>U</i> ', the upper triangle of the distributed matrix sub(<i>A</i>) is overwritten with the upper triangle of the product $U*U'$; if <i>uplo</i> = ' <i>L</i> ', the lower triangle of sub(<i>A</i>) is overwritten with the lower triangle of the product $L*L$.

p?lauum

Computes the product $U*U'$ or $L'*L$, where U and L are upper or lower triangular matrices.

Syntax

```
call pslauum(uplo, n, a, ia, ja, desca)
call pdlauum(uplo, n, a, ia, ja, desca)
call pclauum(uplo, n, a, ia, ja, desca)
call pzlauum(uplo, n, a, ia, ja, desca)
```

Description

This routine computes the product $U*U'$ or $L'*L$, where the triangular factor U or L is stored in the upper or lower triangular part of the matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$.

If $uplo = 'U'$ or $'u'$, then the upper triangle of the result is stored, overwriting the factor U in $\text{sub}(A)$. If $uplo = 'L'$ or $'l'$, then the lower triangle of the result is stored, overwriting the factor L in $\text{sub}(A)$.

This is the blocked form of the algorithm, calling Level 3 PBLAS.

Input Parameters

$uplo$	(global) CHARACTER*1. Specifies whether the triangular factor stored in the matrix $\text{sub}(A)$ is upper or lower triangular: = 'U': upper triangular = 'L': lower triangular.
n	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the triangular factor U or L . $n \geq 0$.
a	(local) REAL for pslauum DOUBLE PRECISION for pdlauum COMPLEX for pclauum COMPLEX*16 for pzlauum.

Pointer into the local memory to an array of `DIMENSION(lld_a, LOCC(ja+n-1))`. On entry, the local pieces of the triangular factor U or L .

ia (global) INTEGER.
The row index in the global array A indicating the first row of `sub(A)`.

ja (global) INTEGER.
The column index in the global array A indicating the first column of `sub(A)`.

desca (global and local) INTEGER array of `DIMENSION(dlen_)`.
The array descriptor for the distributed matrix A .

Output Parameters

a (local)
On exit, if `uplo = 'U'`, the upper triangle of the distributed matrix `sub(A)` is overwritten with the upper triangle of the product $U*U'$; if `uplo = 'L'`, the lower triangle of `sub(A)` is overwritten with the lower triangle of the product $L'*L$.

p?lawil

Forms the Wilkinson transform.

Syntax

```
call pslawil(ii, jj, m, a, desca, h44, h33, h43h34, v)
call pdlawil(ii, jj, m, a, desca, h44, h33, h43h34, v)
```

Description

This routine gets the transform given by $h44$, $h33$, and $h43h34$ into v starting at row m .

Input Parameters

ii (global) INTEGER.
Row owner of $h(m+2, m+2)$.

jj (global) INTEGER.
Column owner of $h(m+2, m+2)$.

m (global) INTEGER.
On entry, the location from where the transform starts (row *m*). Unchanged on exit.

a (global)
REAL for pslawil
DOUBLE PRECISION for pdlawil
Array, DIMENSION (*desca*(*lld_*),*).
On entry, the Hessenberg matrix. Unchanged on exit.

desca (global and local) INTEGER
Array of DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*. Unchanged on exit.

(global)
REAL for pslawil
h43h34 DOUBLE PRECISION for pdlawil
These three values are for the double shift *QR* iteration.
Unchanged on exit.

Output Parameters

v (global)
REAL for pslawil
DOUBLE PRECISION for pdlawil
Array of size 3 that contains the transform on output.

p?org2l/p?ung2l

Generates all or part of the orthogonal/unitary matrix Q from a QL factorization determined by p?geqlf (unblocked algorithm).

Syntax

```
call psorg2l(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorg2l(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pcung2l(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzung2l(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine `p?org2l/p?ung2l` generates an m -by- n real/complex distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal columns, which is defined as the last n columns of a product of k elementary reflectors of order m :

$Q = H(k) * \dots * H(2) * H(1)$ as returned by `p?geqlf`.

Input Parameters

m	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix Q . $m \geq 0$.
n	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix Q . $m \geq n \geq 0$.
k	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . $n \geq k \geq 0$.
a	REAL for <code>psorg2l</code> DOUBLE PRECISION for <code>pdorg2l</code> COMPLEX for <code>pcung2l</code> COMPLEX*16 for <code>pzung2l</code> . Pointer into the local memory to an array, DIMENSION (<code>lld_a</code> , <code>LOCc(ja+n-1)</code>). On entry, the j -th column must contain the vector that defines the elementary reflector $H(j)$, $ja+n-k \leq j \leq ja+n-1$, as returned by <code>p?geqlf</code> in the k columns of its <i>distributed matrix</i> argument $A(ia:*, ja+n-k:ja+n-1)$.
ia	(global) INTEGER. The row index in the global array A indicating the first row of sub(A).
ja	(global) INTEGER. The column index in the global array A indicating the first column of sub(A).

desca (global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix *A*.

tau (local)
REAL for *psorg2l*
DOUBLE PRECISION for *pdorg2l*
COMPLEX for *pcung2l*
COMPLEX*16 for *pzung2l*.
Array, DIMENSION *LOCc(ja+n-1)*.
This array contains the scalar factor *tau(j)* of the elementary reflector $H(j)$, as returned by [p?geqlf](#).

work (local)
REAL for *psorg2l*
DOUBLE PRECISION for *pdorg2l*
COMPLEX for *pcung2l*
COMPLEX*16 for *pzung2l*.
Workspace array, DIMENSION (*lwork*).

lwork (local or global) INTEGER.
The dimension of the array *work*.

lwork is local input and must be at least $lwork \geq mpa0 + \max(1, nqa0)$, where
 $iroffa = \text{mod}(ia-1, mb\_a),$
 $icoffa = \text{mod}(ja-1, nb\_a),$
 $iarow = \text{indxg2p}(ia, mb\_a, myrow, rsrc\_a, nprow),$
 $iacol = \text{indxg2p}(ja, nb\_a, mycol, csrc\_a, npcot),$
 $mpa0 = \text{numroc}(m+iroffa, mb\_a, myrow, iarow, nprow),$
 $nqa0 = \text{numroc}(n+icoffa, nb\_a, mycol, iacol, npcot).$
indxg2p and *numroc* are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine [blacs_gridinfo](#).
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

<i>a</i>	On exit, this array contains the local pieces of the m -by- n distributed matrix Q .
<i>work</i>	On exit, <i>work</i> (1) returns the minimal and optimal <i>lwork</i> .
<i>info</i>	(local) INTEGER. = 0: successful exit < 0: if the i -th argument is an array and the j -entry had an illegal value, then <i>info</i> = - ($i*100 + j$), if the i -th argument is a scalar and had an illegal value, then <i>info</i> = - i .

p?org2r/p?ung2r

Generates all or part of the orthogonal/unitary matrix Q from a QR factorization determined by p?geqrf (unblocked algorithm).

Syntax

```
call psorg2r(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorg2r(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pcung2r(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzung2r(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine p?org2r/p?ung2r generates an m -by- n real/complex matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal columns, which is defined as the first n columns of a product of k elementary reflectors of order m :

$$Q = H(1) * H(2) * \dots * H(k)$$

as returned by p?geqrf.

Input Parameters

m (global) INTEGER.

	The number of rows to be operated on, that is, the number of rows of the distributed submatrix Q. $m \geq 0$.
<i>n</i>	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix Q. $m \geq n \geq 0$.
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q. $n \geq k \geq 0$.
<i>a</i>	REAL for psorg2r DOUBLE PRECISION for pdorg2r COMPLEX for pcung2r COMPLEX*16 for pzung2r. Pointer into the local memory to an array, DIMENSION (<i>lld_a</i>, <i>LOCc</i>(<i>ja</i>+<i>n</i>-1)). On entry, the <i>j</i>-th column must contain the vector that defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by p?geqrf in the <i>k</i> columns of its <i>distributed matrix</i> argument $A(ia:*, ja:ja+k-1)$.
<i>ia</i>	(global) INTEGER. The row index in the global array <i>A</i> indicating the first row of sub(<i>A</i>).
<i>ja</i>	(global) INTEGER. The column index in the global array <i>A</i> indicating the first column of sub(<i>A</i>).
<i>desca</i>	(global and local) INTEGER array of DIMENSION (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i>.
<i>tau</i>	(local) REAL for psorg2r DOUBLE PRECISION for pdorg2r COMPLEX for pcung2r COMPLEX*16 for pzung2r. Array, DIMENSION <i>LOCc</i>(<i>ja</i>+<i>k</i>-1).

This array contains the scalar factor $\tau(j)$ of the elementary reflector $H(j)$, as returned by [p?geqrf](#). This array is tied to the distributed matrix A .

work (local)
 REAL for psorg2r
 DOUBLE PRECISION for pdorg2r
 COMPLEX for pcung2r
 COMPLEX*16 for pzung2r.
 Workspace array, DIMENSION (*lwork*).

lwork (local or global) INTEGER.
 The dimension of the array *work*.
lwork is local input and must be at least $lwork \geq mpa0 + \max(1, nqa0)$,
 where
 $iroffa = \text{mod}(ia-1, mb\_a)$, $icoffa = \text{mod}(ja-1, nb\_a)$,
 $iarow = \text{indxg2p}(ia, mb\_a, myrow, rsrc\_a, nprow)$,
 $iacol = \text{indxg2p}(ja, nb\_a, mycol, csrc\_a, npcot)$,
 $mpa0 = \text{numroc}(m+iroffa, mb\_a, myrow, iarow, nprow)$,
 $nqa0 = \text{numroc}(n+icoffa, nb\_a, mycol, iacol, npcot)$.
 indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine `blacs_gridinfo`.
 If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

a On exit, this array contains the local pieces of the m -by- n distributed matrix Q .

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local) INTEGER.

= 0: successful exit
 < 0: if the i -th argument is an array and the j -entry had an illegal value,
 then $info = - (i*100+j)$,
 if the i -th argument is a scalar and had an illegal value,
 then $info = -i$.

p?orgl2/p?ungl2

Generates all or part of the orthogonal/unitary matrix Q from an LQ factorization determined by p?gelqf (unblocked algorithm).

Syntax

```
call psorgl2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorgl2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pcungl2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzungl2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine p?orgl2/p?ungl2 generates a m -by- n real/complex matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the first m rows of a product of k elementary reflectors of order n

$Q = H(k) * \dots * H(2) * H(1)$ (for real flavors),

$Q = (H(k))^H * \dots * (H(2))^H * (H(1))^H$ (for complex flavors) as returned by p?gelqf.

Input Parameters

m (global) INTEGER.
 The number of rows to be operated on, that is, the number of rows of the distributed submatrix Q . $m \geq 0$.

n (global) INTEGER.
 The number of columns to be operated on, that is, the number of columns of the distributed submatrix Q . $n \geq m \geq 0$.

<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q. $m \geq k \geq 0$.
<i>a</i>	REAL for psorgl2 DOUBLE PRECISION for pdorgl2 COMPLEX for pcungl2 COMPLEX*16 for pzungl2. Pointer into the local memory to an array, DIMENSION (<i>lld_a</i>, <i>LOCc(ja+n-1)</i>). On entry, the <i>i</i>-th row must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by p?gelqf in the <i>k</i> rows of its <i>distributed matrix</i> argument $A(ia:ia+k-1, ja:*)$.
<i>ia</i>	(global) INTEGER. The row index in the global array <i>A</i> indicating the first row of sub(<i>A</i>).
<i>ja</i>	(global) INTEGER. The column index in the global array <i>A</i> indicating the first column of sub(<i>A</i>).
<i>desca</i>	(global and local) INTEGER array of DIMENSION (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i>.
<i>tau</i>	(local) REAL for psorgl2 DOUBLE PRECISION for pdorgl2 COMPLEX for pcungl2 COMPLEX*16 for pzungl2. Array, DIMENSION <i>LOCr(ja+k-1)</i>. This array contains the scalar factors <i>tau(i)</i> of the elementary reflectors $H(i)$, as returned by p?gelqf. This array is tied to the distributed matrix <i>A</i>.
<i>WORK</i>	(local) REAL for psorgl2 DOUBLE PRECISION for pdorgl2 COMPLEX for pcungl2 COMPLEX*16 for pzungl2. Workspace array, DIMENSION (<i>lwork</i>).

lwork

(local or global) INTEGER.

The dimension of the array *work*.

lwork is local input and must be at least $lwork \geq nqa0 + \max(1, mpa0)$, where

$$iroffa = \text{mod}(ia-1, mb\_a),$$

$$icoffa = \text{mod}(ja-1, nb\_a),$$

$$iarow = \text{indxg2p}(ia, mb\_a, myrow, rsrc\_a, nprow),$$

$$iacol = \text{indxg2p}(ja, nb\_a, mycol, csrc\_a, npcol),$$

$$mpa0 = \text{numroc}(m+iroffa, mb\_a, myrow, iarow, nprow),$$

$$nqa0 = \text{numroc}(n+icoffa, nb\_a, mycol, iacol, npcol).$$

indxg2p and *numroc* are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine *blacs_gridinfo*.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by *pxerbla*.

Output Parameters

a

On exit, this array contains the local pieces of the *m*-by-*n* distributed matrix *Q*.

work

On exit, *work*(1) returns the minimal and optimal *lwork*.

info

(local) INTEGER.

= 0: successful exit

< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value,

then *info* = - (*i**100+*j*),

if the *i*-th argument is a scalar and had an illegal value,

then *info* = -*i*.

p?orgr2/p?ungr2

Generates all or part of the orthogonal/unitary matrix Q from an RQ factorization determined by p?gerqf (unblocked algorithm).

Syntax

```
call psorgr2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorgr2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pcungr2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzungr2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine p?orgr2/p?ungr2 generates an m -by- n real/complex matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the last m rows of a product of k elementary reflectors of order n

$Q = H(1) * H(2) * \dots * H(k)$ (for real flavors);

$Q = (H(1))^H * (H(2))^H \dots * (H(k))^H$ (for complex flavors) as returned by p?gerqf.

Input Parameters

m	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix Q . $m \geq 0$.
n	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix Q . $n \geq m \geq 0$.
k	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . $m \geq k \geq 0$.
a	REAL for psorgr2 DOUBLE PRECISION for pdorgr2 COMPLEX for pcungr2

COMPLEX*16 for pzungr2.
 Pointer into the local memory to an array,
 DIMENSION (*lld_a*, *LOCc*(*ja+n-1*)).
 On entry, the *i*-th row must contain the vector that defines
 the elementary reflector $H(i)$, $ia+m-k \leq i \leq ia+m-1$,
 as returned by [p?gerqf](#) in the *k* rows of its *distributed*
matrix argument $A(ia+m-k:ia+m-1, ja:*)$.

ia (global) INTEGER.
 The row index in the global array *A* indicating the first row
 of sub(*A*).

ja (global) INTEGER.
 The column index in the global array *A* indicating the first
 column of sub(*A*).

desca (global and local) INTEGER array of DIMENSION (*dlen_*).
 The array descriptor for the distributed matrix *A*.

tau (local)
 REAL for psorgr2
 DOUBLE PRECISION for pdorgr2
 COMPLEX for pcungr2
 COMPLEX*16 for pzungr2.
 Array, DIMENSION *LOCr*(*ja+m-1*). This array contains the
 scalar factors *tau*(*i*) of the elementary reflectors $H(i)$, as
 returned by [p?gerqf](#). This array is tied to the distributed
 matrix *A*.

work (local)
 REAL for psorgr2
 DOUBLE PRECISION for pdorgr2
 COMPLEX for pcungr2
 COMPLEX*16 for pzungr2.
 Workspace array, DIMENSION (*lwork*).

lwork (local or global) INTEGER.
 The dimension of the array *work*.

lwork is local input and must be at least $lwork \geq nqa0 +$
 $\max(1, mpa0)$, where $iroffa = \text{mod}(ia-1, mb\_a)$,
 $icoffa = \text{mod}(ja-1, nb\_a)$,

```

iarow = indxg2p( ia, mb_a, myrow, rsrc_a, nprow
),
iacol = indxg2p( ja, nb_a, mycol, csrc_a, npcol
),
mpa0 = numroc( m+iroffa, mb_a, myrow, iarow,
nprow ),
nqa0 = numroc( n+icoffa, nb_a, mycol, iacol,
npcol ).

```

`indxg2p` and `numroc` are ScaLAPACK tool functions; `myrow`, `mycol`, `nprow`, and `npcol` can be determined by calling the subroutine `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>a</code>	On exit, this array contains the local pieces of the m -by- n distributed matrix Q .
<code>work</code>	On exit, <code>work(1)</code> returns the minimal and optimal <code>lwork</code> .
<code>info</code>	(local) INTEGER. = 0: successful exit < 0: if the i -th argument is an array and the j -entry had an illegal value, then <code>info = - (i*100+j)</code> , if the i -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

p?orm2l/p?unm2l

Multiplies a general matrix by the orthogonal/unitary matrix from a QL factorization determined by p?geqlf (unblocked algorithm).

Syntax

```
call psorm2l(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)

call pdorm2l(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)

call pcunm2l(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)

call pzunm2l(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

Description

The routine p?orm2l/p?unm2l overwrites the general real/complex m -by- n distributed matrix sub $(C)=C(ic:ic+m-1,jc:jc+n-1)$ with

$Q*\text{sub}(C)$ if $side = 'L'$ and $trans = 'N'$, or

$Q^T*\text{sub}(C) / Q^H*\text{sub}(C)$ if $side = 'L'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors), or

$\text{sub}(C)*Q$ if $side = 'R'$ and $trans = 'N'$, or

$\text{sub}(C)*Q^T / \text{sub}(C)*Q^H$ if $side = 'R'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors).

where Q is a real orthogonal or complex unitary distributed matrix defined as the product of k elementary reflectors

$Q = H(k)*\dots*H(2)*H(1)$ as returned by p?geqlf. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

side (global) CHARACTER.
 = 'L': apply Q or Q^T for real flavors (Q^H for complex flavors) from the left,

	= 'R': apply Q or Q^T for real flavors (Q^H for complex flavors) from the right.
<i>trans</i>	(global) CHARACTER. = 'N': apply Q (no transpose) = 'T': apply Q^T (transpose, for real flavors) = 'C': apply Q^H (conjugate transpose, for complex flavors)
<i>m</i>	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(C)$. $m \geq 0$.
<i>n</i>	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(C)$. $n \geq 0$.
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q. If <i>side</i> = 'L', $m \geq k \geq 0$; if <i>side</i> = 'R', $n \geq k \geq 0$.
<i>a</i>	(local) REAL for psorm2l DOUBLE PRECISION for pdorm2l COMPLEX for pcunm2l COMPLEX*16 for pzunm2l. Pointer into the local memory to an array, DIMENSION(<i>lld_a</i>, <i>LOCc</i>(<i>ja+k-1</i>)). On entry, the <i>j</i>-th row must contain the vector that defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by p?geqlf in the <i>k</i> columns of its distributed matrix argument $A(ia:*, ja:ja+k-1)$. The argument $A(ia:*, ja:ja+k-1)$ is modified by the routine but restored on exit. If <i>side</i> = 'L', $\text{lld\_a} \geq \max(1, \text{LOCr}(ia+m-1))$, if <i>side</i> = 'R', $\text{lld\_a} \geq \max(1, \text{LOCr}(ia+n-1))$.
<i>ia</i>	(global) INTEGER.

	The row index in the global array <i>A</i> indicating the first row of <i>sub(A)</i> .
<i>ja</i>	(global) INTEGER. The column index in the global array <i>A</i> indicating the first column of <i>sub(A)</i> .
<i>desca</i>	(global and local) INTEGER array of DIMENSION (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) REAL for <i>psorm21</i> DOUBLE PRECISION for <i>pdorm21</i> COMPLEX for <i>pcunm21</i> COMPLEX*16 for <i>pzunm21</i> . Array, DIMENSION <i>LOCc(ja+n-1)</i> . This array contains the scalar factor <i>tau(j)</i> of the elementary reflector <i>H(j)</i> , as returned by <i>p?geqlf</i> . This array is tied to the distributed matrix <i>A</i> .
<i>c</i>	(local) REAL for <i>psorm21</i> DOUBLE PRECISION for <i>pdorm21</i> COMPLEX for <i>pcunm21</i> COMPLEX*16 for <i>pzunm21</i> . Pointer into the local memory to an array, DIMENSION (<i>lld_c</i> , <i>LOCc(jc+n-1)</i>). On entry, the local pieces of the distributed matrix <i>sub(c)</i> .
<i>ic</i>	(global) INTEGER. The row index in the global array <i>c</i> indicating the first row of <i>sub(c)</i> .
<i>jc</i>	(global) INTEGER. The column index in the global array <i>c</i> indicating the first column of <i>sub(c)</i> .
<i>descc</i>	(global and local) INTEGER array of DIMENSION (<i>dlen_</i>). The array descriptor for the distributed matrix <i>c</i> .
<i>work</i>	(local) REAL for <i>psorm21</i> DOUBLE PRECISION for <i>pdorm21</i> COMPLEX for <i>pcunm21</i>

lwork COMPLEX*16 for pzunm2l.
Workspace array, DIMENSION (*lwork*).
On exit, *work*(1) returns the minimal and optimal *lwork*.
(local or global) INTEGER.
The dimension of the array *work*.
lwork is local input and must be at least
if *side* = 'L', $lwork \geq mpc0 + \max(1, nqc0)$,
if *side* = 'R', $lwork \geq nqc0 + \max(\max(1, mpc0),$
 $\text{numroc}(\text{numroc}(n+icoffc, nb\_a, 0, 0, npc0),$
 $nb\_a, 0, 0, lcmq))$,
where
 $lcmq = lcm/npcol$,
 $lcm = iclm(nprow, npc0)$,
 $iroffc = \text{mod}(ic-1, mb\_c)$,
 $icoffc = \text{mod}(jc-1, nb\_c)$,
 $icrow = \text{indxg2p}(ic, mb\_c, myrow, rsrc\_c, nprow)$,
 $iccol = \text{indxg2p}(jc, nb\_c, mycol, csrc\_c, npc0)$,
 $Mqc0 = \text{numroc}(m+icoffc, nb\_c, mycol, icrow,$
 $nprow)$,
 $Npc0 = \text{numroc}(n+iroffc, mb\_c, myrow, iccol,$
 $npc0)$,
 $iclm, \text{indxg2p}$, and numroc are ScaLAPACK tool functions;
 $myrow, mycol, nprow$, and $npc0$ can be determined by
calling the subroutine `blacs_gridinfo`.
If *lwork* = -1, then *lwork* is global input and a workspace
query is assumed; the routine only calculates the minimum
and optimal size for all work arrays. Each of these values
is returned in the first entry of the corresponding work array,
and no error message is issued by `pxerbla`.

Output Parameters

c On exit, *c* is overwritten by $Q * \text{sub}(C)$, or $Q^T * \text{sub}(C) /$
 $Q^H * \text{sub}(C)$, or $\text{sub}(C) * Q$, or $\text{sub}(C) * Q^T / \text{sub}(C) * Q^H$
work On exit, *work*(1) returns the minimal and optimal *lwork*.
info (local) INTEGER.
= 0: successful exit

< 0: if the i -th argument is an array and the j -entry had an illegal value,
 then $info = - (i*100+j)$,
 if the i -th argument is a scalar and had an illegal value,
 then $info = -i$.



NOTE. The distributed submatrices $A(ia:*, ja:*)$ and $C(ic:ic+m-1, jc:jc+n-1)$ must verify some alignment properties, namely the following expressions should be true:

If $side = 'L'$, ($mb\_a.eq.mb\_c$.AND. $iroffa.eq.iroffc$.AND. $iarow.eq.icrow$)

If $side = 'R'$, ($mb\_a.eq.nb\_c$.AND. $iroffa.eq.iroffc$).

p?orm2r/p?unm2r

Multiplies a general matrix by the orthogonal/unitary matrix from a QR factorization determined by p?geqrf (unblocked algorithm).

Syntax

```
call psorm2r(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

```
call pdorm2r(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

```
call pcunm2r(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

```
call pzunm2r(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

Description

The routine p?orm2r/p?unm2r overwrites the general real/complex m -by- n distributed matrix sub $(C)=C(ic:ic+m-1, jc:jc+n-1)$ with

$Q*\text{sub}(C)$ if $side = 'L'$ and $trans = 'N'$, or

$Q^T*\text{sub}(C) / Q^H*\text{sub}(C)$ if $side = 'L'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors), or

$\text{sub}(C) * Q$ if $\text{side} = 'R'$ and $\text{trans} = 'N'$, or

$\text{sub}(C) * Q^T / \text{sub}(C) * Q^H$ if $\text{side} = 'R'$ and $\text{trans} = 'T'$ (for real flavors) or $\text{trans} = 'C'$ (for complex flavors).

where Q is a real orthogonal or complex unitary matrix defined as the product of k elementary reflectors

$Q = H(k) * \dots * H(2) * H(1)$ as returned by [p?geqrf](#). Q is of order m if $\text{side} = 'L'$ and of order n if $\text{side} = 'R'$.

Input Parameters

<i>side</i>	(global) CHARACTER. = 'L': apply Q or Q^T for real flavors (Q^H for complex flavors) from the left, = 'R': apply Q or Q^T for real flavors (Q^H for complex flavors) from the right.
<i>trans</i>	(global) CHARACTER. = 'N': apply Q (no transpose) = 'T': apply Q^T (transpose, for real flavors) = 'C': apply Q^H (conjugate transpose, for complex flavors)
<i>m</i>	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(C)$. $m \geq 0$.
<i>n</i>	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(C)$. $n \geq 0$.
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q. If $\text{side} = 'L'$, $m \geq k \geq 0$; if $\text{side} = 'R'$, $n \geq k \geq 0$.
<i>a</i>	(local) REAL for psorm2r DOUBLE PRECISION for pdorm2r COMPLEX for pcunm2r

COMPLEX*16 for pzunm2r.
 Pointer into the local memory to an array,
 DIMENSION(*lld_a*, *LOCc*(*ja+k-1*)).
 On entry, the *j*-th column must contain the vector that
 defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$,
 as returned by `p?geqrf` in the *k* columns of its distributed
 matrix argument $A(ia:*, ja:ja+k-1)$. The argument
 $A(ia:*, ja:ja+k-1)$ is modified by the routine but restored
 on exit.

If *side* = 'L', $lld\_a \geq \max(1, LOCr(ia+m-1))$,
 if *side* = 'R', $lld\_a \geq \max(1, LOCr(ia+n-1))$.

ia (global) INTEGER.
 The row index in the global array *A* indicating the first row
 of sub(*A*).

ja (global) INTEGER.
 The column index in the global array *A* indicating the first
 column of sub(*A*).

desca (global and local) INTEGER array of DIMENSION (*dlen_*).
 The array descriptor for the distributed matrix *A*.

tau (local)
 REAL for psorm2r
 DOUBLE PRECISION for pdorm2r
 COMPLEX for pcunm2r
 COMPLEX*16 for pzunm2r.
 Array, DIMENSION *LOCc*(*ja+k-1*). This array contains the
 scalar factors *tau*(*j*) of the elementary reflector $H(j)$, as
 returned by `p?geqrf`. This array is tied to the distributed
 matrix *A*.

c (local)
 REAL for psorm2r
 DOUBLE PRECISION for pdorm2r
 COMPLEX for pcunm2r
 COMPLEX*16 for pzunm2r.
 Pointer into the local memory to an array,
 DIMENSION(*lld_c*, *LOCc*(*jc+n-1*)).
 On entry, the local pieces of the distributed matrix sub (*c*).

ic (global) INTEGER.
The row index in the global array *c* indicating the first row of sub(*c*).

jc (global) INTEGER.
The column index in the global array *c* indicating the first column of sub(*c*).

descc (global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix *c*.

work (local)
REAL for psorm2r
DOUBLE PRECISION for pdorm2r
COMPLEX for pcunm2r
COMPLEX*16 for pzunm2r.
Workspace array, DIMENSION (*lwork*).

lwork (local or global) INTEGER.
The dimension of the array *work*.
lwork is local input and must be at least
if *side* = 'L', $lwork \geq mpc0 + \max(1, nqc0)$,
if *side* = 'R', $lwork \geq nqc0 + \max(\max(1, mpc0),$
 $\text{numroc}(\text{numroc}(n+icoffc, nb\_a, 0, 0, npcol),$
 $nb\_a, 0, 0, lcmq)),$
where
 $lcmq = lcm/npcol,$
 $lcm = iclm(nprow, npcol),$
 $iroffc = \text{mod}(ic-1, mb\_c),$
 $icoffc = \text{mod}(jc-1, nb\_c),$
 $icrow = \text{indxg2p}(ic, mb\_c, myrow, rsrc\_c, nprow),$
 $iccol = \text{indxg2p}(jc, nb\_c, mycol, csrc\_c, npcol),$
 $Mqc0 = \text{numroc}(m+icoffc, nb\_c, mycol, icrow,$
 $nprow),$
 $Npc0 = \text{numroc}(n+iroffc, mb\_c, myrow, iccol,$
 $npcol),$
ilcm, *indxg2p* and *numroc* are ScaLAPACK tool functions;
myrow, *mycol*, *nprow*, and *npcol* can be determined by
calling the subroutine *blacs_gridinfo*.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<i>c</i>	On exit, <i>c</i> is overwritten by $Q * \text{sub}(C)$, or $Q^T * \text{sub}(C) / Q^H * \text{sub}(C)$, or $\text{sub}(C) * Q$, or $\text{sub}(C) * Q^T / \text{sub}(C) * Q^H$
<i>work</i>	On exit, <i>work</i> (1) returns the minimal and optimal <i>lwork</i> .
<i>info</i>	(local) INTEGER. = 0: successful exit < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = - (<i>i</i> *100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .



NOTE. The distributed submatrices $A(ia:*, ja:*)$ and $C(ic:ic+m-1, jc:jc+n-1)$ must verify some alignment properties, namely the following expressions should be true:

If *side* = 'L', (*mb_a*.eq.*mb_c* .AND. *iroffa*.eq.*iroffc* .AND. *iarow*.eq.*icrow*)

If *side* = 'R', (*mb_a*.eq.*nb_c* .AND. *iroffa*.eq.*iroffc*).

p?orml2/p?unml2

Multiplies a general matrix by the orthogonal/unitary matrix from an LQ factorization determined by p?gelqf (unblocked algorithm).

Syntax

```
call psorml2(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)

call pdorml2(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)

call pcunml2(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)

call pzunml2(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

Description

The routine p?orml2/p?unml2 overwrites the general real/complex m -by- n distributed matrix sub (C)=C(ic:ic+m-1, jc:jc+n-1) with

$Q \cdot \text{sub}(C)$ if $side = 'L'$ and $trans = 'N'$, or

$Q^T \cdot \text{sub}(C) / Q^H \cdot \text{sub}(C)$ if $side = 'L'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors), or

$\text{sub}(C) \cdot Q$ if $side = 'R'$ and $trans = 'N'$, or

$\text{sub}(C) \cdot Q^T / \text{sub}(C) \cdot Q^H$ if $side = 'R'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors).

where Q is a real orthogonal or complex unitary distributed matrix defined as the product of k elementary reflectors

$Q = H(k) \cdot \dots \cdot H(2) \cdot H(1)$ (for real flavors)

$Q = (H(k))^H \cdot \dots \cdot (H(2))^H \cdot (H(1))^H$ (for complex flavors)

as returned by p?gelqf . Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

side (global) CHARACTER.

= 'L': apply Q or Q^T for real flavors (Q^H for complex flavors)
 from the left,
 = 'R': apply Q or Q^T for real flavors (Q^H for complex flavors)
 from the right.

trans (global) CHARACTER.
 = 'N': apply Q (no transpose)
 = 'T': apply Q^T (transpose, for real flavors)
 = 'C': apply Q^H (conjugate transpose, for complex flavors)

m (global) INTEGER.
 The number of rows to be operated on, that is, the number
 of rows of the distributed submatrix $\text{sub}(C)$. $m \geq 0$.

n (global) INTEGER.
 The number of columns to be operated on, that is, the
 number of columns of the distributed submatrix $\text{sub}(C)$. n
 ≥ 0 .

k (global) INTEGER.
 The number of elementary reflectors whose product defines
 the matrix Q .
 If $\text{side} = \text{'L'}$, $m \geq k \geq 0$;
 if $\text{side} = \text{'R'}$, $n \geq k \geq 0$.

a (local)
 REAL for psorml2
 DOUBLE PRECISION for pdorml2
 COMPLEX for pcunml2
 COMPLEX*16 for pzunml2.
 Pointer into the local memory to an array, DIMENSION
 (lld_a , $\text{LOCc}(ja+m-1)$ if $\text{side}=\text{'L'}$,
 (lld_a , $\text{LOCc}(ja+n-1)$ if $\text{side}=\text{'R'}$,
 where $\text{lld\_a} \geq \max(1, \text{LOCr}(ia+k-1))$.
 On entry, the i -th row must contain the vector that defines
 the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as
 returned by [p?gelqf](#) in the k rows of its distributed matrix
 argument $A(ia:ia+k-1, ja:*)$. The argument
 $A(ia:ia+k-1, ja:*)$ is modified by the routine but
 restored on exit.

<i>ia</i>	(global) INTEGER. The row index in the global array <i>A</i> indicating the first row of sub(<i>A</i>).
<i>ja</i>	(global) INTEGER. The column index in the global array <i>A</i> indicating the first column of sub(<i>A</i>).
<i>desca</i>	(global and local) INTEGER array of DIMENSION (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i>.
<i>tau</i>	(local) REAL for psorml2 DOUBLE PRECISION for pdorml2 COMPLEX for pcunml2 COMPLEX*16 for pzunml2. Array, DIMENSION <i>LOCc(ia+k-1)</i>. This array contains the scalar factors <i>tau(i)</i> of the elementary reflector <i>H(i)</i>, as returned by p?gelqf. This array is tied to the distributed matrix <i>A</i>.
<i>c</i>	(local) REAL for psorml2 DOUBLE PRECISION for pdorml2 COMPLEX for pcunml2 COMPLEX*16 for pzunml2. Pointer into the local memory to an array, DIMENSION(<i>lld_c</i>, <i>LOCc(jc+n-1)</i>). On entry, the local pieces of the distributed matrix sub (<i>c</i>).
<i>ic</i>	(global) INTEGER. The row index in the global array <i>c</i> indicating the first row of sub(<i>c</i>).
<i>jc</i>	(global) INTEGER. The column index in the global array <i>c</i> indicating the first column of sub(<i>c</i>).
<i>descc</i>	(global and local) INTEGER array of DIMENSION (<i>dlen_</i>). The array descriptor for the distributed matrix <i>C</i>.
<i>work</i>	(local) REAL for psorml2 DOUBLE PRECISION for pdorml2

COMPLEX for pcunml2
 COMPLEX*16 for pzunml2.
 Workspace array, DIMENSION (*lwork*).
lwork (local or global) INTEGER.
 The dimension of the array *work*.
lwork is local input and must be at least
 if *side* = 'L', $lwork \geq mqc0 + \max(\max(1, npc0),$
 $numroc(numroc(m+icoffc, mb\_a, 0, 0, nprow),$
 $mb\_a, 0, 0, lcm)),$
 if *side* = 'R', $lwork \geq npc0 + \max(1, mqc0),$
 where
 $lcmp = lcm / nprow,$
 $lcm = iclm(nprow, npc0),$
 $iroffc = \text{mod}(ic-1, mb\_c),$
 $icoffc = \text{mod}(jc-1, nb\_c),$
 $icrow = \text{indxg2p}(ic, mb\_c, myrow, rsrc\_c, nprow),$
 $iccol = \text{indxg2p}(jc, nb\_c, mycol, csrc\_c, npc0),$
 $Mpc0 = numroc(m+icoffc, mb\_c, mycol, icrow,$
 $nprow),$
 $Nqc0 = numroc(n+iroffc, nb\_c, myrow, iccol,$
 $npc0),$
 $iclm, \text{indxg2p}$ and $numroc$ are ScaLAPACK tool functions;
 $myrow, mycol, nprow,$ and $npc0$ can be determined by
 calling the subroutine `blacs_gridinfo`.
 If *lwork* = -1, then *lwork* is global input and a workspace
 query is assumed; the routine only calculates the minimum
 and optimal size for all work arrays. Each of these values
 is returned in the first entry of the corresponding work array,
 and no error message is issued by `pxerbla`.

Output Parameters

c On exit, *c* is overwritten by $Q^* \text{sub}(C)$, or $Q^T \text{sub}(C) /$
 $Q^H \text{sub}(C)$, or $\text{sub}(C) * Q$, or $\text{sub}(C) * Q^T / \text{sub}(C) * Q^H$
work On exit, *work*(1) returns the minimal and optimal *lwork*.
info (local) INTEGER.
 = 0: successful exit

< 0: if the i -th argument is an array and the j -entry had an illegal value,
then $info = - (i*100+j)$,
if the i -th argument is a scalar and had an illegal value,
then $info = -i$.



NOTE. The distributed submatrices $A(ia:*, ja:*)$ and $C(ic:ic+m-1, jc:jc+n-1)$ must verify some alignment properties, namely the following expressions should be true:

If $side = 'L'$, $(nb\_a.eq.mb\_c .AND. icoffa.eq.iroffc)$

If $side = 'R'$, $(nb\_a.eq.nb\_c .AND. icoffa.eq.icoffc .AND. iacol.eq.iccol)$.

p?ormr2/p?unmr2

Multiplies a general matrix by the orthogonal/unitary matrix from an RQ factorization determined by p?gerqf (unblocked algorithm).

Syntax

```
call psormr2(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

```
call pdormr2(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

```
call pcunmr2(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

```
call pzunmr2(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

Description

The routine p?ormr2/p?unmr2 overwrites the general real/complex m -by- n distributed matrix sub $(C)=C(ic:ic+m-1, jc:jc+n-1)$ with

$Q^* \text{sub}(C)$ if $side = 'L'$ and $trans = 'N'$, or

$Q^T \text{sub}(C) / Q^H \text{sub}(C)$ if $side = 'L'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors), or

$\text{sub}(C) * Q$ if $\text{side} = 'R'$ and $\text{trans} = 'N'$, or

$\text{sub}(C) * Q^T / \text{sub}(C) * Q^H$ if $\text{side} = 'R'$ and $\text{trans} = 'T'$ (for real flavors) or $\text{trans} = 'C'$ (for complex flavors).

where Q is a real orthogonal or complex unitary distributed matrix defined as the product of k elementary reflectors

$Q = H(1) * H(2) * \dots * H(k)$ (for real flavors)

$Q = (H(1))^H * (H(2))^H * \dots * (H(k))^H$ (for complex flavors)

as returned by [p?gerqf](#). Q is of order m if $\text{side} = 'L'$ and of order n if $\text{side} = 'R'$.

Input Parameters

<i>side</i>	(global) CHARACTER. = 'L': apply Q or Q^T for real flavors (Q^H for complex flavors) from the left, = 'R': apply Q or Q^T for real flavors (Q^H for complex flavors) from the right.
<i>trans</i>	(global) CHARACTER. = 'N': apply Q (no transpose) = 'T': apply Q^T (transpose, for real flavors) = 'C': apply Q^H (conjugate transpose, for complex flavors)
<i>m</i>	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(C)$. $m \geq 0$.
<i>n</i>	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(C)$. $n \geq 0$.
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . If $\text{side} = 'L'$, $m \geq k \geq 0$; if $\text{side} = 'R'$, $n \geq k \geq 0$.
<i>a</i>	(local) REAL for psormr2

DOUBLE PRECISION for pdormr2
 COMPLEX for pcunmr2
 COMPLEX*16 for pzunmr2.

Pointer into the local memory to an array, DIMENSION

(lld_a, LOCc(ja+m-1) if side='L',
 (lld_a, LOCc(ja+n-1) if side='R',

where $lld\_a \geq \max(1, LOCr(ia+k-1))$.

On entry, the i -th row must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by [p?gerqf](#) in the k rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$.

The argument $A(ia:ia+k-1, ja:*)$ is modified by the routine but restored on exit.

ia

(global) INTEGER.

The row index in the global array A indicating the first row of sub(A).

ja

(global) INTEGER.

The column index in the global array A indicating the first column of sub(A).

desca

(global and local) INTEGER array of DIMENSION ($dlen\_$).

The array descriptor for the distributed matrix A .

tau

(local)

REAL for psormr2

DOUBLE PRECISION for pdormr2

COMPLEX for pcunmr2

COMPLEX*16 for pzunmr2.

Array, DIMENSION $LOCc(ia+k-1)$. This array contains the scalar factors $tau(i)$ of the elementary reflector $H(i)$, as returned by [p?gerqf](#). This array is tied to the distributed matrix A .

c

(local)

REAL for psormr2

DOUBLE PRECISION for pdormr2

COMPLEX for pcunmr2

COMPLEX*16 for pzunmr2.

Pointer into the local memory to an array,
 DIMENSION(*lld_c*, *LOCc(jc+n-1)*). On entry, the local
 pieces of the distributed matrix sub (*c*).

ic (global) INTEGER.
 The row index in the global array *c* indicating the first row
 of sub(*c*).

jc (global) INTEGER.
 The column index in the global array *c* indicating the first
 column of sub(*c*).

descc (global and local) INTEGER array of DIMENSION (*dlen_*).
 The array descriptor for the distributed matrix *c*.

work (local)
 REAL for psormr2
 DOUBLE PRECISION for pdormr2
 COMPLEX for pcunmr2
 COMPLEX*16 for pzunmr2.
 Workspace array, DIMENSION (*lwork*).

lwork (local or global) INTEGER.
 The dimension of the array *work*.
lwork is local input and must be at least

```

if side = 'L', lwork ≥ mpc0 + max(max(1, nqc0),
numroc(numroc(m+iroffc, mb_a, 0, 0, nprow),
mb_a, 0, 0, lcmp)),
if side = 'R', lwork ≥ nqc0 + max(1, mpc0),
where lcmp = lcm/nprow,
lcm = iclm(nprow, npc0),
iroffc = mod(ic-1, mb_c),
icoffc = mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, myrow, rsrc_c, nprow),
iccol = indxg2p(jc, nb_c, mycol, csrc_c, npc0),
Mpc0 = numroc(m+iroffc, mb_c, myrow, icrow,
nprow),
Nqc0 = numroc(n+icoffc, nb_c, mycol, iccol,
npc0),

```

`ilcm`, `indxg2p` and `numroc` are ScaLAPACK tool functions; `myrow`, `mycol`, `nprow`, and `npcol` can be determined by calling the subroutine `blacs_gridinfo`.
 If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>c</code>	On exit, <code>c</code> is overwritten by $Q \cdot \text{sub}(C)$, or $Q^T \cdot \text{sub}(C) / Q^H \cdot \text{sub}(C)$, or $\text{sub}(C) \cdot Q$, or $\text{sub}(C) \cdot Q^T / \text{sub}(C) \cdot Q^H$
<code>work</code>	On exit, <code>work(1)</code> returns the minimal and optimal <code>lwork</code> .
<code>info</code>	(local) INTEGER. = 0: successful exit < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <code>info</code> = - (i*100+j), if the <i>i</i> -th argument is a scalar and had an illegal value, then <code>info</code> = -i.



NOTE. The distributed submatrices $A(ia:*, ja:*)$ and $C(ic:ic+m-1, jc:jc+n-1)$ must verify some alignment properties, namely the following expressions should be true:

If `side = 'L'`, (`nb_a.eq.mb_c .AND. icoffa.eq.iroffc`)

If `side = 'R'`, (`nb_a.eq.nb_c .AND. icoffa.eq.icoffc .AND. iacol.eq.iccol`).

p?pbtrsv

*Solves a single triangular linear system via
frontsolve or backsolve where the triangular matrix
is a factor of a banded matrix computed by
p?pbtrf.*

Syntax

```
call pspbtrsv(uplo, trans, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf,  
work, lwork, info)
```

```
call pdpbtrsv(uplo, trans, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf,  
work, lwork, info)
```

```
call pcpbtrsv(uplo, trans, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf,  
work, lwork, info)
```

```
call pzpbttrsv(uplo, trans, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf,  
work, lwork, info)
```

Description

The routine p?pbtrsv solves a banded triangular system of linear equations

$$A(1:n, ja:ja+n-1) * X = B(jb:jb+n-1, 1:nrhs)$$

or

$$A(1:n, ja:ja+n-1)^T * X = B(jb:jb+n-1, 1:nrhs) \text{ for real flavors,}$$

$$A(1:n, ja:ja+n-1)^H * X = B(jb:jb+n-1, 1:nrhs) \text{ for complex flavors,}$$

where $A(1:n, ja:ja+n-1)$ is a banded triangular matrix factor produced by the Cholesky factorization code p?pbtrf and is stored in $A(1:n, ja:ja+n-1)$ and *af*. The matrix stored in $A(1:n, ja:ja+n-1)$ is either upper or lower triangular according to *uplo*.

Routine p?pbtrf must be called first.

Input Parameters

uplo (global) CHARACTER. Must be 'U' or 'L'.
If *uplo* = 'U', upper triangle of $A(1:n, ja:ja+n-1)$ is stored;
If *uplo* = 'L', lower triangle of $A(1:n, ja:ja+n-1)$ is stored.

<i>trans</i>	(global) CHARACTER. Must be 'N' or 'T' or 'C'. If <i>trans</i> = 'N', solve with $A(1:n, ja:ja+n-1)$; If <i>trans</i> = 'T' or 'C' for real flavors, solve with $A(1:n, ja:ja+n-1)^T$. If <i>trans</i> = 'C' for complex flavors, solve with conjugate transpose $(A(1:n, ja:ja+n-1))^H$.
<i>n</i>	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix $A(1:n, ja:ja+n-1)$. $n \geq 0$.
<i>bw</i>	(global) INTEGER. The number of subdiagonals in 'L' or 'U', $0 \leq bw \leq n-1$.
<i>nrhs</i>	(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix $B(jb:jb+n-1, 1:nrhs)$; $nrhs \geq 0$.
<i>a</i>	(local) REAL for pspbtrsv DOUBLE PRECISION for pdpbtrsv COMPLEX for pcpbtrsv COMPLEX*16 for pzpbtrsv. Pointer into the local memory to an array with the first DIMENSION $lld\_a \geq (bw+1)$, stored in <i>desca</i>. On entry, this array contains the local pieces of the n-by-n symmetric banded distributed Cholesky factor L or $L^T * A(1:n, ja:ja+n-1)$. This local portion is stored in the packed banded format used in LAPACK. See the <i>Application Notes</i> below and the ScaLAPACK manual for more detail on the format of distributed matrices.
<i>ja</i>	(global) INTEGER. The index in the global array <i>a</i> that points to the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	(global and local) INTEGER array, DIMENSION (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i>.

	If 1D type (<i>dtype_a</i> = 501), then <i>dlen</i> ≥ 7; If 2D type (<i>dtype_a</i> = 1), then <i>dlen</i> ≥ 9. Contains information on mapping of <i>A</i> to memory. (See ScaLAPACK manual for full description and options.)
<i>b</i>	(local) REAL for pspbtrsv DOUBLE PRECISION for pdpbtrsv COMPLEX for pcpbtrsv COMPLEX*16 for pzpbtrsv. Pointer into the local memory to an array of local lead DIMENSION <i>lld_b</i> ≥ <i>nb</i>. On entry, this array contains the local pieces of the right hand sides <i>B</i>(<i>jb:jb+n-1</i>, 1:<i>nrhs</i>).
<i>ib</i>	(global) INTEGER. The row index in the global array <i>B</i> that points to the first row of the matrix to be operated on (which may be either all of <i>B</i> or a submatrix of <i>B</i>).
<i>descb</i>	(global and local) INTEGER array, DIMENSION (<i>dlen_</i>). The array descriptor for the distributed matrix <i>B</i>. If 1D type (<i>dtype_b</i> = 502), then <i>dlen</i> ≥ 7; If 2D type (<i>dtype_b</i> = 1), then <i>dlen</i> ≥ 9. Contains information on mapping of <i>B</i> to memory. Please, see ScaLAPACK manual for full description and options.
<i>laf</i>	(local) INTEGER. The size of user-input auxiliary Fillin space <i>af</i>. Must be <i>laf</i> ≥ (<i>nb</i>+2*<i>bw</i>)*<i>bw</i>. If <i>laf</i> is not large enough, an error code will be returned and the minimum acceptable size will be returned in <i>af</i>(1).
<i>work</i>	(local) REAL for pspbtrsv DOUBLE PRECISION for pdpbtrsv COMPLEX for pcpbtrsv COMPLEX*16 for pzpbtrsv. The array <i>work</i> is a temporary workspace array of DIMENSION <i>lwork</i>. This space may be overwritten in between calls to routines.

lwork (local or global) INTEGER. The size of the user-input workspace *work*, must be at least $lwork \geq bw * nrhs$. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.

Output Parameters

af (local)
 REAL for pspbtrsv
 DOUBLE PRECISION for pdpbtrsv
 COMPLEX for pcgbtrsv
 COMPLEX*16 for pzgbtrsv.
 The array *af* is of DIMENSION *laf*. It contains auxiliary Fillin space. Fillin is created during the factorization routine *p?pbtrf* and this is stored in *af*. If a linear system is to be solved using *p?pbtrs* after the factorization routine, *af* must not be altered after the factorization.

b On exit, this array contains the local piece of the solutions distributed matrix *x*.

work(1) On exit, *work*(1) contains the minimum value of *lwork*.

info (local) INTEGER.
 = 0: successful exit
 < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value,
 then *info* = $-(i*100+j)$,
 if the *i*-th argument is a scalar and had an illegal value,
 then *info* = $-i$.

Application Notes

If the factorization routine and the solve routine are to be called separately to solve various sets of right-hand sides using the same coefficient matrix, the auxiliary space *af* must not be altered between calls to the factorization routine and the solve routine.

The best algorithm for solving banded and tridiagonal linear systems depends on a variety of parameters, especially the bandwidth. Currently, only algorithms designed for the case $N/P \gg bw$ are implemented. These algorithms go by many names, including Divide and Conquer, Partitioning, domain decomposition-type, etc.

The Divide and Conquer algorithm assumes the matrix is narrowly banded compared with the number of equations. In this situation, it is best to distribute the input matrix A one-dimensionally, with columns atomic and rows divided amongst the processes. The basic algorithm divides the banded matrix up into P pieces with one stored on each processor, and then proceeds in 2 phases for the factorization or 3 for the solution of a linear system.

- 1. Local Phase:** The individual pieces are factored independently and in parallel. These factors are applied to the matrix creating fill-in, which is stored in a non-inspectable way in auxiliary space af . Mathematically, this is equivalent to reordering the matrix A as PAP^T and then factoring the principal leading submatrix of size equal to the sum of the sizes of the matrices factored on each processor. The factors of these submatrices overwrite the corresponding parts of A in memory.
- 2. Reduced System Phase:** A small $(bw * (P-1))$ system is formed representing interaction of the larger blocks and is stored (as are its factors) in the space af . A parallel Block Cyclic Reduction algorithm is used. For a linear system, a parallel front solve followed by an analogous backsolve, both using the structure of the factored matrix, are performed.
- 3. Back Substitution Phase:** For a linear system, a local backsubstitution is performed on each processor in parallel.

p?pttrsv

Solves a single triangular linear system via frontsolve or backsolve where the triangular matrix is a factor of a tridiagonal matrix computed by p?pttrf.

Syntax

```
call pspttrsv(uplo, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work,
lwork, info)

call pdpttrsv(uplo, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work,
lwork, info)

call pcpttrsv(uplo, trans, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf,
work, lwork, info)

call pzpttrsv(uplo, trans, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf,
work, lwork, info)
```

Description

This routine solves a tridiagonal triangular system of linear equations

$$A(1:n, ja:ja+n-1) * X = B(jb:jb+n-1, 1:nrhs)$$

or

$$A(1:n, ja:ja+n-1)^T * X = B(jb:jb+n-1, 1:nrhs) \text{ for real flavors,}$$

$$A(1:n, ja:ja+n-1)^H * X = B(jb:jb+n-1, 1:nrhs) \text{ for complex flavors,}$$

where $A(1:n, ja:ja+n-1)$ is a tridiagonal triangular matrix factor produced by the Cholesky factorization code [p?pttrf](#) and is stored in $A(1:n, ja:ja+n-1)$ and *af*. The matrix stored in $A(1:n, ja:ja+n-1)$ is either upper or lower triangular according to *uplo*.

Routine [p?pttrf](#) must be called first.

Input Parameters

<i>uplo</i>	(global) CHARACTER. Must be 'U' or 'L'. If <i>uplo</i> = 'U', upper triangle of $A(1:n, ja:ja+n-1)$ is stored; If <i>uplo</i> = 'L', lower triangle of $A(1:n, ja:ja+n-1)$ is stored.
<i>trans</i>	(global) CHARACTER. Must be 'N' or 'C'. If <i>trans</i> = 'N', solve with $A(1:n, ja:ja+n-1)$; If <i>trans</i> = 'C' (for complex flavors), solve with conjugate transpose $(A(1:n, ja:ja+n-1))^H$.
<i>n</i>	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix $A(1:n, ja:ja+n-1)$. $n \geq 0$.
<i>nrhs</i>	(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix $B(jb:jb+n-1, 1:nrhs)$; $nrhs \geq 0$.
<i>d</i>	(local) REAL for pspttrsv DOUBLE PRECISION for pdpttrsv COMPLEX for pcpttrsv

	COMPLEX*16 for pzpttrsv. Pointer to the local part of the global vector storing the main diagonal of the matrix; must be of size $\geq \text{desca}(nb\_)$.
<i>e</i>	(local) REAL for pspttrsv DOUBLE PRECISION for pdpttrsv COMPLEX for pcpttrsv COMPLEX*16 for pzpttrsv. Pointer to the local part of the global vector storing the upper diagonal of the matrix; must be of size $\geq \text{desca}(nb\_)$. Globally, $du(n)$ is not referenced, and du must be aligned with d .
<i>ja</i>	(global) INTEGER. The index in the global array A that points to the start of the matrix to be operated on (which may be either all of A or a submatrix of A).
<i>desca</i>	(global and local) INTEGER array, DIMENSION ($dlen\_$). The array descriptor for the distributed matrix A . If 1D type ($dtype\_a = 501$ or 502), then $dlen \geq 7$; If 2D type ($dtype\_a = 1$), then $dlen \geq 9$. Contains information on mapping of A to memory. See ScaLAPACK manual for full description and options.
<i>b</i>	(local) REAL for pspttrsv DOUBLE PRECISION for pdpttrsv COMPLEX for pcpttrsv COMPLEX*16 for pzpttrsv. Pointer into the local memory to an array of local lead DIMENSION $lld\_b \geq nb$. On entry, this array contains the local pieces of the right hand sides $B(jb:jb+n-1, 1:nrhs)$.
<i>ib</i>	(global) INTEGER. The row index in the global array B that points to the first row of the matrix to be operated on (which may be either all of B or a submatrix of B).
<i>descb</i>	(global and local) INTEGER array, DIMENSION ($dlen\_$). The array descriptor for the distributed matrix B .

If 1D type ($dtype\_b = 502$), then $dlen \geq 7$;

If 2D type ($dtype\_b = 1$), then $dlen \geq 9$.

Contains information on mapping of B to memory. See ScaLAPACK manual for full description and options.

laf

(local)

INTEGER. The size of user-input auxiliary Fillin space *af*.

Must be $laf \geq (nb+2*bw)*bw$.

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

work

(local)

REAL for pspttrsv

DOUBLE PRECISION for pdpttrsv

COMPLEX for pcpttrsv

COMPLEX*16 for pzpttrsv.

The array *work* is a temporary workspace array of DIMENSION *lwork*. This space may be overwritten in between calls to routines.

lwork

(local or global) INTEGER. The size of the user-input

workspace *work*, must be at least $lwork \geq (10+2*\min(100, nrhs))*npcol+4*nrhs$. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.

Output Parameters

d, e

(local).

REAL for pspttrsv

DOUBLE PRECISION for pdpttrsv

COMPLEX for pcpttrsv

COMPLEX*16 for pzpttrsv.

On exit, these arrays contain information on the factors of the matrix.

af

(local)

REAL for pspttrsv

DOUBLE PRECISION for pdpttrsv

COMPLEX for pcpttrsv

COMPLEX*16 for pzpttrsv.

	The array <i>af</i> is of DIMENSION <i>laaf</i> . It contains auxiliary Fillin space. Fillin is created during the factorization routine <i>p?pbtrf</i> and this is stored in <i>af</i> . If a linear system is to be solved using <i>p?pttrs</i> after the factorization routine, <i>af</i> must not be altered after the factorization.
<i>b</i>	On exit, this array contains the local piece of the solutions distributed matrix <i>x</i> .
<i>work</i> (1)	On exit, <i>work</i> (1) contains the minimum value of <i>lwork</i> .
<i>info</i>	(local) INTEGER. = 0: successful exit < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = $-(i*100+j)$, if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = $-i$.

p?potf2

Computes the Cholesky factorization of a symmetric/Hermitian positive definite matrix (local unblocked algorithm).

Syntax

```
call pspotf2(uplo, n, a, ia, ja, desca, info)
call pdpotf2(uplo, n, a, ia, ja, desca, info)
call pcpotf2(uplo, n, a, ia, ja, desca, info)
call pzpotf2(uplo, n, a, ia, ja, desca, info)
```

Description

This routine computes the Cholesky factorization of a real symmetric or complex Hermitian positive definite distributed matrix sub (A)=A(ia:ia+n-1, ja:ja+n-1).

The factorization has the form

sub (A) = U'^*U , if *uplo* = 'U', or sub (A) = L^*L' , if *uplo* = 'L',

where *U* is an upper triangular matrix, *L* is lower triangular. X' denotes transpose (conjugate transpose) of *x*.

Input Parameters

<i>uplo</i>	(global) CHARACTER. Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix <i>A</i> is stored. = 'U': upper triangle of sub (<i>A</i>) is stored; = 'L': lower triangle of sub (<i>A</i>) is stored.
<i>n</i>	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix sub (<i>A</i>). $n \geq 0$.
<i>a</i>	(local) REAL for pspotf2 DOUBLE PRECISION or pdpotf2 COMPLEX for pcpotf2 COMPLEX*16 for pzpotf2. Pointer into the local memory to an array of <code>DIMENSION(11d_a, LOCC(ja+n-1))</code> containing the local pieces of the <i>n</i>-by-<i>n</i> symmetric distributed matrix sub(<i>A</i>) to be factored. If <i>uplo</i> = 'U', the leading <i>n</i>-by-<i>n</i> upper triangular part of sub(<i>A</i>) contains the upper triangular matrix and the strictly lower triangular part of this matrix is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i>-by-<i>n</i> lower triangular part of sub(<i>A</i>) contains the lower triangular matrix and the strictly upper triangular part of sub(<i>A</i>) is not referenced.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>A</i> indicating the first row and the first column of the sub(<i>A</i>), respectively.
<i>desca</i>	(global and local) INTEGER array, <code>DIMENSION(dlen_)</code>. The array descriptor for the distributed matrix <i>A</i>.

Output Parameters

<i>a</i>	(local) On exit, if <i>uplo</i> = 'U', the upper triangular part of the distributed matrix contains the Cholesky factor <i>U</i>;

if `uplo = 'L'`, the lower triangular part of the distributed matrix contains the Cholesky factor L .

`info` (local) INTEGER.
 = 0: successful exit
 < 0: if the i -th argument is an array and the j -entry had an illegal value,
 then `info` = $-(i*100+j)$,
 if the i -th argument is a scalar and had an illegal value,
 then `info` = $-i$.
 > 0: if `info` = k , the leading minor of order k is not positive definite, and the factorization could not be completed.

p?rscl

Multiplies a vector by the reciprocal of a real scalar.

Syntax

```
call psrscl(n, sa, sx, ix, jx, descx, incx)
call pdrscl(n, sa, sx, ix, jx, descx, incx)
call pcsrscl(n, sa, sx, ix, jx, descx, incx)
call pzdrscl(n, sa, sx, ix, jx, descx, incx)
```

Description

This routine multiplies an n -element real/complex vector `sub(x)` by the real scalar $1/a$. This is done without overflow or underflow as long as the final result `sub(x)/a` does not overflow or underflow.

`sub(x)` denotes `x(ix:ix+n-1, jx:jx)`, if `incx` = 1,

and `x(ix:ix, jx:jx+n-1)`, if `incx` = `m_x`.

Input Parameters

`n` (global) INTEGER.
 The number of components of the distributed vector `sub(x)`.
 $n \geq 0$.

<i>sa</i>	REAL for psrcsl/pcsrcsl DOUBLE PRECISION for pdrsl/pzdrsl. The scalar <i>a</i> that is used to divide each component of the vector <i>x</i>. This parameter must be ≥ 0.
<i>sx</i>	REAL for psrcsl DOUBLE PRECISION for pdrsl COMPLEX for pcsrcsl COMPLEX*16 for pzdrsl. Array containing the local pieces of a distributed matrix of DIMENSION of at least $((j_x-1)*m_x + i_x + (n-1)*abs(incx))$. This array contains the entries of the distributed vector sub(<i>x</i>).
<i>ix</i>	(global) INTEGER. The row index of the submatrix of the distributed matrix <i>x</i> to operate on.
<i>jx</i>	(global) INTEGER. The column index of the submatrix of the distributed matrix <i>x</i> to operate on.
<i>descx</i>	(global and local). INTEGER. Array of DIMENSION 8. The array descriptor for the distributed matrix <i>x</i>.
<i>incx</i>	(global) INTEGER. The increment for the elements of <i>x</i>. This version supports only two values of <i>incx</i>, namely 1 and m_x.

Output Parameters

<i>sx</i>	On exit, the result x/a .

p?sygs2/p?hegs2

Reduces a symmetric/Hermitian definite generalized eigenproblem to standard form, using the factorization results obtained from p?potrf (local unblocked algorithm).

Syntax

```
call pssygs2(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, info)
call pdsygs2(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, info)
call pchegs2(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, info)
call pzhegs2(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, info)
```

Description

The routine p?sygs2/p?hegs2 reduces a real symmetric-definite or a complex Hermitian-definite generalized eigenproblem to standard form.

Here $\text{sub}(A)$ denotes $A(ia:ia+n-1, ja:ja+n-1)$, and $\text{sub}(B)$ denotes $B(ib:ib+n-1, jb:jb+n-1)$.

If $\text{ibtype} = 1$, the problem is

$$\text{sub}(A) * x = \lambda * \text{sub}(B) * x$$

and $\text{sub}(A)$ is overwritten by

$$\text{inv}(U^T) * \text{sub}(A) * \text{inv}(U) \text{ or } \text{inv}(L) * \text{sub}(A) * \text{inv}(L^T) \quad - \text{ for real flavors, and}$$

$$\text{inv}(U^H) * \text{sub}(A) * \text{inv}(U) \text{ or } \text{inv}(L) * \text{sub}(A) * \text{inv}(L^H) \quad - \text{ for complex flavors.}$$

If $\text{ibtype} = 2$ or 3 , the problem is

$$\text{sub}(A) * \text{sub}(B) x = \lambda * x \text{ or } \text{sub}(B) * \text{sub}(A) x = \lambda * x$$

and $\text{sub}(A)$ is overwritten by

$$U * \text{sub}(A) * U^T \text{ or } L * T * \text{sub}(A) * L \quad - \text{ for real flavors and}$$

$$U * \text{sub}(A) * U^H \text{ or } L * H * \text{sub}(A) * L \quad - \text{ for complex flavors.}$$

The matrix $\text{sub}(B)$ must have been previously factorized as $U^T * U$ or $L * L^T$ (for real flavors), or as $U^H * U$ or $L * L^H$ (for complex flavors) by p?potrf.

Input Parameters

<i>ibtype</i>	(global) INTEGER. = 1: compute $\text{inv}(U^T) * \text{sub}(A) * \text{inv}(U)$, or $\text{inv}(L) * \text{sub}(A) * \text{inv}(L^T)$ for real subroutines, and $\text{inv}(U^H) * \text{sub}(A) * \text{inv}(U)$, or $\text{inv}(L) * \text{sub}(A) * \text{inv}(L^H)$ for complex subroutines; = 2 or 3: compute $U * \text{sub}(A) * U^T$, or $L^T * \text{sub}(A) * L$ for real subroutines, and $U * \text{sub}(A) * U^H$ or $L^H * \text{sub}(A) * L$ for complex subroutines.
<i>uplo</i>	(global) CHARACTER Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix $\text{sub}(A)$ is stored, and how $\text{sub}(B)$ is factorized. = 'U': Upper triangular of $\text{sub}(A)$ is stored and $\text{sub}(B)$ is factorized as $U^T * U$ (for real subroutines) or as $U^H * U$ (for complex subroutines). = 'L': Lower triangular of $\text{sub}(A)$ is stored and $\text{sub}(B)$ is factorized as $L * L^T$ (for real subroutines) or as $L * L^H$ (for complex subroutines)
<i>n</i>	(global) INTEGER. The order of the matrices $\text{sub}(A)$ and $\text{sub}(B)$. $n \geq 0$.
<i>a</i>	(local) REAL for pssygs2 DOUBLE PRECISION for pdsygs2 COMPLEX for pchezs2 COMPLEX*16 for pzhezs2. Pointer into the local memory to an array, $\text{DIMENSION}(\text{lld\_a}, \text{LOCc}(ja+n-1))$. On entry, this array contains the local pieces of the n-by-n symmetric/Hermitian distributed matrix $\text{sub}(A)$. If <i>uplo</i> = 'U', the leading n-by-n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and the strictly lower triangular part of $\text{sub}(A)$ is not referenced.

	If <i>uplo</i> = 'L', the leading <i>n</i> -by- <i>n</i> lower triangular part of sub(<i>A</i>) contains the lower triangular part of the matrix, and the strictly upper triangular part of sub(<i>A</i>) is not referenced.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>A</i> indicating the first row and the first column of the sub(<i>A</i>), respectively.
<i>desca</i>	(global and local) INTEGER array, DIMENSION (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>B</i>	(local) REAL for pssygs2 DOUBLE PRECISION for pdsygs2 COMPLEX for pcheqs2 COMPLEX*16 for pzheqs2. Pointer into the local memory to an array, DIMENSION(<i>lld_b</i> , <i>LOCc(jb+n-1)</i>). On entry, this array contains the local pieces of the triangular factor from the Cholesky factorization of sub(<i>B</i>) as returned by <i>p?potrf</i> .
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the global array <i>B</i> indicating the first row and the first column of the sub(<i>B</i>), respectively.
<i>descb</i>	(global and local) INTEGER array, DIMENSION (<i>dlen_</i>). The array descriptor for the distributed matrix <i>B</i> .

Output Parameters

<i>a</i>	(local) On exit, if <i>info</i> = 0, the transformed matrix is stored in the same format as sub(<i>A</i>).
<i>info</i>	INTEGER. = 0: successful exit. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = - (<i>i</i> *100), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

p?sytd2/p?hetd2

Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation (local unblocked algorithm).

Syntax

```
call pssytd2(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
call pdsytd2(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
call pchetd2(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
call pzhetd2(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
```

Description

The routine p?sytd2/p?hetd2 reduces a real symmetric/complex Hermitian matrix sub(*A*) to symmetric/Hermitian tridiagonal form *T* by an orthogonal/unitary similarity transformation:

$Q^* \text{sub}(A) Q = T$, where $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$.

Input Parameters

<i>uplo</i>	(global) CHARACTER. Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix sub(<i>A</i>) is stored: = 'U': upper triangular = 'L': lower triangular
<i>n</i>	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix sub(<i>A</i>). $n \geq 0$.
<i>a</i>	(local) REAL for pssytd2 DOUBLE PRECISION for pdsytd2 COMPLEX for pchetd2 COMPLEX*16 for pzhetd2. Pointer into the local memory to an array, DIMENSION(<i>lld_a</i> , <i>LOCc</i> (<i>ja</i> + <i>n</i> -1)).

On entry, this array contains the local pieces of the n -by- n symmetric/Hermitian distributed matrix $\text{sub}(A)$.
 If $\text{uplo} = 'U'$, the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and the strictly lower triangular part of $\text{sub}(A)$ is not referenced.
 If $\text{uplo} = 'L'$, the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the matrix, and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.

ia, ja (global) INTEGER.
 The row and column indices in the global array A indicating the first row and the first column of the $\text{sub}(A)$, respectively.

desca (global and local) INTEGER array, DIMENSION ($dlen\_$). The array descriptor for the distributed matrix A .

work (local)
 REAL for pssytd2
 DOUBLE PRECISION for pdsytd2
 COMPLEX for pchetd2
 COMPLEX*16 for pzhetd2.
 The array *work* is a temporary workspace array of
 DIMENSION $lwork$.

Output Parameters

a On exit, if $\text{uplo} = 'U'$, the diagonal and first superdiagonal of $\text{sub}(A)$ are overwritten by the corresponding elements of the tridiagonal matrix T , and the elements above the first superdiagonal, with the array *tau*, represent the orthogonal/unitary matrix Q as a product of elementary reflectors;
 if $\text{uplo} = 'L'$, the diagonal and first subdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T , and the elements below the first subdiagonal, with the array *tau*, represent the orthogonal/unitary matrix Q as a product of elementary reflectors. See the *Application Notes* below.

d (local)
 REAL for pssytd2/pchetd2
 DOUBLE PRECISION for pdsytd2/pzhetd2.

	Array, $\text{DIMENSION}(\text{LOCc}(ja+n-1))$. The diagonal elements of the tridiagonal matrix T: $d(i) = a(i, i)$; d is tied to the distributed matrix A.
<i>e</i>	(local) REAL for pssytd2/pchetd2 DOUBLE PRECISION for pdsytd2/pzhetd2. Array, $\text{DIMENSION}(\text{LOCc}(ja+n-1))$, if <i>uplo</i> = 'U', $\text{LOCc}(ja+n-2)$ otherwise. The off-diagonal elements of the tridiagonal matrix T: $e(i) = a(i, i+1)$ if <i>uplo</i> = 'U', $e(i) = a(i+1, i)$ if <i>uplo</i> = 'L'. e is tied to the distributed matrix A.
<i>tau</i>	(local) REAL for pssytd2 DOUBLE PRECISION for pdsytd2 COMPLEX for pchetd2 COMPLEX*16 for pzhetd2. Array, $\text{DIMENSION}(\text{LOCc}(ja+n-1))$. The scalar factors of the elementary reflectors. <i>tau</i> is tied to the distributed matrix A.
<i>work</i> (1)	On exit, <i>work</i>(1) returns the minimal and optimal value of <i>lwork</i>.
<i>lwork</i>	(local or global) INTEGER. The dimension of the workspace array <i>work</i>. <i>lwork</i> is local input and must be at least $lwork \geq 3n$. If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.
<i>info</i>	(local) INTEGER. = 0: successful exit < 0: if the <i>i</i>-th argument is an array and the <i>j</i>-entry had an illegal value, then <i>info</i> = $-(i*100)$, if the <i>i</i>-th argument is a scalar and had an illegal value, then <i>info</i> = $-i$.

Application Notes

If $uplo = 'U'$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(n-1) * \dots * H(2) * H(1)$$

Each $H(i)$ has the form

$$H(i) = I - \tau v v^T,$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(i+1:n) = 0$ and $v(i) = 1$; $v(1:i-1)$ is stored on exit in $A(ia:ia+i-2, ja+i)$, and τ in $TAU(ja+i-1)$.

If $uplo = 'L'$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(1) * H(2) * \dots * H(n-1).$$

Each $H(i)$ has the form

$$H(i) = I - \tau v v^T,$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i) = 0$ and $v(i+1) = 1$; $v(i+2:n)$ is stored on exit in $A(ia+i+1:ia+n-1, ja+i-1)$, and τ in $TAU(ja+i-1)$.

The contents of sub (A) on exit are illustrated by the following examples with $n = 5$:

$$\begin{array}{cc} \text{if } uplo='U': & \text{if } uplo='L': \\ \begin{bmatrix} d & e & v_2 & v_3 & v_4 \\ & d & e & v_3 & v_4 \\ & & d & e & v_4 \\ & & & d & e \\ & & & & d \end{bmatrix} & \begin{bmatrix} d & & & & \\ e & d & & & \\ v_1 & e & d & & \\ v_1 & v_2 & e & d & \\ v_1 & v_2 & v_3 & e & d \end{bmatrix} \end{array}$$

where d and e denotes diagonal and off-diagonal elements of T , and v_i denotes an element of the vector defining $H(i)$.



NOTE. The distributed submatrix sub(A) must verify some alignment properties, namely the following expression should be true:

($mb\_a.eq.nb\_a$.AND. $iroffa.eq.icoffa$) with $iroffa = \text{mod}(ia - 1, mb\_a)$
and $icoffa = \text{mod}(ja - 1, nb\_a)$.

p?trti2

Computes the inverse of a triangular matrix (local unblocked algorithm).

Syntax

```
call pstrti2(uplo, diag, n, a, ia, ja, desca, info)
call pdtrti2(uplo, diag, n, a, ia, ja, desca, info)
call pctrti2(uplo, diag, n, a, ia, ja, desca, info)
call pztrti2(uplo, diag, n, a, ia, ja, desca, info)
```

Description

This routine computes the inverse of a real/complex upper or lower triangular block matrix sub (A) = A(ia:ia+n-1, ja:ja+n-1).

This matrix should be contained in one and only one process memory space (local operation).

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Specifies whether the matrix sub (A) is upper or lower triangular. = 'U': sub (A) is upper triangular = 'L': sub (A) is lower triangular.
<i>diag</i>	(global) CHARACTER*1. Specifies whether or not the matrix A is unit triangular. = 'N': sub (A) is non-unit triangular = 'U': sub (A) is unit triangular.
<i>n</i>	(global) INTEGER. The number of rows and columns to be operated on, i.e., the order of the distributed submatrix sub (A). $n \geq 0$.
<i>a</i>	(local) REAL for pstrti2 DOUBLE PRECISION for pdtrti2 COMPLEX for pctrti2 COMPLEX*16 for pztrti2.

Pointer into the local memory to an array,

`DIMENSION(lld_a, LOCC(ja+n-1)).`

On entry, this array contains the local pieces of the triangular matrix $\text{sub}(A)$.

If `uplo = 'U'`, the leading n -by- n upper triangular part of the matrix $\text{sub}(A)$ contains the upper triangular part of the matrix, and the strictly lower triangular part of $\text{sub}(A)$ is not referenced.

If `uplo = 'L'`, the leading n -by- n lower triangular part of the matrix $\text{sub}(A)$ contains the lower triangular part of the matrix, and the strictly upper triangular part of $\text{sub}(A)$ is not referenced. If `diag = 'U'`, the diagonal elements of $\text{sub}(A)$ are not referenced either and are assumed to be 1.

`ia, ja`

(global) `INTEGER`.

The row and column indices in the global array `A` indicating the first row and the first column of the $\text{sub}(A)$, respectively.

`desca`

(global and local) `INTEGER` array, `DIMENSION(dlen_)`. The array descriptor for the distributed matrix `A`.

Output Parameters

`a`

On exit, the (triangular) inverse of the original matrix, in the same storage format.

`info`

`INTEGER`.

= 0: successful exit

< 0: if the i -th argument is an array and the j -entry had an illegal value,

then `info = - (i*100)`,

if the i -th argument is a scalar and had an illegal value,

then `info = -i`.

slamsh

Sends multiple shifts through a small (single node) matrix to maximize the number of bulges that can be sent through.

Syntax

```
call slamsh(s, lds, nbulge, jblk, h, ldh, n, ulp)
call dlamsh(s, lds, nbulge, jblk, h, ldh, n, ulp)
```

Description

This routine sends multiple shifts through a small (single node) matrix to see how small consecutive subdiagonal elements are modified by subsequent shifts in an effort to maximize the number of bulges that can be sent through. The subroutine should only be called when there are multiple shifts/bulges (*nbulge* > 1) and the first shift is starting in the middle of an unreduced Hessenberg matrix because of two or more small consecutive subdiagonal elements.

Input Parameters

<i>s</i>	(local) INTEGER. REAL for slamsh DOUBLE PRECISION for dlamsh Array, DIMENSION (<i>lds</i> ,*). On entry, the matrix of shifts. Only the 2x2 diagonal of <i>s</i> is referenced. It is assumed that <i>s</i> has <i>jblk</i> double shifts (size 2).
<i>lds</i>	(local) INTEGER. On entry, the leading dimension of <i>s</i> ; unchanged on exit. $1 < nbulge \leq jblk \leq lds/2$.
<i>nbulge</i>	(local) INTEGER. On entry, the number of bulges to send through <i>h</i> (>1). <i>nbulge</i> should be less than the maximum determined (<i>jblk</i>). $1 < nbulge \leq jblk \leq lds/2$.
<i>jblk</i>	(local) INTEGER. On entry, the leading dimension of <i>s</i> ; unchanged on exit.
<i>h</i>	(local) INTEGER.

REAL for slamsh
 DOUBLE PRECISION for dlamsh
 Array, DIMENSION (*lds*, *n*).
 On entry, the local matrix to apply the shifts on.
h should be aligned so that the starting row is 2.

ldh (local)
 INTEGER.
 On entry, the leading dimension of *H*; unchanged on exit.

n (local) INTEGER.
 On entry, the size of *H*. If all the bulges are expected to go through, *n* should be at least $4nbulge+2$. Otherwise, *nbulge* may be reduced by this routine.

ulp (local)
 REAL for slamsh
 DOUBLE PRECISION for dlamsh
 On entry, machine precision. Unchanged on exit.

Output Parameters

s On exit, the data is rearranged in the best order for applying.

nbulge On exit, the maximum number of bulges that can be sent through.

h On exit, the data is destroyed.

?laref

Applies Householder reflectors to matrices on either their rows or columns.

Syntax

```
call slaref(type, a, lda, wantz, z, ldz, block, irow1, icol1, istart, istop,
itmp1, itmp2, liloz, lihiz, vecs, v2, v3, t1, t2, t3)

call dlaref(type, a, lda, wantz, z, ldz, block, irow1, icol1, istart, istop,
itmp1, itmp2, liloz, lihiz, vecs, v2, v3, t1, t2, t3)
```

Description

This routine applies one or several Householder reflectors of size 3 to one or two matrices (if column is specified) on either their rows or columns.

Input Parameters

<i>type</i>	(global) CHARACTER*1. If <i>type</i> = 'R', apply reflectors to the rows of the matrix (apply from left). Otherwise, apply reflectors to the columns of the matrix. Unchanged on exit.
<i>a</i>	(global) REAL for <i>sref</i> DOUBLE PRECISION for <i>dlaref</i> Array, DIMENSION (<i>lda</i> , *). On entry, the matrix to receive the reflections.
<i>lda</i>	(local) INTEGER. On entry, the leading dimension of <i>A</i> ; unchanged on exit.
<i>wantz</i>	(global) LOGICAL. If <i>wantz</i> = .TRUE., apply any column reflections to <i>Z</i> as well. If <i>wantz</i> = .FALSE., do no additional work on <i>Z</i> .
<i>z</i>	(global) REAL for <i>sref</i> DOUBLE PRECISION for <i>dlaref</i> Array, DIMENSION (<i>ldz</i> , *). On entry, the second matrix to receive column reflections.
<i>ldz</i>	(local) INTEGER. On entry, the leading dimension of <i>Z</i> ; unchanged on exit.
<i>block</i>	(global). LOGICAL. = .TRUE. : apply several reflectors at once and read their data from the <i>vecs</i> array; = .FALSE. : apply the single reflector given by <i>v2</i> , <i>v3</i> , <i>t1</i> , <i>t2</i> , and <i>t3</i> .
<i>irow1</i>	(local) INTEGER. On entry, the local row element of the matrix <i>A</i> .
<i>icol1</i>	(local) INTEGER. On entry, the local column element of the matrix <i>A</i> .
<i>istart</i>	(global) INTEGER.

	Specifies the "number" of the first reflector. <i>istart</i> is used as an index into <i>vecs</i> if <i>block</i> is set. <i>istart</i> is ignored if <i>block</i> is <code>.FALSE.</code> .
<i>istop</i>	(global) INTEGER. Specifies the "number" of the last reflector. <i>istop</i> is used as an index into <i>vecs</i> if <i>block</i> is set. <i>istop</i> is ignored if <i>block</i> is <code>.FALSE.</code> .
<i>itmp1</i>	(local) INTEGER. Starting range into <i>A</i> . For rows, this is the local first column. For columns, this is the local first row.
<i>itmp2</i>	(local) INTEGER. Ending range into <i>A</i> . For rows, this is the local last column. For columns, this is the local last row.
<i>liloz, lihi</i>	(local). INTEGER. Serve the same purpose as <i>itmp1</i> , <i>itmp2</i> but for <i>Z</i> when <i>wantz</i> is set.
<i>vecs</i>	(global) REAL for <code>slaref</code> DOUBLE PRECISION for <code>dlaref</code> . Array of size $3 * n$ (matrix size). This array holds the size 3 reflectors one after another and is only accessed when <i>block</i> is <code>.TRUE.</code> .
<i>v2,v3,t1,t2,t3</i>	(global). INTEGER. REAL for <code>slaref</code> DOUBLE PRECISION for <code>dlaref</code> . These parameters hold information on a single size 3 Householder reflector and are read when <i>block</i> is <code>.FALSE.</code> , and overwritten when <i>block</i> is <code>.TRUE.</code> .

Output Parameters

<i>a</i>	On exit, the updated matrix.
<i>z</i>	Changed only if <i>wantz</i> is set. If <i>wantz</i> is <code>.FALSE.</code> , <i>z</i> is not referenced.
<i>irow1</i>	Undefined.
<i>icol1</i>	Undefined.

v2,v3,t1,t2,t3

These parameters are read when *block* is `.FALSE.`, and overwritten when *block* is `.TRUE.`.

?lasorte

Sorts eigenpairs by real and complex data types.

Syntax

```
call slasorte(s, lds, j, out, info)
```

```
call dlasorte(s, lds, j, out, info)
```

Description

This routine sorts eigenpairs so that real eigenpairs are together and complex eigenpairs are together. This helps to employ 2x2 shifts easily since every second subdiagonal is guaranteed to be zero. This routine does no parallel work and makes no calls.

Input Parameters

<i>s</i>	(local) INTEGER. REAL for <code>slasorte</code> DOUBLE PRECISION for <code>dlasorte</code> Array, DIMENSION (<i>lds</i>). On entry, a matrix already in Schur form.
<i>lds</i>	(local) INTEGER. On entry, the leading dimension of the array <i>s</i> ; unchanged on exit.
<i>j</i>	(local) INTEGER. On entry, the order of the matrix <i>s</i> ; unchanged on exit.
<i>out</i>	(local) INTEGER. REAL for <code>slasorte</code> DOUBLE PRECISION for <code>dlasorte</code> Array, DIMENSION (2*j). The work buffer required by the routine.
<i>info</i>	(local) INTEGER.

Set, if the input matrix had an odd number of real eigenvalues and things could not be paired or if the input matrix s was not originally in Schur form. 0 indicates successful completion.

Output Parameters

s On exit, the diagonal blocks of s have been rewritten to pair the eigenvalues. The resulting matrix is no longer similar to the input.

out Work buffer.

?lasrt2

Sorts numbers in increasing or decreasing order.

Syntax

```
call slasrt2(id, n, d, key, info)
call dlasrt2(id, n, d, key, info)
```

Description

This routine is modified LAPACK routine [?lasrt](#), which sorts the numbers in d in increasing order (if $id = 'I'$) or in decreasing order (if $id = 'D'$). It uses Quick Sort, reverting to Insertion Sort on arrays of size ≤ 20 . Dimension of `STACK` limits n to about 2^{32} .

Input Parameters

id CHARACTER*1.
= 'I': sort d in increasing order;
= 'D': sort d in decreasing order.

n INTEGER. The length of the array d .

d REAL for `slasrt2`
DOUBLE PRECISION for `dlasrt2`.
Array, DIMENSION (n).
On entry, the array to be sorted.

key INTEGER.
Array, DIMENSION (n).

On entry, *key* contains a key to each of the entries in *d()*.
Typically, $key(i) = i$ for all *i*.

Output Parameters

<i>D</i>	On exit, <i>d</i> has been sorted into increasing order ($d(1) \leq \dots \leq d(n)$) or into decreasing order ($d(1) \geq \dots \geq d(n)$), depending on <i>id</i> .
<i>info</i>	INTEGER. = 0: successful exit < 0: if <i>info</i> = -i, the <i>i</i> -th argument had an illegal value.
<i>key</i>	On exit, <i>key</i> is permuted in exactly the same manner as <i>d()</i> was permuted from input to output. Therefore, if $key(i)$ = <i>i</i> for all <i>i</i> upon input, then $d_{out}(i) = d_{in}(key(i))$.

?stein2

*Computes the eigenvectors corresponding to
specified eigenvalues of a real symmetric
tridiagonal matrix, using inverse iteration.*

Syntax

```
call sstein2(n, d, e, m, w, iblock, isplit, orfac, z, ldz, work, iwork, ifail,
info)
call dstein2(n, d, e, m, w, iblock, isplit, orfac, z, ldz, work, iwork, ifail,
info)
```

Description

This routine is a modified LAPACK routine [?stein](#). It computes the eigenvectors of a real symmetric tridiagonal matrix *T* corresponding to specified eigenvalues, using inverse iteration.

The maximum number of iterations allowed for each eigenvector is specified by an internal parameter *maxits* (currently set to 5).

Input Parameters

<i>n</i>	INTEGER. The order of the matrix <i>T</i> ($n \geq 0$).

<i>m</i>	INTEGER. The number of eigenvectors to be found ($0 \leq m \leq n$).
<i>d</i> , <i>e</i> , <i>w</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays: <i>d</i>(*), DIMENSION (<i>n</i>). The <i>n</i> diagonal elements of the tridiagonal matrix <i>T</i>. <i>e</i>(*), DIMENSION (<i>n</i>). The (<i>n</i>-1) subdiagonal elements of the tridiagonal matrix <i>T</i>, in elements 1 to <i>n</i>-1. <i>e</i>(<i>n</i>) need not be set. <i>w</i>(*), DIMENSION (<i>n</i>). The first <i>m</i> elements of <i>w</i> contain the eigenvalues for which eigenvectors are to be computed. The eigenvalues should be grouped by split-off block and ordered from smallest to largest within the block. (The output array <i>w</i> from ?stebz with ORDER = 'B' is expected here). The dimension of <i>w</i> must be at least $\max(1, n)$.
<i>iblock</i>	INTEGER. Array, DIMENSION (<i>n</i>). The submatrix indices associated with the corresponding eigenvalues in <i>w</i> ; <i>iblock</i>(<i>i</i>) = 1, if eigenvalue <i>w</i>(<i>i</i>) belongs to the first submatrix from the top, <i>iblock</i>(<i>i</i>) = 2, if eigenvalue <i>w</i>(<i>i</i>) belongs to the second submatrix, etc. (The output array <i>iblock</i> from ?stebz is expected here).
<i>isplit</i>	INTEGER. Array, DIMENSION (<i>n</i>). The splitting points, at which <i>T</i> breaks up into submatrices. The first submatrix consists of rows/columns 1 to <i>isplit</i>(1), the second submatrix consists of rows/columns <i>isplit</i>(1)+1 through <i>isplit</i>(2), etc. (The output array <i>isplit</i> from ?stebz is expected here).
<i>orfac</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. <i>orfac</i> specifies which eigenvectors should be orthogonalized. Eigenvectors that correspond to eigenvalues which are within $orfac * T$ of each other are to be orthogonalized.

<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i> ; <i>ldz</i> $\geq \max(1, n)$.
<i>work</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Workspace array, DIMENSION (5 <i>n</i>).
<i>iwork</i>	INTEGER. Workspace array, DIMENSION (<i>n</i>).

Output Parameters

<i>z</i>	REAL for sstein2 DOUBLE PRECISION for dstein2 Array, DIMENSION (<i>ldz</i> , <i>m</i>). The computed eigenvectors. The eigenvector associated with the eigenvalue $w(i)$ is stored in the <i>i</i> -th column of <i>z</i> . Any vector that fails to converge is set to its current iterate after <i>maxits</i> iterations.
<i>ifail</i>	INTEGER. Array, DIMENSION (<i>m</i>). On normal exit, all elements of <i>ifail</i> are zero. If one or more eigenvectors fail to converge after <i>maxits</i> iterations, then their indices are stored in the array <i>ifail</i> .
<i>info</i>	INTEGER. <i>info</i> = 0, the exit is successful. <i>info</i> < 0: if <i>info</i> = - <i>i</i> , the <i>i</i> -th had an illegal value. <i>info</i> > 0: if <i>info</i> = <i>i</i> , then <i>i</i> eigenvectors failed to converge in <i>maxits</i> iterations. Their indices are stored in the array <i>ifail</i> .

?dbtf2

Computes an LU factorization of a general band matrix with no pivoting (local unblocked algorithm).

Syntax

```
call sdbtf2(m, n, kl, ku, ab, ldab, info)
call ddbtf2(m, n, kl, ku, ab, ldab, info)
call cdbtf2(m, n, kl, ku, ab, ldab, info)
call zdbtf2(m, n, kl, ku, ab, ldab, info)
```

Description

This routine computes an *LU* factorization of a general real/complex *m*-by-*n* band matrix *A* without using partial pivoting with row interchanges.

This is the unblocked version of the algorithm, calling [BLAS Routines and Functions](#).

Input Parameters

<i>m</i>	INTEGER. The number of rows of the matrix <i>A</i> ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in <i>A</i> ($n \geq 0$).
<i>kl</i>	INTEGER. The number of sub-diagonals within the band of <i>A</i> ($kl \geq 0$).
<i>ku</i>	INTEGER. The number of super-diagonals within the band of <i>A</i> ($ku \geq 0$).
<i>ab</i>	REAL for sdbtf2 DOUBLE PRECISION for ddbtf2 COMPLEX for cdbtf2 COMPLEX*16 for zdbtf2. Array, DIMENSION (<i>ldab</i>, <i>n</i>). The matrix <i>A</i> in band storage, in rows <i>kl</i>+1 to <i>2kl</i>+<i>ku</i>+1; rows 1 to <i>kl</i> of the array need not be set. The <i>j</i>-th column of <i>A</i> is stored in the <i>j</i>-th column of the array <i>ab</i> as follows: $ab(kl+ku+1+i-j, j) = A(i, j)$ for $\max(1, j-ku) \leq i \leq \min(m, j+kl)$.

ldab INTEGER. The leading dimension of the array *ab*.
($ldab \geq 2kl + ku + 1$)

Output Parameters

ab On exit, details of the factorization: *u* is stored as an upper triangular band matrix with $kl+ku$ superdiagonals in rows 1 to $kl+ku+1$, and the multipliers used during the factorization are stored in rows $kl+ku+2$ to $2*kl+ku+1$. See the *Application Notes* below for further details.

info INTEGER.
= 0: successful exit
< 0: if *info* = - *i*, the *i*-th argument had an illegal value,
> 0: if *info* = + *i*, *u*(*i*,*i*) is 0. The factorization has been completed, but the factor *u* is exactly singular. Division by 0 will occur if you use the factor *u* for solving a system of linear equations.

Application Notes

The band storage scheme is illustrated by the following example, when $m = n = 6$, $kl = 2$, $ku = 1$:

on entry	on exit
$\begin{bmatrix} * & a_{12} & a_{23} & a_{34} & a_{45} & a_{56} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & a_{66} \\ a_{21} & a_{32} & a_{43} & a_{54} & a_{65} & * \\ a_{31} & a_{42} & a_{53} & a_{64} & * & * \end{bmatrix}$	$\begin{bmatrix} * & u_{12} & u_{23} & u_{34} & u_{45} & u_{56} \\ u_{11} & u_{22} & u_{33} & u_{44} & u_{55} & u_{66} \\ m_{21} & m_{32} & m_{43} & m_{54} & m_{65} & * \\ m_{31} & m_{42} & m_{53} & m_{64} & * & * \end{bmatrix}$

The routine does not use array elements marked *; elements marked + need not be set on entry, but the routine requires them to store elements of *u*, because of fill-in resulting from the row interchanges.

?dbtrf

Computes an LU factorization of a general band matrix with no pivoting (local blocked algorithm).

Syntax

```
call sdbtrf(m, n, kl, ku, ab, ldab, info)
call ddbtrf(m, n, kl, ku, ab, ldab, info)
call cdbtrf(m, n, kl, ku, ab, ldab, info)
call zdbtrf(m, n, kl, ku, ab, ldab, info)
```

Description

This routine computes an LU factorization of a real m -by- n band matrix A without using partial pivoting or row interchanges.

This is the blocked version of the algorithm, calling [BLAS Routines and Functions](#).

Input Parameters

m	INTEGER. The number of rows of the matrix A ($m \geq 0$).
n	INTEGER. The number of columns in A ($n \geq 0$).
kl	INTEGER. The number of sub-diagonals within the band of A ($kl \geq 0$).
ku	INTEGER. The number of super-diagonals within the band of A ($ku \geq 0$).
ab	REAL for sdbtrf DOUBLE PRECISION for ddbtrf COMPLEX for cdbtrf COMPLEX*16 for zdbtrf. Array, DIMENSION ($ldab, n$). The matrix A in band storage, in rows $kl+1$ to $2kl+ku+1$; rows 1 to kl of the array need not be set. The j-th column of A is stored in the j-th column of the array ab as follows: $ab(kl+ku+1+i-j, j) = A(i, j)$ for $\max(1, j-ku) \leq i \leq \min(m, j+kl)$.

ldab INTEGER. The leading dimension of the array *ab*.
 $(ldab \geq 2kl + ku + 1)$

Output Parameters

ab On exit, details of the factorization: *u* is stored as an upper triangular band matrix with $kl+ku$ superdiagonals in rows 1 to $kl+ku+1$, and the multipliers used during the factorization are stored in rows $kl+ku+2$ to $2*kl+ku+1$. See the *Application Notes* below for further details.

info INTEGER.
 = 0: successful exit
 < 0: if *info* = - *i*, the *i*-th argument had an illegal value,
 > 0: if *info* = + *i*, *u*(*i*,*i*) is 0. The factorization has been completed, but the factor *u* is exactly singular. Division by 0 will occur if you use the factor *u* for solving a system of linear equations.

Application Notes

The band storage scheme is illustrated by the following example, when $m = n = 6$, $kl = 2$, $ku = 1$:

on entry	on exit
$\begin{bmatrix} * & a12 & a23 & a34 & a45 & a56 \\ a11 & a22 & a33 & a44 & a55 & a66 \\ a21 & a32 & a43 & a54 & a65 & * \\ a31 & a42 & a53 & a64 & * & * \end{bmatrix}$	$\begin{bmatrix} * & u12 & u23 & u34 & u45 & u56 \\ u11 & u22 & u33 & u44 & u55 & u66 \\ m21 & m32 & m43 & m54 & m65 & * \\ m31 & m42 & m53 & m64 & * & * \end{bmatrix}$

The routine does not use array elements marked *.

?dtttrf

Computes an LU factorization of a general tridiagonal matrix with no pivoting (local blocked algorithm).

Syntax

```
call sdttrf(n, dl, d, du, info)
call ddttrf(n, dl, d, du, info)
call cdttrf(n, dl, d, du, info)
call zdttrf(n, dl, d, du, info)
```

Description

This routine computes an LU factorization of a real or complex tridiagonal matrix A using elimination without partial pivoting.

The factorization has the form $A = L^*U$, where L is a product of unit lower bidiagonal matrices and U is upper triangular with nonzeros only in the main diagonal and first superdiagonal.

Input Parameters

n INTEGER. The order of the matrix A ($n \geq 0$).

dl, d, du REAL for sdttrf
DOUBLE PRECISION for ddttrf
COMPLEX for cdttrf
COMPLEX*16 for zdttrf.
Arrays containing elements of A .
The array dl of DIMENSION $(n - 1)$ contains the sub-diagonal elements of A .
The array d of DIMENSION n contains the diagonal elements of A .
The array du of DIMENSION $(n - 1)$ contains the super-diagonal elements of A .

Output Parameters

dl Overwritten by the $(n-1)$ multipliers that define the matrix L from the LU factorization of A .

<i>d</i>	Overwritten by the <i>n</i> diagonal elements of the upper triangular matrix <i>U</i> from the <i>LU</i> factorization of <i>A</i> .
<i>du</i>	Overwritten by the <i>(n-1)</i> elements of the first super-diagonal of <i>U</i> .
<i>info</i>	INTEGER. = 0: successful exit < 0: if <i>info</i> = - <i>i</i> , the <i>i</i> -th argument had an illegal value, > 0: if <i>info</i> = <i>i</i> , <i>u(i,i)</i> is exactly 0. The factorization has been completed, but the factor <i>U</i> is exactly singular. Division by 0 will occur if you use the factor <i>U</i> for solving a system of linear equations.

?dttrsv

Solves a general tridiagonal system of linear equations using the LU factorization computed by ?dttrf.

Syntax

```
call sdttrsv(uplo, trans, n, nrhs, dl, d, du, b, ldb, info)
call ddttrsv(uplo, trans, n, nrhs, dl, d, du, b, ldb, info)
call cdttrsv(uplo, trans, n, nrhs, dl, d, du, b, ldb, info)
call zdttrsv(uplo, trans, n, nrhs, dl, d, du, b, ldb, info)
```

Description

This routine solves one of the following systems of linear equations:

$$L * X = B, L^T * X = B, \text{ or } L^H * X = B,$$

$$U * X = B, U^T * X = B, \text{ or } U^H * X = B$$

with factors of the tridiagonal matrix *A* from the *LU* factorization computed by [?dttrf](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether to solve with <i>L</i> or <i>U</i> .
<i>trans</i>	CHARACTER. Must be 'N' or 'T' or 'C'.

Indicates the form of the equations:

If $trans = 'N'$, then $A^*X=B$ is solved for X (no transpose).

If $trans = 'T'$, then $A^T * X = B$ is solved for X (transpose).

If $trans = 'C'$, then $A^H * X = B$ is solved for X (conjugate transpose).

n INTEGER. The order of the matrix A ($n \geq 0$).

$nrhs$ INTEGER. The number of right-hand sides, that is, the number of columns in the matrix B ($nrhs \geq 0$).

dl, d, du, b REAL for sdttrsv
DOUBLE PRECISION for ddttrsv
COMPLEX for cdttrsv
COMPLEX*16 for zdttrsv.
Arrays of DIMENSIONS: $dl(n-1)$, $d(n)$, $du(n-1)$, $b(ldb, nrhs)$.
The array dl contains the $(n-1)$ multipliers that define the matrix L from the LU factorization of A .
The array d contains n diagonal elements of the upper triangular matrix U from the LU factorization of A .
The array du contains the $(n-1)$ elements of the first super-diagonal of U .
On entry, the array b contains the right-hand side matrix B .

ldb INTEGER. The leading dimension of the array b ; $ldb \geq \max(1, n)$.

Output Parameters

b Overwritten by the solution matrix X .

$info$ INTEGER. If $info=0$, the execution is successful.
If $info = -i$, the i -th parameter had an illegal value.

?pttrsv

Solves a symmetric (Hermitian) positive-definite tridiagonal system of linear equations, using the $L^*D^*L^H$ factorization computed by ?pttrf.

Syntax

```
call spttrsv(trans, n, nrhs, d, e, b, ldb, info)
call dpttrsv(trans, n, nrhs, d, e, b, ldb, info)
call cpttrsv(uplo, trans, n, nrhs, d, e, b, ldb, info)
call zpttrsv(uplo, trans, n, nrhs, d, e, b, ldb, info)
```

Description

This routine solves one of the triangular systems:

$L^T * X = B$, or $L * X = B$ for real flavors,

or

$L * X = B$, or $L^H * X = B$,

$U * X = B$, or $U^H * X = B$ for complex flavors,

where L (or U for complex flavors) is the Cholesky factor of a Hermitian positive-definite tridiagonal matrix A such that

$A = L^*D^*L^H$ (computed by [spttrf/dpttrf](#))

or

$A = U^H * D * U$ or $A = L^*D^*L^H$ (computed by [cpttrf/zpttrf](#)).

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Specifies whether the superdiagonal or the subdiagonal of the tridiagonal matrix A is stored and the form of the factorization: If <i>uplo</i> = 'U', e is the superdiagonal of U , and $A = U^H * D * U$ or $A = L^*D^*L^H$; if <i>uplo</i> = 'L', e is the subdiagonal of L , and $A = L^*D^*L^H$. The two forms are equivalent, if A is real.

<i>trans</i>	CHARACTER. Specifies the form of the system of equations: for real flavors: if <i>trans</i> = 'N': $L^*X = B$ (no transpose) if <i>trans</i> = 'T': $L^T * X = B$ (transpose) for complex flavors: if <i>trans</i> = 'N': $U^*X = B$ or $L^*X = B$ (no transpose) if <i>trans</i> = 'C': $U^H * X = B$ or $L^H * X = B$ (conjugate transpose).
<i>n</i>	INTEGER. The order of the tridiagonal matrix <i>A</i> . $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right hand sides, that is, the number of columns of the matrix <i>B</i> . $nrhs \geq 0$.
<i>d</i>	REAL array, DIMENSION (<i>n</i>). The <i>n</i> diagonal elements of the diagonal matrix <i>D</i> from the factorization computed by ?pttrf .
<i>e</i>	COMPLEX array, DIMENSION (<i>n</i> -1). The (<i>n</i> -1) off-diagonal elements of the unit bidiagonal factor <i>U</i> or <i>L</i> from the factorization computed by ?pttrf . See <i>uplo</i> .
<i>b</i>	COMPLEX array, DIMENSION (<i>ldb</i> , <i>nrhs</i>). On entry, the right hand side matrix <i>B</i> .
<i>ldb</i>	INTEGER. The leading dimension of the array <i>b</i> . $ldb \geq \max(1, n)$.

Output Parameters

<i>b</i>	On exit, the solution matrix <i>x</i> .
<i>info</i>	INTEGER. = 0: successful exit < 0: if <i>info</i> = -<i>i</i>, the <i>i</i>-th argument had an illegal value.

?steqr2

Computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method.

Syntax

```
call ssteqr2(compz, n, d, e, z, ldz, nr, work, info)
call dsteqr2(compz, n, d, e, z, ldz, nr, work, info)
```

Description

This routine is a modified version of LAPACK routine [?steqr](#). The routine [?steqr2](#) computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method. [?steqr2](#) is modified from [?steqr](#) to allow each ScaLAPACK process running [?steqr2](#) to perform updates on a distributed matrix Q . Proper usage of [?steqr2](#) can be gleaned from examination of ScaLAPACK routine [p?syev](#).

Input Parameters

<i>compz</i>	CHARACTER*1. Must be 'N' or 'I'. If <i>compz</i> = 'N', the routine computes eigenvalues only. If <i>compz</i> = 'I', the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix T . z must be initialized to the identity matrix by p?laset or ?laset prior to entering this subroutine.
<i>n</i>	INTEGER. The order of the matrix T ($n \geq 0$).
<i>d, e, work</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays: d contains the diagonal elements of T . The dimension of d must be at least $\max(1, n)$. e contains the $(n-1)$ subdiagonal elements of T . The dimension of e must be at least $\max(1, n-1)$. $work$ is a workspace array. The dimension of $work$ is $\max(1, 2*n-2)$. If <i>compz</i> = 'N', then $work$ is not referenced.
<i>z</i>	(local) REAL for ssteqr2 DOUBLE PRECISION for dsteqr2

Array, global DIMENSION (n, n), local DIMENSION (ldz, nr).
 If *compz* = 'V', then *z* contains the orthogonal matrix used
 in the reduction to tridiagonal form.

ldz INTEGER. The leading dimension of the array *z*. Constraints:
 $ldz \geq 1$,
 $ldz \geq \max(1, n)$, if eigenvectors are desired.

nr INTEGER. $nr = \max(1, \text{numroc}(n, nb, myprow, 0, nprocs))$.
 If *compz* = 'N', then *nr* is not referenced.

Output Parameters

d REAL array, DIMENSION (n), for *ssteqr2*.
 DOUBLE PRECISION array, DIMENSION (n), for *dsteqr2*.
 On exit, the eigenvalues in ascending order, if *info* = 0.
 See also *info*.

e REAL array, DIMENSION ($n-1$), for *ssteqr2*.
 DOUBLE PRECISION array, DIMENSION ($n-1$), for *dsteqr2*.
 On exit, *e* has been destroyed.

z (local)
 REAL for *ssteqr2*
 DOUBLE PRECISION for *dsteqr2*
 Array, global DIMENSION (n, n), local DIMENSION (ldz, nr).
 On exit, if *info* = 0, then,
 if *compz* = 'V', *z* contains the orthonormal eigenvectors
 of the original symmetric matrix, and if *compz* = 'I', *z*
 contains the orthonormal eigenvectors of the symmetric
 tridiagonal matrix. If *compz* = 'N', then *z* is not referenced.

info INTEGER.
info = 0, the exit is successful.
info < 0: if *info* = $-i$, the i -th had an illegal value.
info > 0: the algorithm has failed to find all the
 eigenvalues in a total of $30n$ iterations;
 if *info* = i , then i elements of *e* have not converged to
 zero; on exit, *d* and *e* contain the elements of a symmetric
 tridiagonal matrix, which is orthogonally similar to the
 original matrix.

Utility Functions and Routines

This section describes ScaLAPACK utility functions and routines. Summary information about these routines is given in the following table:

Table 7-2 ScaLAPACK Utility Functions and Routines

Routine Name	Data Types	Description
<code>p?labad</code>	<code>s, d</code>	Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.
<code>p?lachieee</code>	<code>s, d</code>	Performs a simple check for the features of the IEEE standard. (C interface function).
<code>p?lamch</code>	<code>s, d</code>	Determines machine parameters for floating-point arithmetic.
<code>p?lasnbt</code>	<code>s, d</code>	Computes the position of the sign bit of a floating-point number. (C interface function).
<code>pxerbla</code>		Error handling routine called by ScaLAPACK routines.

`p?labad`

Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.

Syntax

```
call pslabad(ictxt, small, large)
```

```
call pdlabad(ictxt, small, large)
```

Description

This routine takes as input the values computed by `p?lamch` for underflow and overflow, and returns the square root of each of these values if the log of `large` is sufficiently large. This subroutine is intended to identify machines with a large exponent range, such as the Crays, and redefine the underflow and overflow limits to be the square roots of the values computed by `p?lamch`. This subroutine is needed because `p?lamch` does not compensate for poor arithmetic in the upper half of the exponent range, as is found on a Cray.

In addition, this routine performs a global minimization and maximization on these values, to support heterogeneous computing networks.

Input Parameters

<i>ictxt</i>	(global) INTEGER. The BLACS context handle in which the computation takes place.
<i>small</i>	(local). REAL PRECISION for pslabad. DOUBLE PRECISION for pdlabad. On entry, the underflow threshold as computed by p?lamch.
<i>large</i>	(local). REAL PRECISION for pslabad. DOUBLE PRECISION for pdlabad. On entry, the overflow threshold as computed by p?lamch.

Output Parameters

<i>small</i>	(local). On exit, if $\log_{10}(\textit{large})$ is sufficiently large, the square root of <i>small</i> , otherwise unchanged.
<i>large</i>	(local). On exit, if $\log_{10}(\textit{large})$ is sufficiently large, the square root of <i>large</i> , otherwise unchanged.

p?lachkieee

Performs a simple check for the features of the IEEE standard. (C interface function).

Syntax

```
void pslachkieee(int *isieee, float *rmax, float *rmin);
void pdlachkieee(int *isieee, float *rmax, float *rmin);
```

Description

This routine performs a simple check to make sure that the features of the IEEE standard are implemented. In some implementations, p?lachkieee may not return.

Note that all arguments are call-by-reference so that this routine can be directly called from Fortran code.

This is a ScaLAPACK internal subroutine and arguments are not checked for unreasonable values.

Input Parameters

<i>rmax</i>	(local). REAL for pslach IEEE DOUBLE PRECISION for pdlach IEEE The overflow threshold (= ?lamch ('O')).
<i>rmin</i>	(local). REAL for pslach IEEE DOUBLE PRECISION for pdlach IEEE The underflow threshold (= ?lamch ('U')).

Output Parameters

<i>isieee</i>	(local). INTEGER. On exit, <i>isieee</i> = 1 implies that all the features of the IEEE standard that we rely on are implemented. On exit, <i>isieee</i> = 0 implies that some the features of the IEEE standard that we rely on are missing.

p?lamch

Determines machine parameters for floating-point arithmetic.

Syntax

```
val = pslamch(ictxt, cmach)
val = pdlamch(ictxt, cmach)
```

Description

This routine determines single precision machine parameters.

Input Parameters

ictxt (global). INTEGER. The BLACS context handle in which the computation takes place.

cmach (global) CHARACTER*1.
Specifies the value to be returned by p?lamch:

- = 'E' or 'e', p?lamch := eps
- = 'S' or 's', p?lamch := sfmin
- = 'B' or 'b', p?lamch := base
- = 'P' or 'p', p?lamch := eps*base
- = 'N' or 'n', p?lamch := t
- = 'R' or 'r', p?lamch := rnd
- = 'M' or 'm', p?lamch := emin
- = 'U' or 'u', p?lamch := rmin
- = 'L' or 'l', p?lamch := emax
- = 'O' or 'o', p?lamch := rmax,

where

- eps = relative machine precision
- sfmin = safe minimum, such that 1/sfmin does not overflow
- base = base of the machine
- prec = eps*base
- t = number of (base) digits in the mantissa
- rnd = 1.0 when rounding occurs in addition, 0.0 otherwise
- emin = minimum exponent before (gradual) underflow
- rmin = underflow threshold - $\text{base}^{(\text{emin}-1)}$
- emax = largest exponent before overflow
- rmax = overflow threshold - $(\text{base}^{\text{emax}}) * (1-\text{eps})$

Output Parameters

val Value returned by the routine.

pslasnbt

Computes the position of the sign bit of a floating-point number. (C interface function).

Syntax

```
void pslasnbt(int *ieflag);
```

```
void pdlasnbt(int *ieflag);
```

Description

This routine finds the position of the signbit of a single/double precision floating point number. This routine assumes IEEE arithmetic, and hence, tests only the 32-nd bit (for single precision) or 32-nd and 64-th bits (for double precision) as a possibility for the signbit. `sizeof(int)` is assumed equal to 4 bytes.

If a compile time flag (`NO_IEEE`) indicates that the machine does not have IEEE arithmetic, `ieflag = 0` is returned.

Output Parameters

ieflag

INTEGER.

This flag indicates the position of the signbit of any single/double precision floating point number.

ieflag = 0, if the compile time flag `NO_IEEE` indicates that the machine does not have IEEE arithmetic, or if `sizeof(int)` is different from 4 bytes.

ieflag = 1 indicates that the signbit is the 32-nd bit for a single precision routine.

In the case of a double precision routine:

ieflag = 1 indicates that the signbit is the 32-nd bit (Big Endian).

ieflag = 2 indicates that the signbit is the 64-th bit (Little Endian).

pxerbla

Error handling routine called by ScaLAPACK routines.

Syntax

```
call pxerbla(ictxt, sname, info)
```

Description

This routine is an error handler for the *ScaLAPACK* routines. It is called by a *ScaLAPACK* routine if an input parameter has an invalid value. A message is printed. Program execution is not terminated. For the ScaLAPACK driver and computational routines, a `RETURN` statement is issued following the call to `pxerbla`.

Control returns to the higher-level calling routine, and it is left to the user to determine how the program should proceed. However, in the specialized low-level ScaLAPACK routines (auxiliary routines that are Level 2 equivalents of computational routines), the call to `pxerbla()` is immediately followed by a call to `BLACS_ABORT()` to terminate program execution since recovery from an error at this level in the computation is not possible.

It is always good practice to check for a nonzero value of *info* on return from a ScaLAPACK routine. Installers may consider modifying this routine in order to call system-specific exception-handling facilities.

Input Parameters

<i>ictxt</i>	(global) INTEGER The BLACS context handle, indicating the global context of the operation. The context itself is global.
<i>sname</i>	(global) CHARACTER*6 The name of the routine which called <code>pxerbla</code> .
<i>info</i>	(global) INTEGER. The position of the invalid parameter in the parameter list of the calling routine.

Sparse Solver Routines

Intel® Math Kernel Library (Intel® MKL) provides user-callable sparse solver software to solve real or complex, symmetric, structurally symmetric or non-symmetric, positive definite, indefinite or Hermitian sparse linear system of equations.

The terms and concepts required to understand the use of the Intel MKL sparse solver routines are discussed in the [Appendix A "Linear Solvers Basics"](#) [Appendix "Linear Solvers Basics"](#) . If you are familiar with linear sparse solvers and sparse matrix storage schemes, you can skip these sections and go directly to the interface descriptions.

This chapter describes the direct sparse solver [PARDISO*](#) and the alternative interface for the direct sparse solver referred to here as [DSS interface](#); [iterative sparse solvers \(ISS\)](#) based on the reverse communication interface (RCI); and [two preconditioners](#) based on the incomplete LU factorization technique.

PARDISO* - Parallel Direct Sparse Solver Interface

This section describes the interface to the shared-memory multiprocessing parallel direct sparse solver known as the PARDISO* solver. The interface is Fortran, but can be called from C programs by observing Fortran parameter passing and naming conventions used by the supported compilers and operating systems. A discussion of the algorithms used in the PARDISO* software and more information on the solver can be found at <http://www.pardiso-project.org>.

The current implementation of the PARDISO* solver additionally supports the out-of-core (OOC) version.

The PARDISO* package is a high-performance, robust, memory efficient and easy to use software for solving large sparse symmetric and unsymmetric linear systems of equations on shared memory multiprocessors. The solver uses a combination of left- and right-looking Level-3 BLAS supernode techniques [[Schenk00-2](#)]. To improve sequential and parallel sparse numerical factorization performance, the algorithms are based on a Level-3 BLAS update and pipelining parallelism is used with a combination of left- and right-looking supernode techniques [[Schenk00](#), [Schenk01](#), [Schenk02](#), [Schenk03](#)]. The parallel pivoting methods allow complete supernode pivoting to compromise numerical stability and scalability during the factorization process. For sufficiently large problem sizes, numerical experiments demonstrate that the scalability of the parallel algorithm is nearly independent of the shared-memory multiprocessing architecture and a speedup of up to seven using eight processors has been observed.

The PARDISO* solver supports a wide range of sparse matrix types (see [Figure 8-1](#) below) and computes the solution of real or complex, symmetric, structurally symmetric or unsymmetric, positive definite, indefinite or Hermitian sparse linear system of equations on shared-memory multiprocessing architectures.

Figure 8-1 Sparse Matrices That Can Be Solved with the PARDISO* Solver



You can find a code example that uses the PARDISO interface routine to solve systems of linear equations in [PARDISO Code Examples](#) section in the [Appendix C](#).

pardiso

Calculates the solution of a set of sparse linear equations with multiple right-hand sides.

Syntax

Fortran:

```
call pardiso (pt, maxfct, mnum, type, phase, n, a, ia, ja, perm, nrhs, iparm,
msglvl, b, x, error)
```


C:

```
pardiso (pt, &maxfct, &mnum, &mtype, &phase, &n, a, ia, ja, perm, &nrhs,
iparm, &msglvl, b, x, &error);
```

(An underscore may or may not be required after “pardiso” depending on the OS and compiler conventions for that OS).

Interface:

```
SUBROUTINE pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja, perm, nrhs,
iparm, msglvl, b, x, error)
```

```
INTEGER *4 pt (64)
```

```
INTEGER *4 maxfct, mnum, mtype, phase, n, nrhs, error, ia(*), ja(*), perm(*),
iparm(*)
```

```
REAL *8 a(*), b(n,nrhs), x(n,nrhs)
```

Note that the above interface is given for the 32-bit architectures. For 64-bit architectures, the argument `pt(64)` must be defined as `INTEGER*8` type.

Description

PARDISO calculates the solution of a set of sparse linear equations

$$A \cdot X = B$$

with multiple right-hand sides, using a parallel LU , LDL or LL^T factorization, where A is an n -by- n matrix, and X and B are n -by- $nrhs$ matrices. PARDISO performs the following analysis steps depending on the structure of the input matrix A .

Symmetric Matrices: The solver first computes a symmetric fill-in reducing permutation P based on either the minimum degree algorithm [Liu85] or the nested dissection algorithm from the METIS package [Karypis98] (included with Intel MKL), followed by the parallel left-right looking numerical Cholesky factorization [Schenk00-2] of $PAP^T = LL^T$ for symmetric positive-definite matrices, or $PAP^T = LDL^T$ for symmetric indefinite matrices. The solver uses diagonal pivoting, or 1x1 and 2x2 Bunch and Kaufman pivoting for symmetric indefinite matrices, and an approximation of x is found by forward and backward substitution and iterative refinements.

The coefficient matrix is perturbed whenever numerically acceptable 1x1 and 2x2 pivots cannot be found within the diagonal supernode block. One or two passes of iterative refinements may be required to correct the effect of the perturbations. This restricting notion of pivoting with iterative refinements is effective for highly indefinite symmetric systems. Furthermore the

accuracy of this method is for a large set of matrices from different applications areas as accurate as a direct factorization method that uses complete sparse pivoting techniques [Schenk04].

Another possibility to improve the pivoting accuracy is to use symmetric weighted matching algorithms. These methods identify large entries in the coefficient matrix A that, if permuted close to the diagonal, permit the factorization process to identify more acceptable pivots and proceed with fewer pivot perturbations. The methods are based on maximum weighted matchings and improve the quality of the factor in a complementary way to the alternative idea of using more complete pivoting techniques.

The inertia is also computed for real symmetric indefinite matrices.

Structurally Symmetric Matrices: The solver first computes a symmetric fill-in reducing permutation P followed by the parallel numerical factorization of $PAP^T = QLU^T$. The solver uses partial pivoting in the supernodes and an approximation of X is found by forward and backward substitution and iterative refinements.

Unsymmetric Matrices: The solver first computes a non-symmetric permutation P_{MPS} and scaling matrices D_x and D_c with the aim to place large entries on the diagonal that enhances reliability of the numerical factorization process [Duff99]. In the next step the solver computes a fill-in reducing permutation P based on the matrix $P_{MPS}A + (P_{MPS}A)_T$ followed by the parallel numerical factorization

$$QLUR = PP_{MPS}D_xAD_cP$$

with supernode pivoting matrices Q and R . When the factorization algorithm reaches a point where it cannot factorize the supernodes with this pivoting strategy, it uses a pivoting perturbation strategy similar to [Li99]. The magnitude of the potential pivot is tested against a constant threshold of $\alpha = \epsilon * ||A2||_{inf}$, where ϵ is the machine precision, $A2 = P * P_{MPS} * D_x * A * D_c * P$, and $||A2||_{inf}$ is the infinity norm of the scaled and permuted matrix A . Any tiny pivots encountered during elimination are set to the sign $(l_{II}) * \epsilon * ||A2||_{inf}$, which trades off some numerical stability for the ability to keep pivots from getting too small. Although many failures could render the factorization well-defined but essentially useless, in practice the diagonal elements are rarely modified for a large class of matrices. The result of this pivoting approach is that the factorization is, in general, not exact and iterative refinement may be needed.

Direct-Iterative Preconditioning for Unsymmetric Linear Systems. The solver also allows for a combination of direct and iterative methods [Sonn89] to accelerate the linear solution process for transient simulation. Most of applications of sparse solvers require solutions of systems with gradually changing values of the nonzero coefficient matrix, but the same identical sparsity pattern. In these applications, the analysis phase of the solvers has to be performed

only once and the numerical factorizations are the important time-consuming steps during the simulation. PARDISO uses a numerical factorization $A = LU$ for the first system and applies the factors L and U for the next steps in a preconditioned Krylow-Subspace iteration. If the iteration does not converge, the solver automatically switches back to the numerical factorization. This method can be applied to unsymmetric matrices in PARDISO and the user can select the method using only one input parameter. For further details see the parameter description (*iparm(4)*, *iparm(20)*).

The sparse data storage in PARDISO follows the scheme described in [Sparse Matrix Storage Format](#) section with *ja* standing for *columns*, *ia* for *rowIndex*, and *a* for *values*. The algorithms in PARDISO require column indices *ja* to be increasingly ordered per row and the presence of the diagonal element per row for any symmetric or structurally symmetric matrix. The unsymmetric matrices need no diagonal elements in the PARDISO solver.

PARDISO performs four tasks:

- analysis and symbolic factorization
- numerical factorization
- forward and backward substitution including iterative refinement
- termination to release all internal solver memory.

When an input data structure is not accessed in a call, a `NULL` pointer or any valid address can be passed as a place holder for that argument.

Input Parameters

pt `INTEGER*4` for 32 -bit operating systems;
 `INTEGER*8` for 64 -bit operating systems
 Array, `DIMENSION (64)`
 On entry, solver internal data address pointer. These addresses are passed to the solver and all related internal memory management is organized through this pointer



NOTE. *pt* is an integer array with 64 entries. It is very important that the pointer is initialized with zero at the first call of PARDISO. After that first call you should never modify the pointer, as a serious memory leak can occur. The integer length should be 4-byte on 32-bit operating systems and 8-byte on 64-bit operating systems.

maxfct

INTEGER

Maximal number of factors with identical nonzero sparsity structure that the user would like to keep at the same time in memory. It is possible to store several different factorizations with the same nonzero structure at the same time in the internal data management of the solver. In most of the applications this value is equal to 1.

PARDISO can process several matrices with identical matrix sparsity pattern and store the factors of these matrices at the same time. Matrices with a different sparsity structure can be kept in memory with different memory address pointers *pt*.

mnum

INTEGER

Actual matrix for the solution phase. With this scalar you can define the matrix that you would like to factorize. The value must be: $1 \leq mnum \leq maxfct$.

In most applications this value is equal to 1.

mtype

INTEGER

This scalar value defines the matrix type. The PARDISO solver supports the following matrices:

- = 1 real and structurally symmetric matrix
- = 2 real and symmetric positive definite matrix
- = -2 real and symmetric indefinite matrix
- = 3 complex and structurally symmetric matrix
- = 4 complex and Hermitian positive definite matrix
- = -4 complex and Hermitian indefinite matrix
- = 6 complex and symmetric matrix
- = 11 real and unsymmetric matrix
- = 13 complex and unsymmetric matrix

This parameter influences the pivoting method.

phase

INTEGER

Controls the execution of the solver. It is a two-digit integer ij ($10I + j$, $1 \leq I \leq 3$, $I < j \leq 3$ for normal execution modes). The I digit indicates the starting phase of execution, and j indicates the ending phase. PARDISO has the following phases of execution:

- Phase 1: Fill-reduction analysis and symbolic factorization
- Phase 2: Numerical factorization
- Phase 3: Forward and Backward solve including iterative refinements
- Termination and Memory Release Phase ($phase \leq 0$)

If a previous call to the routine has computed information from previous phases, execution may start at any phase. The *phase* parameter can have the following values:

<i>phase</i>	Solver Execution Steps
11	Analysis
12	Analysis, numerical factorization
13	Analysis, numerical factorization, solve, iterative refinement
22	Numerical factorization
23	Numerical factorization, solve, iterative refinement
33	Solve, iterative refinement
0	Release internal memory for L and U matrix number <i>mnum</i>
-1	Release all internal memory for all matrices

<i>n</i>	INTEGER Number of equations in the sparse linear systems of equations $A * X = B$. Constraint: $n > 0$.
<i>a</i>	REAL/COMPLEX Array. Contains the nonzero values of the coefficient matrix A corresponding to the indices in <i>ja</i> . The size of <i>a</i> is the same as that of <i>ja</i> and the coefficient matrix can be either

real or complex. The matrix must be stored in compressed sparse row format with increasing values of ja for each row. Refer to *values* array description in [Sparse Matrix Storage Format](#) for more details.



NOTE. The nonzeros of each row of the matrix A must be stored in increasing order. For symmetric or structural symmetric matrices, it is also important that the diagonal elements are available and stored in the matrix. If the matrix is symmetric, the array a is only accessed in the factorization phase, in the triangular solution and iterative refinement phase. Unsymmetric matrices are accessed in all phases of the solution process.

ia

INTEGER

Array, dimension $(n+1)$. For $I \leq n$, $ia(I)$ points to the first column index of row I in the array ja in compressed sparse row format. That is, $ia(I)$ gives the index of the element in array a that contains the first non-zero element from row I of A . The last element $ia(n+1)$ is taken to be equal to the number of non-zeros in A , plus one. Refer to *rowIndex* array description in [Sparse Matrix Storage Format](#) for more details. The array ia is also accessed in all phases of the solution process. Note that the row and columns numbers start from 1.

ja

INTEGER

Array. $ja(*)$ contains column indices of the sparse matrix A stored in compressed sparse row format. The indices in each row must be sorted in increasing order. The array ja is also accessed in all phases of the solution process. For symmetric and structurally symmetric matrices it is assumed that zero diagonal elements are also stored in the list of nonzeros in a and ja . For symmetric matrices, the solver needs only the upper triangular part of the system as is shown for *columns* array in [Sparse Matrix Storage Format](#).

<i>perm</i>	INTEGER Array, dimension (n). Holds the permutation vector of size n. The array <i>perm</i> is defined as follows. Let A be the original matrix and $B = P^*A^*P^T$ be the permuted matrix. Row (column) i of A is the $perm(i)$ row (column) of B. The numbering of the array must start with 1 and must describe a permutation. On entry, you can apply your own fill-in reducing ordering to the solver. The permutation vector <i>perm</i> is only accessed if $iparm(5) = 1$.
<i>nrhs</i>	INTEGER Number of right-hand sides that need to be solved for.
<i>iparm</i>	INTEGER Array, dimension (64). This array is used to pass various parameters to PARDISO and to return some useful information after execution of the solver. If $iparm(1) = 0$, PARDISO fills $iparm(1)$, and $iparm(4)$ through $iparm(64)$ with default values and uses them. Note that there is no default value for $iparm(3)$ and the user should supply this value, whether $iparm(1)$ is 0 or 1. Individual components of the <i>iparm</i> array are described below (some of them are described in the Output Parameters section). <i>iparm(1)</i> - use default values. If $iparm(1) = 0$, $iparm(2)$ and $iparm(4)$ through $iparm(64)$ are filled with default values, otherwise the user has to supply all values in <i>iparm</i> from $iparm(2)$ to $iparm(64)$. <i>iparm(2)</i> - fill-in reducing ordering. $iparm(2)$ controls the fill-in reducing ordering for the input matrix. If $iparm(2)$ is 0, the minimum degree algorithm is applied [Li99], if $iparm(2)$ is 2, the solver uses the nested dissection algorithm from the METIS package [Karypis98]. The default value of $iparm(2)$ is 2. <i>iparm(3)</i> - number of processors. $iparm(3)$ must contain the number of processors that are available for the parallel execution. The number must be equal to the MKL environment variable <code>MKL_NUM_THREADS</code>.



CAUTION. User is supposed to control the parallel execution of the solver by explicitly setting `MKL_NUM_THREADS`. If less processors are available than specified, the execution may slow down instead of speeding up.

There is no default value for `iparm(3)`.

`iparm(4)` - preconditioned CGS.

This parameter controls preconditioned CGS [Sonn89] for unsymmetric or structural symmetric matrices and Conjugate-Gradients for symmetric matrices. `iparm(4)` has the form

`iparm(4) = 10 * L + K`

The K and L values have the meanings as follow.

Value of κ Description

0	The factorization is always computed as required by <code>phase</code> .
1	CGS iteration replaces the computation of LU . The preconditioner is LU that was computed at a previous step (the first step or last step with a failure) in a sequence of solutions needed for identical sparsity patterns.
2	CG iteration for symmetric matrices replaces the computation of LU . The preconditioner is LU that was computed at a previous step (the first step or last step with a failure) in a sequence of solutions needed for identical sparsity patterns.

Value L :

The value L controls the stopping criterion of the Krylow-Subspace iteration:

$\text{eps}_{\text{CGS}} = 10^{-L}$ is used in the stopping criterion

$\|dx_I\| / \|dx_1\| < \text{eps}_{\text{CGS}}$

with $||dx_I|| = ||\text{inv}(L*U)*r_I||$ and r_I is the residuum at iteration I of the preconditioned Krylow-Subspace iteration.

Strategy: A maximum number of 150 iterations is fixed by expecting that the iteration will converge before consuming half the factorization time. Intermediate convergence rates and residuum excursions are checked and can terminate the iteration process. If $phase = 23$, then the factorization for a given A is automatically recomputed in these cases where the Krylow-Subspace iteration failed, and the corresponding direct solution is returned. Otherwise the solution from the preconditioned Krylow-Subspace iteration is returned. Using $phase = 33$ results in an error message ($error = 4$) if the stopping criteria for the Krylow-Subspace iteration can not be reached. More information on the failure can be obtained from $iparm(20)$.

The default is $iparm(4) = 0$, and other values are only recommended for an advanced user. $iparm(4)$ must be greater or equal to zero.

Examples:

$iparm(4)$	Description
31	LU -preconditioned CGS iteration with a stopping criterion of 1.0E-3 for unsymmetric matrices
61	LU -preconditioned CGS iteration with a stopping criterion of 1.0E-6 for unsymmetric matrices
62	LU -preconditioned CGS iteration with a stopping criterion of 1.0E-6 for symmetric matrices

$iparm(5)$ - user permutation.

This parameter controls whether the user supplied fill-in reducing permutation is used instead of the integrated multiple-minimum degree or nested dissection algorithms.

This option may be useful for testing reordering algorithms or adapting the code to special applications problems (for instance, to move zero diagonal elements to the end $P^*A^*P^T$). For definition of the permutation, see description of the *perm* parameter.

The default value of *iparm*(5) is 0.

iparm(6) - write solution on *x*.

If *iparm*(6) is 0 (which is the default), then the array *x* contains the solution and the value of *b* is not changed. If *iparm*(6) is 1, then the solver will store the solution on the right hand side *b*.

Note that the array *x* is always used. The default value of *iparm*(6) is 0.

iparm(8)

On entry to the solve and iterative refinement step, *iparm*(8) should be set to the maximum number of iterative refinement steps that the solver will perform. The solver will not perform more than the absolute value of *iparm*(8) steps of iterative refinement and will stop the process if a satisfactory level of accuracy of the solution in terms of backward error has been achieved.

Note that if *iparm*(8) < 0, the accumulation of the residuum is using extended precision real and complex data types. Perturbed pivots result in iterative refinement (independent of *iparm*(8)=0) and the iteration number executed is reported on *iparm*(7).

The solver will automatically perform two steps of iterative refinements when perturbed pivots have been obtained during the numerical factorization and *iparm*(8) was equal to zero.

The number of performed iterative refinement steps is reported on *iparm*(8).

The default value for *iparm*(8) is 0.

iparm(9)

This parameter is reserved for future use. Its value must be set to 0.

iparm(10) - pivoting perturbation.

This parameter instructs PARDISO how to handle small pivots or zero pivots for unsymmetric matrices ($mtype = 11$ or $mtype = 13$) and symmetric matrices ($mtype = -2$, $mtype = -4$, or $mtype = 6$). For these matrices the solver uses a complete supernode pivoting approach. When the factorization algorithm reaches a point where it cannot factorize the supernodes with this pivoting strategy, it uses a pivoting perturbation strategy similar to [Li99], [Schenk04].

The magnitude of the potential pivot is tested against a constant threshold of

$$\alpha = \text{eps} * ||A2||_{\text{inf}},$$

where $\text{eps} = 10^{(-iparm(10))}$, $A2 = P * P_{MPS} * D_r * A * D_c * P$, and $||A2||_{\text{inf}}$ is the infinity norm of the scaled and permuted matrix A . Any tiny pivots encountered during elimination are set to the sign $(l_{II}) * \text{eps} * ||A2||_{\text{inf}}$ - this trades off some numerical stability for the ability to keep pivots from getting too small. Small pivots are therefore perturbed with $\text{eps} = 10^{(-iparm(10))}$.

The default value of $iparm(10)$ is 13 and therefore $\text{eps} = 1.0\text{E-}13$ for unsymmetric matrices ($mtype = 11$ or $mtype = 13$).

The default value of $iparm(10)$ is 8, and therefore $\text{eps} = 1.0\text{E-}8$ for symmetric indefinite matrices ($mtype = -2$, $mtype = -4$, or $mtype = 6$).

$iparm(11)$ - scaling vectors.

PARDISO uses a maximum weight matching algorithm to permute large elements on the diagonal and to scale the matrix so that the diagonal elements are equal to 1 and the absolute values of the off-diagonal entries are less or equal to 1. This scaling method is applied only to unsymmetric matrices ($mtype = 11$ or $mtype = 13$). The scaling can also be used for symmetric indefinite matrices ($mtype = -2$, $mtype = -4$, or $mtype = 6$) when the symmetric weighted matchings are applied ($iparm(13) = 1$).

Use $iparm(11) = 1$ (scaling) and $iparm(13) = 1$ (matching) for highly indefinite symmetric matrices, for example, from interior point optimizations or saddle point

problems. Note that in the analysis phase (*phase*=11) you must provide the numerical values of the matrix *A* in case of scaling and symmetric weighted matching.

The default value of *iparm*(11) is 1 for unsymmetric matrices (*mtype* =11 or *mtype* =13). The default value of *iparm*(11) is 0 for symmetric indefinite matrices (*mtype* =-2, *mtype* =-4, or *mtype* =6).

iparm(12)

This parameter is reserved for future use. Its value must be set to 0.

iparm(13) - improved accuracy using (non-)symmetric weighted matchings.

PARDISO can use a maximum weighted matching algorithm to permute large elements close the diagonal. This strategy adds an additional level of reliability to our factorization methods and can be seen as a complement to the alternative idea of using more complete pivoting techniques during the numerical factorization.

It is recommended to use *iparm*(11) =1 (scalings) and *iparm*(13) =1 (matchings) for highly indefinite symmetric matrices, for example from interior point optimizations or saddle point problems. It is also very important to note that in the analysis phase (*phase* =11) you must provide the numerical values of the matrix *A* in the case of scalings and symmetric weighted matchings.

The default value of *iparm*(13) is 1 for unsymmetric matrices (*mtype* =11 or *mtype* =13). The default value of *iparm*(13) is 0 for symmetric matrices (*mtype* =-2, *mtype* =-4, or *mtype* =6).

iparm(18)

The solver will report the numbers of nonzeros on the factors if *iparm*(18) < 0 on entry.

The default value of *iparm*(18) is -1.

iparm(19) - MFlops of factorization.

If *iparm*(19) < 0 on entry, the solver will report MFlop (1.0E6) that are necessary to factor the matrix *A*. This will increase the reordering time.

The default value of *iparm*(19) is 0.

iparm(21) - pivoting for symmetric indefinite matrices

iparm(21) controls the pivoting method for sparse symmetric indefinite matrices. If *iparm*(21) is 0, then 1x1 diagonal pivoting is used. If *iparm*(21) is 1, then 1x1 and 2x2 Bunch and Kaufman pivoting will be used in the factorization process. If *iparm*(21) is 2, a very robust preprocessing method based on symmetric weighted matchings and 1x1 and 2x2 Bunch and Kaufman pivoting will be used in the factorization process. The default value of *iparm*(21) is 1. Bunch and Kaufman pivoting is available for matrices: *mtype*=-2, *mtype*=-4, or *mtype*=6.

iparm(60)

iparm(60) controls what version of PARDISO - out-of-core (OOC) version or in-core version - is used. The OOC PARDISO can solve very large problems by holding the matrix factors in files on the disk. Because of that the amount of main memory required by OOC PARDISO is significantly reduced.



WARNING. The current OOC version does not use threading, thus both the parameter *iparm*(3) and the variable `MKL_NUM_THREADS` must be set to 1 when *iparm*(60) is equal to 1 or 2.

If *iparm*(60) is set to 0, then the in-core PARDISO is used. If *iparm*(60) is set to 2 - the OOC PARDISO is used. If *iparm*(60) is set to 1 - the in-core PARDISO is used if the total memory (in MBytes) needed for storing the matrix factors is less than the value of the environment variable `MKL_PARDISO_OOC_MAX_CORE_SIZE` (its default value is 2000), and OOC PARDISO is used otherwise. The default value of *iparm*(60) is 0.

Note that if *iparm*(60) is equal to 1 or 2, and the total peak memory needed for strong local arrays is less than `MKL_PARDISO_OOC_MAX_CORE_SIZE`, the program stops with error -9. In this case, increase of `MKL_PARDISO_OOC_MAX_CORE_SIZE` is recommended.

OOO parameters can be set in the configuration file. You can set the path to this file and its name via environmental variable `MKL_PARDISO_OOC_CFG_PATH` and

`MKL_PARDISO_OOC_CFG_FILE_NAME`.

Path and name are as follows:

`<MKL_PARDISO_OOC_CFG_PATH >/<`

`MKL_PARDISO_OOC_CFG_FILE_NAME>` for Linux* OS, and

`<MKL_PARDISO_OOC_CFG_PATH >\<`

`MKL_PARDISO_OOC_CFG_FILE_NAME>` for Windows* OS.

By default, the name of the file is `pardiso_ooc.cfg` and it is placed to the current directory.

All temporary data files can be deleted or stored when the calculations are completed in accordance with the value of the environmental variable `MKL_PARDISO_OOC_KEEP_FILE`. If it is set to 1 (default value) - all files are deleted, if it is set to 0 - all files are stored.

By default, the OOO PARDISO uses the current directory for storing data, and all work arrays associated with the matrix factors are stored in files named `ooc_temp` with different extensions. These default values can be changed by using the environmental variable `MKL_PARDISO_OOC_PATH`.

To set the environmental variables

`MKL_PARDISO_OOC_MAX_CORE_SIZE`,

`MKL_PARDISO_OOC_KEEP_FILE`, and

`MKL_PARDISO_OOC_PATH`, the configuration file should be created with the following lines:

`MKL_PARDISO_OOC_PATH = <path>\ooc_file`

`MKL_PARDISO_OOC_MAX_CORE_SIZE = N`

`MKL_PARDISO_OOC_KEEP_FILE = 0 (or 1)`

where `<path>` - the directory for storing data, `ooc_file` - file name without extension, `N` - memory size in MBytes, it is not recommended to set this value greater than the size of the available RAM (default value - 2000 MBytes).

Alternatively the environment variables can be set via command line:

`export MKL_PARDISO_OOC_PATH = <path>/ooc_file`

`export MKL_PARDISO_OOC_MAX_CORE_SIZE = N`

`export MKL_PARDISO_OOC_KEEP_FILE = 0 (or 1)`

for Linux* OS, and

```
set MKL_PARDISO_OOC_PATH = <path>\ooc_file
set MKL_PARDISO_OOC_MAX_CORE_SIZE = N
set MKL_PARDISO_OOC_KEEP_FILE = 0 (or 1)
for Windows* OS.
```



NOTE. The values specified in a command line have higher priorities - it means that if variable is changed in the configuration file and in the command line, OOC PARDISO uses only value defined in the command line. Setting OOC parameters via command line is recommended.

msglvl

INTEGER

Message level information. If *msglvl* = 0 then PARDISO generates no output, if *msglvl* = 1 the solver prints statistical information to the screen.

b

REAL/COMPLEX

Array, dimension (*n*, *nrhs*). On entry, contains the right hand side vector/matrix *B*. Note that *b* is only accessed in the solution phase.

Output Parameters

pt

This parameter contains internal address pointers.

iparm

On output, some *iparm* values will report useful information, for example, numbers of nonzeros in the factors, and so on.
iparm(7) - number of performed iterative refinement steps. The number of iterative refinement steps that are actually performed during the solve step.
iparm(14) - number of perturbed pivots.
 After factorization, *iparm*(14) contains the number of perturbed pivots during the elimination process for *mtype* = 11, *mtype* = 13, *mtype* = -2, *mtype* = -4, or *mtype* = -6.
iparm(15) - peak memory symbolic factorization.

The parameter *iparm*(15) provides the user with the total peak memory in KBytes that the solver needed during the analysis and symbolic factorization phase. This value is only computed in phase 1.

iparm(16) - permanent memory symbolic factorization.

The parameter *iparm*(16) provides the user with the permanent memory in KBytes that the solver needed from the analysis and symbolic factorization phase in the factorization and solve phases. This value is only computed in phase 1.

iparm(17) - memory numerical factorization and solution.

The parameter *iparm*(17) provides the user with the total double precision memory consumption (KBytes) of the solver for the factorization and solve phases. This value is only computed in phase 2.

Note that the total peak memory solver consumption is

$\max(iparm(15), iparm(16) + iparm(17))$

iparm(18) - number of nonzeros in factors.

The solver will report the numbers of nonzeros on the factors if *iparm*(18) < 0 on entry.

iparm(19) - MFlops of factorization.

Number of operations in MFlop (1.0E6 operations) that are necessary to factor the matrix *A* are returned to the user if *iparm*(19) < 0 on entry.

iparm(20) - CG/CGS diagnostics.

The value is used to give CG/CGS diagnostics (for example, the number of iterations and cause of failure):

If *iparm*(20) > 0, CGS succeeded, and the number of iterations executed are reported in *iparm*(20).

If *iparm*(20) < 0, iterations executed, but CG/CGS failed.

The error report details in *iparm*(20) are of the form:

iparm(20) = - *it_cgs**10 - *cgs_error*.

If *phase* is 23, then the factors *L*, *U* are recomputed for the matrix *A* and the error flag *error* is zero in case of a successful factorization. If *phase* is 33, then *error* = -4 signals the failure.

Description of *cgs_error* is given in the table below:

cgs_error Description

1	- fluctuations of the residuum are too large
2	- $\ dx_{\max_it_cgs}/2\ $ too large (slow convergence)
3	- stopping criterion not reached at $\max\_it\_cgs$
4	- perturbed pivots caused iterative refinement
5	- factorization is too fast for this matrix. It is better to use the factorization method with $iparm(4)=0$

iparm(22) - inertia: number of positive eigenvalues.

The parameter *iparm(22)* reports the number of positive eigenvalues for symmetric indefinite matrices.

iparm(23) - inertia: number of negative eigenvalues.

The parameter *iparm(23)* reports the number of negative eigenvalues for symmetric indefinite matrices.

iparm(24) to *iparm(59)*

These parameters are reserved for future use. Their values must be set to 0.

iparm(61) - the total peak memory in MBytes that the solver used during the analysis and symbolic factorization phases if the in-core PARDISO is used. *iparm(61)* is similar to *iparm(15)*, but *iparm(15)* returns the value of the total peak memory if the OOC PARDISO is used.

iparm(62) - the total double precision memory consumption in MBytes that the solver used during the analysis and symbolic factorization phase in the factorization and solver phases if the in-core PARDISO is used. *iparm(62)* is similar to *iparm(16)*, but *iparm(16)* returns the value of the memory consumption if the OOC PARDISO is used.

iparm(63) - the total double precision memory consumption in MBytes that the solver used for factorization and solution phases if the in-core PARDISO is used. Value of *iparm(63)* is similar to *iparm(17)*, but *iparm(17)* returns the value of the memory consumption if the OOC PARDISO is used.

<i>b</i>	On output, the array is replaced with the solution if $iparm(6) = 1$.
<i>x</i>	REAL/COMPLEX Array, dimension $(n, nrhs)$. On output, contains solution if $iparm(6)=0$. Note that <i>x</i> is only accessed in the solution phase.
<i>error</i>	INTEGER The error indicator according to the below table:
error	Information
0	no error
-1	input inconsistent
-2	not enough memory
-3	reordering problem
-4	zero pivot, numerical factorization or iterative refinement problem
-5	unclassified (internal) error
-6	preordering failed (matrix types 11, 13 only)
-7	diagonal matrix problem
-8	32-bit integer overflow problem
-9	not enough memory for OOC
-10	problems with opening OOC temporary files
-11	read/write problems with the OOC data file

Direct Sparse Solver (DSS) Interface Routines

Intel MKL supports an alternative to the PARDISO* interface for the direct sparse solver referred to here as DSS interface. The DSS interface implements a group of user-callable routines that are used in the step-by-step solving process and exploits the general scheme described in [Linear Solvers Basics](#) for solving sparse systems of linear equations. This interface also includes one routine for gathering statistics related to the solving process and an auxiliary routine for passing character strings from Fortran routines to C routines.

The current implementation of the DSS interface additionally supports the out-of-core (OOC) version.

Table 8-1 lists the names of the routines and describes their general use.

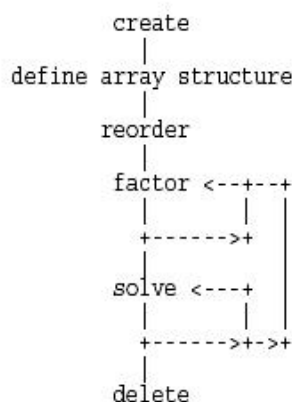
Table 8-1 DSS Interface Routines

Routine	Description
<code>dss_create</code>	Initializes the solver and creates the basic data structures necessary for the solver. This routine must be called before any other DSS routine.
<code>dss_define_structure</code>	Informs the solver of the locations of the non-zero elements of the array.
<code>dss_reorder</code>	Based on the non-zero structure of the matrix, this routine computes a permutation vector to reduce fill-in during the factoring process.
<code>dss_factor_real,</code> <code>dss_factor_complex</code>	Computes the LU , LDT^T or LL^T factorization of a real or complex matrix.
<code>dss_solve_real, dss_solve_complex</code>	Computes the solution vector for a system of equations based on the factorization computed in the previous phase.
<code>dss_delete</code>	Deletes all data structures created during the solving process.
<code>dss_statistics</code>	Returns statistics about various phases of the solving process. Gathers the following statistics: time taken to do reordering, time taken to do factorization, problem solving duration, determinant of a matrix, inertia of a matrix, number of floating point operations taken during factorization, peak memory symbolic factorization, permanent memory symbolic factorization, and memory numerical factorization and solution. This routine can be invoked in any phase of the solving process after the "reorder" phase, but before the "delete" phase. Note that appropriate argument(s) must be supplied to this routine to match the phase in which it is invoked.
<code>mkl_cvt_to_null_terminated_str</code>	Passes character strings from Fortran routines to C routines.

To find a single solution vector for a single system of equations with a single right hand side, the Intel MKL DSS interface routines are invoked in the order in which they are listed in [Table 8-1](#) , with the exception of `dss_statistics`, which is invoked as described in the table.

However, in certain applications it is necessary to produce solution vectors for multiple right-hand sides for a given factorization and/or factor several matrices with the same non-zero structure. Consequently, it is necessary to invoke the Intel MKL sparse routines in an order other than listed in the table. The DSS interface provides such option. The solving process is conceptually divided into six phases, as shown in [Figure 8-2](#) , that indicates the typical order(s) in which the DSS interface routines can be invoked.

Figure 8-2 Typical order for invoking DSS interface routines



See code examples that uses the DSS interface routines to solve systems of linear equations in [Direct Sparse Solver Examples](#) section in the appendix C.

DSS Interface Description

As noted in [Memory Allocation and Handles](#) section, each DSS routine reads or writes an opaque data object called a handle. It is declared as being of type `MKL_DSS_HANDLE` in this documentation. You can refer to [Memory Allocation and Handles](#) to determine the correct method for declaring a handle argument.

All other types in this documentation refer to the common Fortran types, `INTEGER`, `REAL`, `COMPLEX`, `DOUBLE PRECISION`, and `DOUBLE COMPLEX`.

C and C++ programmers should refer to [Calling Direct Sparse Solver and Preconditioner Routines From C/C++](#) section for information on mapping Fortran types to C/C++ types.

Routine Options

The DSS routines have an integer argument (referred below to as *opt*) for passing various options to the routines. The permissible values for *opt* should be specified using only the symbol constants defined in the language-specific header files (see [Implementation Details](#)). The routines accept options for setting the message and termination levels as described in [Table 8-2](#) . Additionally, the routines accept the option `MKL_DSS_DEFAULTS` that establishes the documented default options for each DSS routine.

Table 8-2 Symbolic Names for the Message and Termination Levels Options

Message Level	Termination Level
<code>MKL_DSS_MSG_LVL_SUCCESS</code>	<code>MKL_DSS_TERM_LVL_SUCCESS</code>
<code>MKL_DSS_MSG_LVL_INFO</code>	<code>MKL_DSS_TERM_LVL_INFO</code>
<code>MKL_DSS_MSG_LVL_WARNING</code>	<code>MKL_DSS_TERM_LVL_WARNING</code>
<code>MKL_DSS_MSG_LVL_ERROR</code>	<code>MKL_DSS_TERM_LVL_ERROR</code>
<code>MKL_DSS_MSG_LVL_FATAL</code>	<code>MKL_DSS_TERM_LVL_FATAL</code>

The settings for message and termination levels can be set on any call to a DSS routine. However, once set to a particular level, they remain at that level until they are changed in another call to a DSS routine.

Users can specify multiple options for a DSS routine by adding the options together. For example, to set the message level to `debug` and the termination level to `error` for all the DSS routines, use the following call:

```
CALL dss_create( handle, MKL_DSS_MSG_LVL_INFO + MKL_DSS_TERM_LVL_ERROR)
```

User Data Arrays

Many of the DSS routines take arrays of user data as input. For example, user arrays are passed to the routine `dss_define_structure` to describe the location of the non-zero entries in the matrix. To minimize storage requirements and improve overall run-time efficiency, the Intel MKL DSS routines do not make copies of the user input arrays.



WARNING. The contents of these arrays cannot be modified after they are passed to one of the solver routines.

dss_create

Initializes the solver.

Syntax

```
dss_create(handle, opt)
```

Description

The routine `dss_create` initializes the solver. After the call to `dss_create`, all subsequent invocations of the Intel MKL DSS routines must use the value of `handle` returned by the routine `dss_create`.



WARNING. Do not write the value of `handle` directly.

The default value of the parameter `opt` is

```
MKL_DSS_MSG_LVL_WARNING + MKL_DSS_TERM_LVL_ERROR .
```

This parameter can contain additionally one of two possible values to launch the out-of-core version of DSS (OOC DSS): `MKL_DSS_OOC_VARIABLE` and `MKL_DSS_OOC_STRONG`.

`MKL_DSS_OOC_STRONG` - OOC DSS is used.

`MKL_DSS_OOC_VARIABLE` - OOC DSS uses the in-core kernels of PARDISO* if there is enough memory for storing work arrays associated with the matrix factors in the main memory. Specifically, if the memory needed for the matrix factors is less than the value of the environment variable `MKL_PARDISO_OOC_MAX_CORE_SIZE`, then the OOC version of PARDISO* uses the in-core computations, otherwise it uses the OOC computations.

The variable `MKL_PARDISO_OOC_MAX_CORE_SIZE` defines the maximum size of the main memory allowed for storing work arrays associated with the matrix factors. It is ignored if `MKL_DSS_OOC_STRONG` is set. The default value of `MKL_PARDISO_OOC_MAX_CORE_SIZE` is 2000 MBytes. This value and default path and file name for storing temporary data can be changed using the configuration file `pardiso_ooc.cfg` or command line (See more details in the [PARDISO Parameters](#) description above).



WARNING. Do not change the behavior of the OOC DSS after it is specified in the routine `dss_create`.

Input Parameters

opt INTEGER Options passing argument. The default value is `MKL_DSS_MSG_LVL_WARNING + MKL_DSS_TERM_LVL_ERROR`.

Output Parameters

handle Data object of the `MKL_DSS_HANDLE` type (see [Interface Description](#)).

Return Values

`MKL_DSS_SUCCESS`

`MKL_DSS_INVALID_OPTION`

`MKL_DSS_OUT_OF_MEMORY`

dss_define_structure

Communicates locations of non-zero elements in the matrix to the solver.

Syntax

```
dss_define_structure(handle, opt, rowIndex, nRows, nCols, columns, nNonZeros);
```

Description

The routine `dss_define_structure` communicates the locations of the *nNonZeros* number of non-zero elements in a matrix of *nRows* by *nCols* size to the solver.

Note that currently the Intel MKL DSS software operates only on square matrices, so *nRows* must be equal to *nCols*.

To communicate the locations of non-zeros in the matrix, do the following:

1. Define the general non-zero structure of the matrix by specifying one of the following values for the options argument *opt*:

- `MKL_DSS_SYMMETRIC_STRUCTURE`

- MKL_DSS_SYMMETRIC
- MKL_DSS_NON_SYMMETRIC

2. Provide the actual locations of the non-zeros by means of the arrays *rowIndex* and *columns* (see [Sparse Matrix Storage Format](#)).

Input Parameters

<i>opt</i>	INTEGER. Options passing argument. The default option for the matrix structure is MKL_DSS_SYMMETRIC.
<i>rowIndex</i>	INTEGER. Array of size $\min(nRows, nCols)+1$. Defines the location of non-zero entries in the matrix.
<i>nRows</i>	INTEGER. Number of rows in the matrix.
<i>nCols</i>	INTEGER. Number of columns in the matrix.
<i>columns</i>	INTEGER. Array of size <i>nNonZeros</i> . Defines the location of non-zero entries in the matrix.
<i>nNonZeros</i>	INTEGER. Number of non-zero elements in the matrix.



WARNING.

Pointers to the *rowIndex* and *columns* are stored in the handle. These arrays are used when the solver runs. Do not free or delete them before calling `dss_solve_real` or `dss_solver_complex`.

Output Parameters

<i>handle</i>	Data object of the MKL_DSS_HANDLE type (see Interface Description).

Return Values

MKL_DSS_SUCCESS
 MKL_DSS_STATE_ERR
 MKL_DSS_INVALID_OPTION
 MKL_DSS_COL_ERR

MKL_DSS_NOT_SQUARE
 MKL_DSS_TOO_FEW_VALUES
 MKL_DSS_TOO_MANY_VALUES

dss_reorder

Computes permutation vector that minimizes the fill-in during the factorization phase.

Syntax

`dss_reorder(handle, opt, perm)`

Description

If *opt* contains the option MKL_DSS_AUTO_ORDER, then the routine `dss_reorder` computes a permutation vector that minimizes the fill-in during the factorization phase. For this option, the *perm* array is never accessed.

If *opt* contains the option MKL_DSS_MY_ORDER, then the array *perm* is considered to be a permutation vector supplied by the user. In this case, the array *perm* is of length *nRows*, where *nRows* is the number of rows in the matrix as defined by the previous call to [dss_define_structure](#).

Input Parameters

<i>opt</i>	INTEGER. Option passing argument. The default option for the permutation type is MKL_DSS_AUTO_ORDER.
<i>perm</i>	INTEGER. Array of length <i>nRows</i> . Contains a user-defined permutation vector (accessed only if <i>opt</i> contains MKL_DSS_MY_ORDER).

Output Parameters

<i>handle</i>	Data object of the MKL_DSS_HANDLE type (see Interface Description).

Return Values

MKL_DSS_SUCCESS
 MKL_DSS_STATE_ERR

MKL_DSS_INVALID_OPTION

MKL_DSS_OUT_OF_MEMORY

dss_factor_real, dss_factor_complex

Compute factorization of the matrix with previously specified location.

Syntax

```
dss_factor_real(handle, opt, rValues)
```

```
dss_factor_complex(handle, opt, cValues)
```

Description

These routines compute factorization of the matrix whose non-zero locations were previously specified by a call to [dss_define_structure](#) and whose non-zero values are given in the array *rValues* or *cValues*. The arrays *rValues* and *cValues* are assumed to be of length *nNonZeros* as defined in a previous call to [dss_define_structure](#).

The *opt* argument should contain one of the following options:

- MKL_DSS_POSITIVE_DEFINITE,
- MKL_DSS_INDEFINITE,
- MKL_DSS_HERMITIAN_POSITIVE_DEFINITE,
- MKL_DSS_HERMITIAN_INDEFINITE ,

depending on whether the non-zero values in *rValues* and *cValues* describe a positive definite, indefinite, or Hermitian matrix.

Input Parameters

<i>handle</i>	Data object of the MKL_DSS_HANDLE type (see Interface Description).
<i>opt</i>	INTEGER Options passing argument. The default option for the matrix type is MKL_DSS_POSITIVE_DEFINITE.
<i>rValues</i>	DOUBLE PRECISION. Array of size <i>nNonZeros</i> . Contains real non-zero elements of the matrix.

cValues DOUBLE COMPLEX. Array of size *nNonZeros*. Contains complex non-zero elements of the matrix.

Return Values

MKL_DSS_SUCCESS
 MKL_DSS_STATE_ERR
 MKL_DSS_INVALID_OPTION
 MKL_DSS_OPTION_CONFLICT
 MKL_DSS_OUT_OF_MEMORY
 MKL_DSS_ZERO_PIVOT

dss_solve_real, dss_solve_complex

Compute the corresponding solutions vector and place it in the output array.

Syntax

```
dss_solve_real(handle, opt, rRhsValues, nRhs, rSolValues)
dss_solve_complex(handle, opt, cRhsValues, nRhs, cSolValues)
```

Description

For each right hand side column vector defined in the arrays *rRhsValues* or *cRhsValues*, these routines compute the corresponding solutions vector and place it in the arrays *rSolValues* or *cSolValues*.

The lengths of the right-hand side and solution vectors, *nCols* and *nRows* respectively, are assumed to have been defined in a previous call to [dss_define_structure](#).

Input Parameters

handle Data object of the MKL_DSS_HANDLE type (see [Interface Description](#)).

opt INTEGER. Options passing argument.

nRhs INTEGER. Number of the right-hand sides in the linear equation.

<i>rRhsValues</i>	DOUBLE PRECISION. Array of size <i>nRows</i> by <i>nRhs</i> . Contains real right-hand side vectors.
<i>cRhsValues</i>	DOUBLE COMPLEX. Array of size <i>nRows</i> by <i>nRhs</i> . Contains complex right-hand side vectors.

Output Parameters

<i>rSolValues</i>	DOUBLE PRECISION. Array of size <i>nCols</i> by <i>nRhs</i> . Contains real solution vectors.
<i>cSolValues</i>	DOUBLE COMPLEX. Array of size <i>nCols</i> by <i>nRhs</i> . Contains complex solution vectors.

Return Values

MKL_DSS_SUCCESS
 MKL_DSS_STATE_ERR
 MKL_DSS_INVALID_OPTION
 MKL_DSS_OUT_OF_MEMORY

dss_delete

Deletes all of data structures created during the solutions process.

Syntax

`dss_delete(handle, opt)`

Description

The routine `dss_delete` deletes all data structures created during the solving process.

Input Parameters

<i>opt</i>	INTEGER. Options passing argument. The default value is MKL_DSS_MSG_LVL_WARNING + MKL_DSS_TERM_LVL_ERROR.

Output Parameters

handle Data object of the MKL_DSS_HANDLE type (see [Interface Description](#)).

Return Values

MKL_DSS_SUCCESS

MKL_DSS_INVALID_OPTION

MKL_DSS_OUT_OF_MEMORY

dss_statistics

Returns statistics about various phases of the solving process.

Syntax

```
dss_statistics(handle, opt, statArr, retValues)
```

Description

The `dss_statistics` routine returns statistics about various phases of the solving process. This routine gathers the following statistics:

- time taken to do reordering,
- time taken to do factorization,
- problem solving duration,
- determinant of a matrix,
- inertia of a matrix,
- number of floating point operations taken during factorization,
- total peak memory needed during the analysis and symbolic factorization,
- permanent memory needed from the analysis and symbolic factorization,
- memory consumption for the factorization and solve phases.

Statistics are returned in accordance with the input string specified by the parameter *statArr*. The value of the statistics is returned in double precision in a return array allocated by user.

For multiple statistics, multiple string constants separated by commas can be used as input. Return values are put into the return array in the same order as specified in the input string.

Statistics must be requested only at the appropriate stages of the solving process. For example, inquiring about `FactorTime` before a matrix is factored leads to errors.

The following table shows the point at which each individual statistics item can be called:

Table 8-3 Statistics Calling Sequences

Type of Statistics	When to Call
<code>ReorderTime</code>	After <code>dss_reorder</code> is completed successfully.
<code>FactorTime</code>	After <code>dss_factor_real</code> or <code>dss_factor_complex</code> is completed successfully.
<code>SolveTime</code>	After <code>dss_solve_real</code> or <code>dss_solve_complex</code> is completed successfully.
<code>Determinant</code>	After <code>dss_factor_real</code> or <code>dss_factor_complex</code> is completed successfully.
<code>Inertia</code>	After <code>dss_factor_real</code> is completed successfully and the matrix is real, symmetric, and indefinite.
<code>Flops</code>	After <code>dss_factor_real</code> or <code>dss_factor_complex</code> is completed successfully.
<code>Peakmem</code>	After <code>dss_reorder</code> is completed successfully.
<code>Factormem</code>	After <code>dss_reorder</code> is completed successfully.
<code>Solvemem</code>	After <code>dss_factor_real</code> or <code>dss_factor_complex</code> is completed successfully.

Input Parameters

<i>handle</i>	Data object of the <code>MKL_DSS_HANDLE</code> type (see Interface Description).								
<i>opt</i>	INTEGER. Options passing argument.								
<i>statArr</i>	STRING. Input string that defines the type of the returned statistics. The parameter can include one or more of the following string constants (case of the input string has no effect): <table> <tr> <td><code>ReorderTime</code></td><td>Amount of time taken to do the reordering.</td></tr> <tr> <td><code>FactorTime</code></td><td>Amount of time taken to do the factorization.</td></tr> <tr> <td><code>SolveTime</code></td><td>Amount of time taken to solve the problem after factorization.</td></tr> <tr> <td><code>Determinant</code></td><td>Determinant of the matrix <i>A</i>.</td></tr> </table>	<code>ReorderTime</code>	Amount of time taken to do the reordering.	<code>FactorTime</code>	Amount of time taken to do the factorization.	<code>SolveTime</code>	Amount of time taken to solve the problem after factorization.	<code>Determinant</code>	Determinant of the matrix <i>A</i> .
<code>ReorderTime</code>	Amount of time taken to do the reordering.								
<code>FactorTime</code>	Amount of time taken to do the factorization.								
<code>SolveTime</code>	Amount of time taken to solve the problem after factorization.								
<code>Determinant</code>	Determinant of the matrix <i>A</i> .								

	For real matrices: the determinant is returned as <code>det_pow</code>, <code>det_base</code> in two consecutive return array locations, where $1.0 \leq \text{abs}(\text{det\_base}) < 10.0$ and $\text{determinant} = \text{det\_base} * 10^{(\text{det\_pow})}$. For complex matrices: the determinant is returned as <code>det_pow</code>, <code>det_re</code>, <code>det_im</code> in three consecutive return array locations, where $1.0 \leq \text{abs}(\text{det\_re}) + \text{abs}(\text{det\_im}) < 10.0$ and $\text{determinant} = \text{det\_re} + \text{det\_im} * 10^{(\text{det\_pow})}$.
Inertia	Inertia of a real symmetric matrix is defined as a triplet of nonnegative integers (p, n, z), where p is the number of positive eigenvalues, n is the number of negative eigenvalues, and z is the number of zero eigenvalues. <code>Inertia</code> is returned as three consecutive return array locations as p, n, z. Computing <code>Inertia</code> is recommended only for stable matrices. Unstable matrices can lead to incorrect results. <code>Inertia</code> of a k-by-k real symmetric positive definite matrix is always $(k, 0, 0)$. Therefore <code>Inertia</code> is returned only in cases of real symmetric indefinite matrices. For all other matrix types, an error message is returned.
Flops	Number of floating point operations performed during the factorization.
Peakmem	Total peak memory in KBytes that the solver needs during the analysis and symbolic factorization phase.
Factormem	Permanent memory in KBytes that the solver needs from the analysis and symbolic factorization phase in the factorization and solve phases.

Solvemem

Total double precision memory consumption (KBytes) of the solver for the factorization and solve phases.



NOTE. To avoid problems in passing strings from Fortran to C, Fortran users must call the `mkl_cvt_to_null_terminated_str` routine before calling `dss_statistics`. Refer to the description of [mkl_cvt_to_null_terminated_str](#) for details.

Output Parameters

retValues

DOUBLE PRECISION. Value of the statistics returned.

Finding “time used to reorder” and “inertia” of a matrix.

The example below illustrates the use of the `dss_statistics` routine.

To find the above values, call `dss_statistics(handle, opt, statArr, retValues)`, where *statArr* is “ReorderTime,Inertia”

In this example, *retValues* has the following values:

Return Values

MKL_DSS_SUCCESS

MKL_DSS_STATISTICS_INVALID_MATRIX

MKL_DSS_STATISTICS_INVALID_STATE

MKL_DSS_STATISTICS_INVALID_STRING

mkl_cvt_to_null_terminated_str

Passes character strings from Fortran routines to C routines.

Syntax

```
mkl_cvt_to_null_terminated_str (destStr, destLen, srcStr)
```


Description

The routine `mkl_cvt_to_null_terminated_str` passes character strings from Fortran routines to C routines. The strings are converted into integer arrays before being passed to C. Using this routine avoids the problems that can occur on some platforms when passing strings from Fortran to C. The use of this routine is highly recommended.

Input Parameters

<i>destLen</i>	INTEGER. Length of the output array <i>destStr</i> .
<i>srcStr</i>	STRING. Input string.

Output Parameters

<i>destStr</i>	INTEGER. One-dimensional array of integer.

Implementation Details

Several aspects of the Intel MKL DSS interface are platform-specific and language-specific. To promote portability across platforms and ease of use across different languages, one of the following Intel MKL DSS language-specific header files can be included:

- `mkl_dss.f77` for F77 programs
- `mkl_dss.f90` for F90 programs
- `mkl_dss.h` for C programs

These header files define symbolic constants for error returns, function options, certain defined data types, and function prototypes.



NOTE. Constants for options, error returns, and message severities must be referred only by the symbolic names that are defined in these header files. Use of the Intel MKL DSS software without including one of the above header files is not supported.

Memory Allocation and Handles

To simplify the use of the Intel MKL DSS routines, they do not require the user to allocate any temporary working storage. The solver itself allocates any required storage. To enable multiple users to access the solver simultaneously, the solver keeps track of the storage allocated for a particular application by using an opaque data object called a *handle*.

Each of the Intel MKL DSS routines creates, uses or deletes a handle. Consequently, user programs must be able to allocate storage for a handle. The exact syntax for allocating storage for a handle varies from language to language. To standardize the handle declarations, the language-specific header files declare constants and defined data types that should be used when declaring a handle object in the user code.

Fortran 90 programmers should declare a handle as:

```
INCLUDE "mkl_dss.f90"

TYPE (MKL_DSS_HANDLE) handle
```

C and C++ programmers should declare a handle as:

```
#include "mkl_dss.h"

_MKL_DSS_HANDLE_t handle;
```

FORTRAN 77 programmers using compilers that support eight byte integers, should declare a handle as:

```
INCLUDE "mkl_dss.f77"

INTEGER*8 handle
```

Otherwise they should replace the `INTEGER*8` data types with the `DOUBLE PRECISION` data type.

In addition to the definition for the correct declaration of a handle, the include file also defines the following:

- function prototypes for languages that support prototypes
- symbolic constants that are used for the error returns
- user options for the solver routines
- message severity.

Iterative Sparse Solvers based on Reverse Communication Interface (RCI ISS)

Intel MKL supports the iterative sparse solvers (ISS) based on the reverse communication interface (RCI), referred to here as . The RCI ISS interface implements a group of user-callable routines that are used in the step-by-step solving process of a symmetric positive definite

system (RCI Conjugate Gradient Solver, or RCI CG), and of a non-symmetric indefinite (non-degenerate) system (RCI Flexible Generalized Minimal RESidual Solver, or RCI FGMRES) of linear algebraic equations and exploits the general RCI scheme described in [Dong95].

See the [Linear Solvers Basics](#) for discussion of terms and concepts related to the ISS routines.

RCI means that when the solver needs the results of certain operations (for example, matrix-vector multiplications), the user himself must perform them and pass the result to the solver. This gives the great universality to the solver as it is independent of the specific implementation of the operations like the matrix-vector multiplication. To perform such operations, the user can use the built-in sparse matrix-vector multiplications and triangular solvers routines (see [Sparse BLAS Level 2 and Level 3](#)).



NOTE. The RCI CG solver is implemented in two versions: for system of equations with single right hand side, and for system of equations with multiple right hand sides.

The CG method may fail to compute the solution or compute the wrong solution if the matrix of the system is not symmetric and positive definite.

The FGMRES method may fail if the matrix is degenerate.

Table 8-4 lists the names of the routines, and describes their general use.

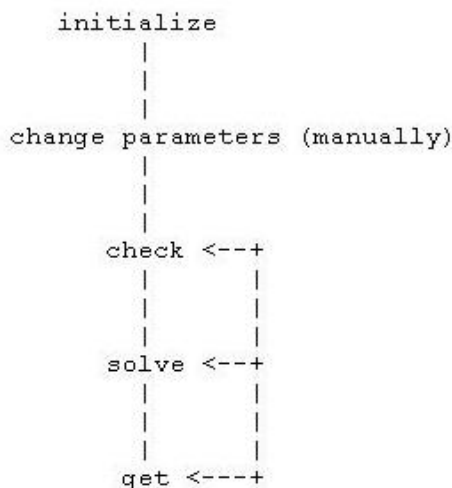
Table 8-4 RCI ISS Interface Routines

Routine	Description
<code>dcg_init</code> , <code>dcgmrhs_init</code> , <code>dfgmres_init</code>	Initializes the solver.
<code>dcg_check</code> , <code>dcgmrhs_check</code> , <code>dfgmres_check</code>	Checks the consistency and correctness of the user defined data.
<code>dcg</code> , <code>dcgmrhs</code> , <code>dfgmres</code>	Computes the approximate solution vector.
<code>dcg_get</code> , <code>dcgmrhs_get</code> , <code>dfgmres_get</code>	Retrieves the number of the current iteration.

The Intel MKL RCI ISS interface routines are normally invoked in the order listed in [Table8-4](#), with the exception of `dcg_get`, `dcgmrhs_get`, and `dfgmres_get` routines that can be invoked at any place in the code. However, in this case some precautions should be taken to avoid the wrong results. Advanced users can change that order if they need it. Others should follow the above order of calls.

The following diagram in [Figure 8-3](#) indicates the typical order in which the RCI ISS interface routines can be invoked.

Figure 8-3 Typical Order for Invoking RCI ISS interface Routines



The following pseudocode shows the general schemes of using the RCI CG routines.

...

generate matrix *A*

generate preconditioner *C* (optional)

call `dcg_init(n, x, b, RCI_request, ipar, dpar, tmp)`

change parameters in *ipar*, *dpar* if necessary

call `dcg_check(n, x, b, RCI_request, ipar, dpar, tmp)`

1 call `dcg(n, x, b, RCI_request, ipar, dpar, tmp)`

if (*RCI_request*.eq.1) then

multiply the matrix *A* by *tmp*(1:*n*,1) and put the result in *tmp*(1:*n*,2)

It is possible to use [MKL Sparse BLAS Level 2](#) subroutines for this operation

c proceed with CG iterations

goto 1

```

endif
if (RCI_request.eq.2)then
  do the stopping test
  if (test not passed) then
c  proceed with CG iterations
    go to 1
  else
c  stop CG iterations
    goto 2
  endif
endif
if (RCI_request.eq.3) then (optional)
  apply the preconditioner  $C$  inverse to  $tmp(1:n,3)$  and put the result in  $tmp(1:n,4)$ 
c  proceed with CG iterations
  goto 1
end
2 call dcg_get( $n, x, b, RCI\_request, ipar, dpar, tmp, itercount$ )
  current iteration number is in  $itercount$ 
  the computed approximation is in the array  $x$ 

```

And the following pseudocode shows the general schemes of using the RCI FGMRES routines.

...

generate matrix A

generate preconditioner C (optional)

```

  call dfgmres_init( $n, x, b, RCI\_request, ipar, dpar, tmp$ )
  change parameters in  $ipar, dpar$  if necessary
  call dfgmres_check( $n, x, b, RCI\_request, ipar, dpar, tmp$ )
1 call dfgmres( $n, x, b, RCI\_request, ipar, dpar, tmp$ )
  if ( $RCI\_request.eq.1$ ) then
    multiply the matrix  $A$  by  $tmp(ipar(22))$  and put the result in  $tmp(ipar(23))$ 

```

It is possible to use [MKL Sparse BLAS Level 2](#) subroutines for this operation

```

c  proceed with FGMRES iterations
    goto 1
endif
if (RCI_request.eq.2) then
    do the stopping test
    if (test not passed) then
c  proceed with FGMRES iterations
        go to 1
    else
c  stop FGMRES iterations
        goto 2
    endif
endif
if (RCI_request.eq.3) then (optional)
apply the preconditioner  $C$  inverse to  $tmp(ipar(22))$  and put the result in  $tmp(ipar(23))$ 
c  proceed with FGMRES iterations
    goto 1
endif
if (RCI_request.eq.4) then
check the norm of the next orthogonal vector, it is contained in  $dpar(7)$ 
    if (the norm is not zero up to rounding/computational errors) then
c  proceed with FGMRES iterations
        goto 1
    else
c  stop FGMRES iterations
        goto 2
    endif
endif
endif

```

2 call dfgmres_get(*n*, *x*, *b*, *RCI_request*, *ipar*, *dpar*, *tmp*, *itercount*)

current iteration number is in *itercount*

the computed approximation is in the array *x*

Note that for the FGMRES method the array *x* initially contains the current initial approximation to the solution that can be updated only by calling the routine `dfgmres_get`. It updates the solution in accordance with the computations performed by the routine `dfgmres`.

The above pseudocodes demonstrate two main differences in the use of RCI CG and RCI FGMRES interfaces. The first difference relates to *RCI_request*=3 (different locations in the *tmp* array, which is 2-dimensional for CG and 1-dimensional for FGMRES). The second difference relates to *RCI_request*=4 (the RCI CG interface never produces *RCI_request*=4).

Code examples that use the RCI ISS interface routines to solve systems of linear equations can be found in the [Iterative Sparse Solver Code Example](#) section in the Appendix C.

CG Interface Description

All types in this documentation refer to the common Fortran types, `INTEGER`, and `DOUBLE PRECISION`.

C and C++ programmers should refer to the section [Calling Sparse Solver and Preconditioner Routines From C/C++](#) for information on mapping Fortran types to C/C++ types.

Each routine for the RCI CG solver is implemented in two versions: for a system of equations with a single right hand side (SRHS), and for a system of equations with multiple right hand sides (MRHS). The names of routines for a system with MRHS contain the suffix `mrhs`.



NOTE. The routines for a system with MRHS can be used with the names without the suffix `mrhs`. To do this, the user must switch on the compiler preprocessor and include the files `mkl_solver.h` for C/C++, or `mkl_solver.f77` for FORTRAN.

Routines Options

All of the RCI CG routines have parameters for passing various options to the routines. The values for these parameters should be specified very carefully (see [CG Common Parameters](#)), and they can be changed during computations according to the user's needs.



NOTE. Provide correct and consistent parameters to the subroutines to avoid fails or wrong results.

User Data Arrays

Many of the RCI CG routines take arrays of user data as input. For example, user arrays are passed to the routine `dcg` to compute the solution of a system of linear algebraic equations. The Intel MKL RCI CG routines do not make copies of the user input arrays to minimize storage requirements and improve overall run-time efficiency.

CG Common Parameters



NOTE. The default and initial values listed below are assigned to the parameters by calling the `dcg_init/dcgmrhs_init` routine.

<i>n</i>	- INTEGER, this parameter sets the size of the problem in the <code>dcg_init/dcgmrhs_init</code> routine. All the other routines uses the <code>ipar(1)</code> parameter instead.
<i>x</i>	- DOUBLE PRECISION array of size <i>n</i> for SRHS, or matrix of size <i>n</i> -by- <i>nrhs</i> for MRHS. This parameter contains the current approximation to the solution. Before the first call to the <code>dcg/dcgmrhs</code> routine, it contains the initial approximation to the solution.
<i>nrhs</i>	- INTEGER, this parameter sets the number of right-hand sides for MRHS routines.
<i>b</i>	- DOUBLE PRECISION array containing a single right-hand side vector, or matrix (<i>nrhs</i> , <i>n</i>) containing right-hand side vectors.
<i>RCI_request</i>	- INTEGER, this parameter informs about the result of work of the RCI CG routines. The negative values of the parameter indicate that the routine is completed with errors or warnings. The 0 value indicates the successful completion of the task. The positive values mean that the user must perform certain actions, specifically: <i>RCI_request</i>= 1 - to multiply the matrix by <i>tmp</i> (1:<i>n</i>,1), put the result in <i>tmp</i>(1:<i>n</i>,2), and return the control to the <code>dcg/dcgmrhs</code> routine; <i>RCI_request</i>= 2 - to perform the stopping tests. If they fail, return the control to the <code>dcg/dcgmrhs</code> routine. Otherwise, the solution is found and stored in the <i>x</i> array;

RCI_request= 3 - for SRHS: to apply the preconditioner to *tmp*(1:*n*,3), put the result in *tmp*(1:*n*,4), and return the control to the *dcg* routine;
 - for MRHS: to apply the preconditioner to *tmp*(:,3+*ipar*(3)), put the result in *tmp*(:,3), and return the control to the *dcgmrhs* routine.

Note that the *dcg_get*/*dcgmrhs_get* routine does not change the parameter *RCI_request*. This enables to use this routine inside the Reverse Communication computations.

ipar

- INTEGER array, of size 128 for SRHS, and of size (128+2**nrhs*) for MRHS; this parameter specifies the integer set of data for the RCI CG computations:

ipar(1) - specifies the size of the problem. The *dcg_init*/*dcgmrhs_init* routine assigns *ipar*(1)=*n*. All the other routines uses this parameter instead of *n*. There is no default value for this parameter.

ipar(2) - specifies the type of output for error and warning messages generated by the RCI CG routines. The default value 6 means that all messages are displayed on the screen. Otherwise the error and warning messages are written to the newly created files *dcg_errors.txt* and *dcg_check_warnings.txt* respectively. Note that if *ipar*(6) and *ipar*(7) parameters are set to 0, error and warning messages are not generated at all.

ipar(3) - for SRHS: contains the current stage of the RCI CG computations, the initial value is 1;
 - for MRHS: contains the right-hand side for which the calculations are currently performed.



WARNING. Avoid altering this variable during computations.

<i>ipar</i> (4)	- contains the current iteration number, the initial value is 0.
<i>ipar</i> (5)	- specifies the maximum number of iterations, the default value is $\min\{150, n\}$.
<i>ipar</i> (6)	- if the value is not equal to 0, the routines output error messages in accordance with the parameter <i>ipar</i> (2). Otherwise, the routines do not output error messages at all, but return a negative value of the parameter <i>RCI_request</i> . The default value is 1.
<i>ipar</i> (7)	- if the value is not equal to 0, the routines output warning messages in accordance with the parameter <i>ipar</i> (2). Otherwise, the routines do not output warning messages at all, but they return a negative value of the parameter <i>RCI_request</i> . The default value is 1.
<i>ipar</i> (8)	- if the value is not equal to 0, the <i>dcg/dcgmrhs</i> routine performs the stopping test for the maximum number of iterations, namely, $ipar(4) \leq ipar(5)$. Otherwise, the method is stopped and the corresponding value is assigned to the <i>RCI_request</i> . If the value is 0, the routine does not perform this stopping test. The default value is 1.
<i>ipar</i> (9)	- if the value is not equal to 0, the <i>dcg/dcgmrhs</i> routine performs the residual stopping test, namely, $dpar(5) \leq dpar(4) = dpar(1) * dpar(3) + dpar(2)$. Otherwise, the method is stopped and corresponding value is assigned to the <i>RCI_request</i> . If the value is 0, the routine does not perform this stopping test. The default value is 0.
<i>ipar</i> (10)	- if the value is not equal to 0, the <i>dcg/dcgmrhs</i> routine requests a user-defined stopping test by setting the output parameter

RCI_request=2. If the value is 0, the routine does not perform the user defined stopping test. The default value is 1.



NOTE. At least one of the parameters *ipar*(8)-*ipar*(10) must be set to 1.

ipar(11) - if the value is equal to 0, the *dcg/dcgmrhs* routine runs the non-preconditioned version of the corresponding Conjugate Gradient method. Otherwise, the routine runs the preconditioned version of the Conjugate Gradient method, and requests the user to perform the preconditioning step by setting the output parameter *RCI_request*=3. The default value is 0.

ipar(11:128), are reserved and not used in the current RCI
ipar(11:128+2**nrhs*) CG SRHS and MRHS routines respectively.



NOTE. Advanced users can define the array in the code using RCI CG SRHS as `INTEGER ipar(11)`. However, to guarantee compatibility with the future releases of Intel MKL, declaring the array *ipar* with length 128 is highly recommended.

dpar - DOUBLE PRECISION array, for SRHS of size 128, for MRHS of size (128+2**nrhs*); this parameter is used to specify the double precision set of data for the RCI CG computations, specifically:

dpar(1) - specifies the relative tolerance. The default value is 1.0D-6.

dpar(2) - specifies the absolute tolerance. The default value is 0.0D-0.

<code>dpar(3)</code>	- specifies the square norm of the initial residual (if it is computed in the <code>dcg/dcgmrhs</code> routine). The initial value is 0.
<code>dpar(4)</code>	- service variable equal to $dpar(1) * dpar(3) + dpar(2)$ (if it is computed in the <code>dcg/dcgmrhs</code> routine). The initial value is 0.
<code>dpar(5)</code>	- specifies the square norm of the current residual. The initial value is 0.0.
<code>dpar(6)</code>	- specifies the square norm of residual from the previous iteration step (if available). The initial value is 0.0.
<code>dpar(7)</code>	- contains the <i>alpha</i> parameter of the CG method. The initial value is 0.0.
<code>dpar(8)</code>	- contains the <i>beta</i> parameter of the CG method, it is equal to $dpar(5) / dpar(6)$. The initial value is 0.0.
<code>dpar(9:128)</code> , <code>dpar(9:128+2*nrhs)</code>	are reserved and not used in the current RCI CG SRHS and MRHS routines respectively.



NOTE. Advanced users can define this array in the code using RCI CG SRHS as `DOUBLE PRECISION dpar(8)`. However, to guarantee compatibility with the future releases of Intel MKL, declaring the array `dpar` with length 128 is highly recommended.

`tmp`

- DOUBLE PRECISION array of size $(n, 4)$ for SRHS, and $(n, 3+nrhs)$ for MRHS. This parameter is used to supply the double precision temporary space for the RCI CG computations, specifically:

<code>tmp(:, 1)</code>	- specifies the current search direction. The initial value is 0.0.

<code>tmp(:, 2)</code>	- contains the matrix multiplied by the current search direction. The initial value is 0.0.
<code>tmp(:, 3)</code>	- contains the current residual. The initial value is 0.0.
<code>tmp(:, 4)</code>	- contains the inverse of the preconditioner applied to the current residual. There is no initial value for this parameter.

 **NOTE.** Advanced users can define this array in the code using RCI CG SRHS as `DOUBLE PRECISION tmp(n, 3)` if they run only non-preconditioned CG iterations.

FGMRES Interface Description

All types in this documentation refer to the common Fortran types, `INTEGER`, and `DOUBLE PRECISION`.

C and C++ programmers should refer to the section [Calling Sparse Solver Routines From C/C++](#) for information on mapping Fortran types to C/C++ types.

Routines Options

All of the RCI FGMRES routines have parameters for passing various options to the routines. The values for these parameters should be specified very carefully (see [FGMRES Common Parameters](#)), and they can be changed during computations according to the user's needs.

 **NOTE.** Provide correct and consistent parameters to the subroutines to avoid fails or wrong results.

User Data Arrays

Many of the RCI FGMRES routines take arrays of user data as input. For example, user arrays are passed to the routine `dfgmres` to compute the solution of a system of linear algebraic equations. To minimize storage requirements and improve overall run-time efficiency, the Intel MKL RCI FGMRES routines do not make copies of the user input arrays.

FGMRES Common Parameters



NOTE. The default and initial values listed below are assigned to the parameters by calling the `dfgmres_init` routine.

<i>n</i>	- INTEGER, this parameter sets the size of the problem in the <code>dfgmres_init</code> routine. All the other routines uses <code>ipar(1)</code> parameter instead.
<i>x</i>	- DOUBLE PRECISION array, this parameter contains the current approximation to the solution vector. Before the first call to the <code>dfgmres</code> routine, it contains the initial approximation to the solution vector.
<i>b</i>	- DOUBLE PRECISION array, this parameter contains the right-hand side vector. Depending on user requests, it may later contain the approximate solution.
<i>RCI_request</i>	- INTEGER, this parameter informs about the result of work of the RCI FGMRES routines. The negative values of the parameter indicate that the routine is completed with errors or warnings. The 0 value indicates the successful completion of the task. The positive values mean that the user must perform certain actions, specifically: <i>RCI_request</i> = 1 - multiply the matrix by <code>tmp(ipar(22))</code>, put the result in <code>tmp(ipar(23))</code>, and return the control to the <code>dfgmres</code> routine; <i>RCI_request</i> = 2 - perform the stopping tests. If they fail, return the control to the <code>dfgmres</code> routine. Otherwise, the solution can be updated by a subsequent call to <code>dfgmres_get</code> routine;

	<i>RCI_request</i> = 3	- apply the preconditioner to <i>tmp(ipar(22))</i> , put the result in <i>tmp(ipar(23))</i> , and return the control to the <i>dfgmres</i> routine.
	<i>RCI_request</i> = 4	- check if the norm of the current orthogonal vector is not zero within the rounding/computational errors. Return the control to the <i>dfgmres</i> routine if it is not zero, otherwise complete the solution process by calling <i>dfgmres_get</i> routine.
<i>ipar</i> (128)		- INTEGER array, this parameter specifies the integer set of data for the RCI FGMRES computations:
<i>ipar</i> (1)		- specifies the size of the problem. The <i>dfgmres_init</i> routine assigns <i>ipar</i> (1)= <i>n</i> . All the other routines uses this parameter instead of <i>n</i> . There is no default value for this parameter.
<i>ipar</i> (2)		- specifies the type of output for error and warning messages that are generated by the RCI FGMRES routines. The default value 6 means that all messages are displayed on the screen. Otherwise the error and warning messages are written to the newly created file <i>MKL_RCI_FGMRES_Log.txt</i> . Note that if <i>ipar</i> (6) and <i>ipar</i> (7) parameters are set to 0, error and warning messages are not generated at all.
<i>ipar</i> (3)		- contains the current stage of the RCI FGMRES computations, the initial value is 1.
		 WARNING. Avoid altering this variable during computations.
<i>ipar</i> (4)		- contains the current iteration number. The initial value is 0.

<i>ipar</i> (5)	- specifies the maximum number of iterations. The default value is $\min\{150, n\}$.
<i>ipar</i> (6)	- if the value is not equal to 0, the routines output error messages in accordance with the parameter <i>ipar</i> (2). Otherwise, the routines do not output error messages at all, but return a negative value of the parameter <i>RCI_request</i> . The default value is 1.
<i>ipar</i> (7)	- if the value is not equal to 0, the routines output warning messages in accordance with the parameter <i>ipar</i> (2). Otherwise, the routines do not output warning messages at all, but they return a negative value of the parameter <i>RCI_request</i> . The default value is 1.
<i>ipar</i> (8)	- if the value is not equal to 0, the <i>dfgmres</i> routine performs the stopping test for the maximum number of iterations, namely, $ipar(4) \leq ipar(5)$. Otherwise, the method is stopped and the corresponding value is assigned to the <i>RCI_request</i> . If the value is 0, the <i>dfgmres</i> routine does not perform this stopping test. The default value is 1.
<i>ipar</i> (9)	- if the value is not equal to 0, the <i>dfgmres</i> routine performs the residual stopping test, namely, $dpar(5) \leq dpar(4) = dpar(1) \cdot dpar(3) + dpar(2)$. If the criterion is met, the method is stopped and corresponding value is assigned to the output parameter <i>RCI_request</i> . If the value is 0, the <i>dfgmres</i> routine does not perform this stopping test. The default value is 0.
<i>ipar</i> (10)	- if the value is not equal to 0, the <i>dfgmres</i> routine requests for the user defined stopping test by setting <i>RCI_request</i> =2. If the value

is 0, the `dfgmres` routine does not perform the user defined stopping test. The default value is 1.



NOTE. At least one of the parameters $ipar(8)$ - $ipar(10)$ must be set to 1.

<code>ipar(11)</code>	- if the value is equal to 0, the <code>dfgmres</code> routine runs the non-preconditioned version of the FGMRES method. Otherwise, the routine runs the preconditioned version of the FGMRES method, and requests the user to perform the preconditioning step by setting the output parameter <code>RCI_request=3</code> . The default value is 0.
<code>ipar(12)</code>	- if the value is not equal to 0, the <code>dfgmres</code> routine performs the automatic test for zero norm of the currently generated vector, namely, $dpar(7) \leq dpar(8)$, where $dpar(8)$ contains the tolerance value. Otherwise, the routine requests the user to perform this check by setting the output parameter <code>RCI_request=4</code> . The default value is 0.
<code>ipar(13)</code>	- if the value is equal to 0, the <code>dfgmres_get</code> routine updates the solution to the vector x according to the computations done by the <code>dfgmres</code> routine. If the value is positive, the routine writes the solution to the right hand side vector b . If the value is negative, the routine returns only the number of the current iteration, and does not update the solution. The default value is 0



NOTE. Advanced users may use the `dfgmres_get` routine at any place in the code. In this case special attention should be paid to the parameter `ipar(13)`. The RCI FGMRES iterations can be continued after the call to `dfgmres_get` routine only if the parameter `ipar(13)` is not equal to zero. If `ipar(13)` is positive, then the updated solution overwrites the right hand side in the vector *b*. If the user wants to run the restarted version of FGMRES with the same right hand side, then it must be saved in a different memory location before the first call to the `dfgmres_get` routine with positive `ipar(13)`.

`ipar(14)`

- contains the internal iteration counter that counts the number of iterations before the restart takes place. The initial value is 0.



WARNING. Avoid altering this variable during computations.

`ipar(15)`

- specifies the length of the non-restarted FGMRES iterations. To run the restarted version of the FGMRES method, the user must assign the number of iterations to `ipar(15)` before the restart. The default value is $\min\{150, n\}$, that is, by default the non-restarted version of FGMRES method is used.

`ipar(16)`

- service variable, specifies the location of the rotated Hessenberg matrix from which the matrix stored in the packed format (see [Matrix Arguments](#) in the Appendix B for details) is started in the *tmp* array.

<i>ipar</i> (17)	- service variable, specifies the location of the rotation cosines from which the vector of cosines is started in the <i>tmp</i> array.
<i>ipar</i> (18)	- service variable, specifies the location of the rotation sines from which the vector of sines is started in the <i>tmp</i> array.
<i>ipar</i> (19)	- service variable, specifies the location of the rotated residual vector from which the vector is started in the <i>tmp</i> array.
<i>ipar</i> (20)	- service variable, specifies the location of the least squares solution vector from which the vector is started in the <i>tmp</i> array.
<i>ipar</i> (21)	- service variable, specifies the location of the set of preconditioned vectors from which the set is started in the <i>tmp</i> array. The memory locations in the <i>tmp</i> array starting from <i>ipar</i> (21) are used only for the preconditioned FGMRES method.
<i>ipar</i> (22)	- specifies the memory location from which the first vector (source) used in operations requested via <i>RCI_request</i> is started in the <i>tmp</i> array.
<i>ipar</i> (23)	- specifies the memory location from which the second vector (source) used in operations requested via <i>RCI_request</i> is started in the <i>tmp</i> array.
<i>ipar</i> (24:128)	are reserved and not used in the current RCI FGMRES routines.



NOTE. Advanced users can define the array in the code as `INTEGER ipar(23)`. However, to guarantee compatibility with the future releases of Intel MKL, declaring the array *ipar* with length 128 is highly recommended.

<i>dpar</i> (128)	- DOUBLE PRECISION array, this parameter specifies the double precision set of data for the RCI CG computations, specifically:
<i>dpar</i> (1)	- specifies the relative tolerance. The default value is 1.0D-6.
<i>dpar</i> (2)	- specifies the absolute tolerance. The default value is 0.0D-0.
<i>dpar</i> (3)	- specifies the Euclidean norm of the initial residual (if it is computed in the <i>dfgmres</i> routine). The initial value is 0.0.
<i>dpar</i> (4)	- service variable equal to $dpar(1) * dpar(3) + dpar(2)$ (if it is computed in the <i>dfgmres</i> routine). The initial value is 0.0.
<i>dpar</i> (5)	- specifies the Euclidean norm of the current residual. The initial value is 0.0.
<i>dpar</i> (6)	- specifies the Euclidean norm of residual from the previous iteration step (if available). The initial value is 0.0.
<i>dpar</i> (7)	- contains the norm of the generated vector. The initial value is 0.0.
 NOTE. For reference only: in terms of [Saad03] this parameter is the coefficient $h_{k+1,k}$ of the Hessenberg matrix.	
<i>dpar</i> (8)	- contains the tolerance for the zero norm of the currently generated vector. The default value is 1.0D-12.
<i>dpar</i> (9:128)	are reserved and not used in the current RCI FGMRES routines.



NOTE. Advanced users can define this array in the code as follows: `DOUBLE PRECISION dpar(8)`. However, to guarantee compatibility with the future releases of Intel MKL, declaring the array `dpar` with length 128 is highly recommended.

tmp

-DOUBLE PRECISION array of size $((2*ipar(15)+1)*n + ipar(15)*(ipar(15)+9)/2 + 1))$, this parameter is used to supply the double precision temporary space for the RCI FGMRES computations, specifically:

tmp(1:ipar(16)-1) - contains the sequence of generated by the FGMRES method vectors. The initial value is 0.0 for the first part of this memory of length *n*;

tmp(ipar(16):ipar(17)-1) contains the rotated Hessenberg matrix generated by the FGMRES method; the matrix is stored in the packed format. There is no initial value for this part of *tmp* array.

tmp(ipar(17):ipar(18)-1) - contains the rotation cosines vector generated by the FGMRES method. There is no initial value for this part of *tmp* array.

tmp(ipar(18):ipar(19)-1) contains the rotation sines vector generated by the FGMRES method. There is no initial value for this part of *tmp* array.

tmp(ipar(19):ipar(20)-1) contains the rotated residual vector generated by the FGMRES method. There is no initial value for this part of *tmp* array.

tmp(ipar(20):ipar(21)-1) contains the solution vector to the least squares problem generated by the FGMRES method. There is no initial value for this part of *tmp* array.

`tmp(ipar(21) :)` contains the set of preconditioned vectors generated for the FGMRES method by the user. This part of `tmp` array is not used if non-preconditioned version of FGMRES method is called. There is no initial value for this part of `tmp` array.



NOTE. Advanced users can define this array in the code as `DOUBLE PRECISION tmp((2*ipar(15)+1)*n + ipar(15)*(ipar(15)+9)/2 + 1))` if they run only non-preconditioned FGMRES iterations.

dcg_init

Initializes the solver.

Syntax

```
dcg_init(n, x, b, RCI_request, ipar, dpar, tmp)
```

Description

The routine `dcg_init` initializes the solver. After initialization, all subsequent invocations of the Intel MKL RCI CG routines use the values of all parameters returned by the routine `dcg_init`. Advanced users can skip this step and set these parameters directly in the corresponding routines.



WARNING. Users can modify the contents of these arrays after they are passed to the solver routine only if they are sure that the values are correct and consistent. Basic check for correctness and consistency can be done by calling the `dcg_check` routine, but it does not guarantee that the method will work correctly.

Input Parameters

n	INTEGER. Contains the size of the problem, and the sizes of arrays x and b .
x	DOUBLE PRECISION array of size n . Contains the initial approximation to the solution vector. Normally it is equal to 0 or to b .
b	DOUBLE PRECISION array of size n . Contains the right-hand side vector.

Output Parameters

$RCI\_request$	INTEGER. Informs about result of work of the routine.
$ipar$	INTEGER array of size 128. Refer to the CG Common Parameters .
$dpar$	DOUBLE PRECISION array of size 128. Refer to the CG Common Parameters .
tmp	DOUBLE PRECISION array of size $(n, 4)$. Refer to the CG Common Parameters .

Return Values

$RCI\_request = 0$	The routine completes the task normally.
$RCI\_request = -10000$	The routine fails to complete the task.

dcg_check

Checks consistency and correctness of the user defined data.

Syntax

`dcg_check( $n$ ,  $x$ ,  $b$ ,  $RCI\_request$ ,  $ipar$ ,  $dpar$ ,  $tmp$ )`

Description

The routine `dcg_check` checks consistency and correctness of the parameters to be passed to the solver routine `dcg`. However this operation does not guarantee that the method gives the correct result. It only reduces the chance of making a mistake in the parameters of the method. Advanced users may skip this operation if they are sure that the correct data is specified in the solver parameters.



WARNING. Users can modify the contents of these arrays after they are passed to the solver routine only if they are sure that the values are correct and consistent. Basic check for correctness and consistency can be done by calling the `dcg_check` routine, but it does not guarantee that the method will work correctly.

Note that lengths of all vectors are assumed to have been defined in a previous call to the `dcg_init` routine.

Input Parameters

<i>n</i>	INTEGER. Contains the size of the problem, and size of arrays <i>x</i> and <i>b</i> .
<i>x</i>	DOUBLE PRECISION array of size <i>n</i> . Contains the initial approximation to the solution vector. Normally it is equal to 0 or to <i>b</i> .
<i>b</i>	DOUBLE PRECISION array of size <i>n</i> . Contains the right-hand side vector.

Output Parameters

<i>RCI_request</i>	INTEGER. Informs about result of work of the routine.
<i>ipar</i>	INTEGER array of size 128. Refer to the CG Common Parameters .
<i>dpar</i>	DOUBLE PRECISION array of size 128. Refer to the CG Common Parameters .
<i>tmp</i>	DOUBLE PRECISION array of size $(n, 4)$. Refer to the CG Common Parameters .

Return Values

<code>RCI_request= 0</code>	The routine completes task normally.
<code>RCI_request= -1100</code>	The routine is interrupted, errors occur.
<code>RCI_request= -1001</code>	The routine returns some warning messages.
<code>RCI_request= -1010</code>	The routine changes some parameters to make them consistent or correct.
<code>RCI_request= -1011</code>	The routine returns some warning messages and changes some parameters.

dcg

Computes the approximate solution vector.

Syntax

```
dcg(n, x, b, RCI_request, ipar, dpar, tmp)
```

Description

The routine `dcg` computes the approximate solution vector using the CG method [Young71]. , The routine `dcg` uses the value that was in the vector `x` before the first call as an initial approximation to the solution. The parameter `RCI_request` informs the user about the task completion and requests results of certain operations that are required by the solver.

Note that lengths of all vectors are assumed to have been defined in a previous call to the `dcg_init` routine.

Input Parameters

<code>n</code>	INTEGER. Contains the size of the problem, and size of arrays <code>x</code> and <code>b</code> .
<code>x</code>	DOUBLE PRECISION array of size <code>n</code> . Contains the initial approximation to the solution vector.
<code>b</code>	DOUBLE PRECISION array of size <code>n</code> . Contains the right-hand side vector.
<code>tmp</code>	DOUBLE PRECISION array of size $(n, 4)$. Refer to the CG Common Parameters .

Output Parameters

<i>RCI_request</i>	INTEGER. Informs about result of work of the routine.
<i>x</i>	DOUBLE PRECISION array of size <i>n</i> . Contains the updated approximation to the solution vector.
<i>ipar</i>	INTEGER array of size 128. Refer to the CG Common Parameters .
<i>dpar</i>	DOUBLE PRECISION array of size 128. Refer to the CG Common Parameters .
<i>tmp</i>	DOUBLE PRECISION array of size $(n, 4)$. Refer to the CG Common Parameters .

Return Values

<i>RCI_request</i> =0	The routine completes task normally, the solution is found and stored in the vector <i>x</i> . This occurs only if the stopping tests are fully automatic. For the user defined stopping tests, see the comments to the <i>RCI_request</i> = 2.
<i>RCI_request</i> =-1	The routine is interrupted because the maximal number of iterations is reached, but the relative stopping criterion is not met - this situation occurs only if both tests are requested by the user.
<i>RCI_request</i> =-2	The routine is interrupted because attempt to divide by zero occurs. This situation happens if the matrix is (almost) non-positive definite.
<i>RCI_request</i> =- 10	The routine is interrupted because the residual norm is invalid. Probably, the value <i>dpar</i> (6) was altered outside of the routine, or the <i>dcg_check</i> routine was not called.
<i>RCI_request</i> =-11	The routine is interrupted because it enters the infinite cycle. Probably, the values <i>ipar</i> (8), <i>ipar</i> (9), <i>ipar</i> (10) were altered outside of the routine, or the <i>dcg_check</i> routine was not called.
<i>RCI_request</i> = 1	Requests the user to multiply the matrix by <i>tmp</i> (1: <i>n</i> ,1), put the result in the <i>tmp</i> (1: <i>n</i> ,2), and return the control back to the routine <i>dcg</i> .

`RCI_request= 2`

Requests the user to perform the stopping tests. If they fail, the control must be returned back to the `dcg` routine. Otherwise, the solution is found and stored in the vector `x`.

`RCI_request= 3`

Requests the user to apply the preconditioner to `tmp(:, 3)`, put the result in the `tmp(:, 4)`, and return the control back to the routine `dcg`.

dcg_get

Retrieves the number of the current iteration.

Syntax

`dcg_get(n, x, b, RCI_request, ipar, dpar, tmp, itercount)`

Description

The routine `dcg_get` retrieves the current iteration number of the solutions process.

Input Parameters

<code>n</code>	INTEGER. Contains the size of the problem, and the sizes of arrays <code>x</code> and <code>b</code> .
<code>x</code>	DOUBLE PRECISION array of size <code>n</code> . Contains the initial approximation vector to the solution.
<code>b</code>	DOUBLE PRECISION array of size <code>n</code> . Contains the right-hand side vector.
<code>RCI_request</code>	INTEGER. This parameter is not used.
<code>ipar</code>	INTEGER array of size 128. Refer to the CG Common Parameters .
<code>dpar</code>	DOUBLE PRECISION array of size 128. Refer to the CG Common Parameters .
<code>tmp</code>	DOUBLE PRECISION array of size $(n, 4)$. Refer to the CG Common Parameters .

Output Parameters

itercount INTEGER argument. Contains the value of the current iteration number.

Return Values

The routine `dcg_get` does not return any value.

dcgmrhs_init

Initializes the RCI CG solver with MHRS.

Syntax

```
dcgmrhs_init(n, x, nrhs, b, method, RCI_request, ipar, dpar, tmp)
```

Description

The routine `dcgmrhs_init` initializes the solver. After initialization all subsequent invocations of the Intel MKL RCI CG with multiple right hand sides (MRHS) routines use the values of all parameters that are returned by `dcgmrhs_init`. Advanced users may skip this step and set the values to these parameters directly in the corresponding routines.



WARNING. Users can modify the contents of these arrays after they are passed to the solver routine only if they are sure that the values are correct and consistent. Basic check for correctness and consistency can be done by calling the `dcgmrhs_check` routine, but it does not guarantee that the method will work correctly.



NOTE. To use this routine with the name `dcg_init`, switch on the compiler preprocessor and include the files `mkl_solver.h` for C/C++, or `mkl_solver.f77` for FORTRAN.

Input Parameters

n INTEGER. Contains the size of the problem, and the sizes of arrays *x* and *b*.

x DOUBLE PRECISION matrix of size *n*-by-*nrhs*. Contains the initial approximation to the solution vectors. Normally it is equal to 0 or to *b*.

<i>nrhs</i>	INTEGER. This parameter sets the number of right-hand sides.
<i>b</i>	DOUBLE PRECISION matrix of size $(nrhs, n)$. Contains the right-hand side vectors.
<i>method</i>	INTEGER. Specifies the method of solution: 1 - CG with multiple right hand sides; default value 2 - Block-CG (not supported now).

Output Parameters

<i>RCI_request</i>	INTEGER. Informs about result of work of the routine.
<i>ipar</i>	INTEGER array of size $(128+2*nrhs)$. Refer to the CG Common Parameters .
<i>dpar</i>	DOUBLE PRECISION array of size $(128+2*nrhs)$. Refer to the CG Common Parameters .
<i>tmp</i>	DOUBLE PRECISION array of size $(n, 3+nrhs)$. Refer to the CG Common Parameters .

Return Values

<i>RCI_request</i> = 0	The routine completes task normally.
<i>RCI_request</i> = -10000	The routine fails to complete the task.

dcgmrhs_check

Checks consistency and correctness of the user defined data.

Syntax

```
dcgmrhs_check(n, x, nrhs, b, RCI_request, ipar, dpar, tmp)
```

Description

The routine `dcgmrhs_check` checks consistency and correctness of the parameters to be passed to the solver routine `dcgmrhs`. However this operation does not guarantee that the method gives the correct result. It only reduces the chance of making a mistake in the parameters of the method. Advanced users may skip this operation if they are sure that the correct data is specified in the solver parameters.



WARNING. Users can modify the contents of these arrays after they are passed to the solver routine only if they are sure that the values are correct and consistent. Basic check for correctness and consistency can be done by calling the `dcgmrhs_check` routine, but it does not guarantee that the method will work correctly.

Note that lengths of all vectors are assumed to have been defined in a previous call to the `dcgmrhs_init` routine.



NOTE. To use this routine with the name `dcg_check`, switch on the compiler preprocessor and include the files `mkl_solver.h` for C/C++, or `mkl_solver.f77` for FORTRAN.

Input Parameters

<i>n</i>	INTEGER. Contains the size of the problem, and the sizes of arrays <i>x</i> and <i>b</i> .
<i>x</i>	DOUBLE PRECISION matrix of size <i>n</i> by <i>nrhs</i> . Contains the initial approximation to the solution vectors. Normally it is equal to 0 or to <i>b</i> .
<i>nrhs</i>	INTEGER. This parameter sets the number of right-hand sides.
<i>b</i>	DOUBLE PRECISION matrix of size (<i>nrhs</i> , <i>n</i>). Contains the right-hand side vectors.

Output Parameters

<i>RCI_request</i>	INTEGER. Informs about result of work of the routine.
<i>ipar</i>	INTEGER array of size (<i>128+2*nrhs</i>). Refer to the CG Common Parameters .
<i>dpar</i>	DOUBLE PRECISION array of size (<i>128+2*nrhs</i>). Refer to the CG Common Parameters .
<i>tmp</i>	DOUBLE PRECISION array of size (<i>n</i> , <i>3+nrhs</i>). Refer to the CG Common Parameters .

Return Values

RCI_request= 0 The routine completes task normally.

<code>RCI_request= -1100</code>	The routine is interrupted, errors occur.
<code>RCI_request= -1001</code>	The routine returns some warning messages.
<code>RCI_request= -1010</code>	The routine changes some parameters to make them consistent or correct.
<code>RCI_request= -1011</code>	The routine returns some warning messages and changes some parameters.

dcgmrhs

Computes the approximate solution vectors.

Syntax

`dcgmrhs(n, x, nrhs, b, RCI_request, ipar, dpar, tmp)`

Description

The routine `dcgmrhs` computes the approximate solution vectors using the CG with multiple right hand sides (MRHS) method [Young71]. The routine `dcgmrhs` uses the value that was in the `x` before the first call as an initial approximation to the solution. The parameter `RCI_request` informs the user about task completion status and requests results of certain operations that are required by the solver.

Note that lengths of all vectors are assumed to have been defined in a previous call to the `dcgmrhs_init` routine.



NOTE. To use this routine with the name `dcg`, the user must switch on the compiler's preprocessor and include the files `mkl_solver.h` for C/C++, or `mkl_solver.f77` for FORTRAN.

Input Parameters

<code>n</code>	INTEGER. Contains the size of the problem, and the sizes of arrays <code>x</code> and <code>b</code> .
<code>x</code>	DOUBLE PRECISION matrix of size <code>n-by-nrhs</code> . Contains the initial approximation to the solution vectors.
<code>nrhs</code>	INTEGER. This parameter sets the number of right-hand sides.

<i>b</i>	DOUBLE PRECISION matrix of size $(nrhs, n)$. Contains the right-hand side vectors.
<i>tmp</i>	DOUBLE PRECISION array of size $(n, 3+nrhs)$. Refer to the CG Common Parameters .

Output Parameters

<i>RCI_request</i>	INTEGER. Informs about result of work of the routine.
<i>x</i>	DOUBLE PRECISION matrix of size n -by- $nrhs$. Contains the updated approximation to the solution vectors.
<i>ipar</i>	INTEGER array of size $(128+2*nrhs)$. Refer to the CG Common Parameters .
<i>dpar</i>	DOUBLE PRECISION array of size $(128+2*nrhs)$. Refer to the CG Common Parameters .
<i>tmp</i>	DOUBLE PRECISION array of size $(n, 3+nrhs)$. Refer to the CG Common Parameters .

Return Values

<i>RCI_request</i> =0	The routine completes task normally, the solution is found and stored in the matrix <i>x</i> . This occurs only if the stopping tests are fully automatic. For the user defined stopping tests, see the comments to the <i>RCI_request</i> = 2.
<i>RCI_request</i> =-1	The routine is interrupted because the maximal number of iterations is reached, but the relative stopping criterion is not satisfied. This situation occurs only if both tests are requested by the user.
<i>RCI_request</i> =-2	The routine is interrupted because attempt to divide by zero occurs. This happens if the matrix is (almost) non-positive definite.
<i>RCI_request</i> =- 10	The routine is interrupted because the residual norm is invalid. Probably, the value <i>dpar</i> (6) was altered outside of the routine, or the routine <i>dpgmrhs_check</i> was not called.

`RCI_request=-11`

The routine is interrupted because it enters the infinite cycle. Probably, the values `ipar(8)`, `ipar(9)`, `ipar(10)` were altered outside of the routine, or the routine `dcgmrhs_check` was not called.

`RCI_request= 1`

Requests the user to multiply the matrix by `tmp(1:n,1)`, put the result in the `tmp(1:n,2)`, and return the control back to the routine `dcgmrhs`.

`RCI_request= 2`

Requests the user to perform the stopping tests. If they fail, the control must be returned back to the routine `dcgmrhs`. Otherwise, the solution is found and stored in the matrix `x`.

`RCI_request= 3`

Requests the user to apply the preconditioner to `tmp(:,3+ipar(3))`, put the result in `tmp(:,3)`, and return the control back to the routine `dcgmrhs`.

dcgmrhs_get

Retrieves the number of the current iteration.

Syntax

`dcgmrhs_get(n, x, b, RCI_request, ipar, dpar, tmp, itercount)`

Description

The routine `dcgmrhs_get` retrieves the current iteration number of the solving process.



NOTE. To use this routine with the name `dcg_get`, switch on the compiler preprocessor and include the files `mk1_solver.h` for C/C++, or `mk1_solver.f77` for FORTRAN.

Input Parameters

<code>n</code>	INTEGER. Contains the size of the problem, and the sizes of arrays <code>x</code> and <code>b</code> .
<code>x</code>	DOUBLE PRECISION matrix of size <code>n</code> -by- <code>nrhs</code> . Contains the initial approximation to the solution vectors.

<i>nrhs</i>	INTEGER. This parameter sets the number of right-hand sides.
<i>b</i>	DOUBLE PRECISION matrix of size $(nrhs, n)$. Contains the right-hand side .
<i>RCI_request</i>	INTEGER. This parameter is not used.
<i>ipar</i>	INTEGER array of size $(128+2*nrhs)$. Refer to the CG Common Parameters .
<i>dpar</i>	DOUBLE PRECISION array of size $(128+2*nrhs)$. Refer to the CG Common Parameters .
<i>tmp</i>	DOUBLE PRECISION array of size $(n, 3+nrhs)$. Refer to the CG Common Parameters .

Output Parameters

<i>itercount</i>	INTEGER argument. Contains the value of the current iteration number.

Return Values

The routine `dcgmrhs_get` does not return any value.

dfgmres_init

Initializes the solver.

Syntax

```
dfgmres_init(n, x, b, RCI_request, ipar, dpar, tmp)
```

Description

The routine `dfgmres_init` initializes the solver. After initialization all subsequent invocations of Intel MKL RCI FGMRES routines use the values of all parameters that are returned by `dfgmres_init`. Advanced users may skip this step and set the values to these parameters directly in the corresponding routines.



WARNING. Users can modify the contents of these arrays after they are passed to the solver routine only if they are sure that the values are correct and consistent. Basic check for correctness and consistency can be done by calling the `dfgmres_check` routine, but it does not guarantee that the method will work correctly.

Input Parameters

n	INTEGER. Contains the size of the problem, and the sizes of arrays x and b .
x	DOUBLE PRECISION array of size n . Contains the initial approximation to the solution vector. Normally it is equal to 0 or to b .
b	DOUBLE PRECISION array of size n . Contains the right-hand side vector.

Output Parameters

$RCI\_request$	INTEGER. Informs about result of work of the routine.
$ipar$	INTEGER array of size 128. Refer to the FGMRES Common Parameters .
$dpar$	DOUBLE PRECISION array of size 128. Refer to the FGMRES Common Parameters .
tmp	DOUBLE PRECISION array of size $((2*ipar(15)+1)*n+ipar(15)*ipar(15)+9)/2 + 1$. Refer to the FGMRES Common Parameters .

Return Values

$RCI\_request = 0$	The routine completes task normally.
$RCI\_request = -10000$	The routine fails to complete the task.

dfgmres_check

Checks consistency and correctness of the user defined data.

Syntax

`dfgmres_check( $n$ ,  $x$ ,  $b$ ,  $RCI\_request$ ,  $ipar$ ,  $dpar$ ,  $tmp$ )`

Description

The routine `dfgmres_check` checks consistency and correctness of the parameters to be passed to the solver routine `dfgmres`. However this operation does not guarantee that the method gives the correct result. It only reduces the chance of making a mistake in the parameters of the method. Advanced users may skip it if they are sure that the correct data is specified in the solver parameters.



WARNING. Users can modify the contents of these arrays after they are passed to the solver routine only if they are sure that the values are correct and consistent. Basic check for correctness and consistency can be done by calling the `dfgmres_check` routine, but it does not guarantee that the method will work correctly.

Note that lengths of all vectors are assumed to have been defined in a previous call to the `dfgmres_init` routine.

Input Parameters

<i>n</i>	INTEGER. Contains the size of the problem, and the sizes of arrays <i>x</i> and <i>b</i> .
<i>x</i>	DOUBLE PRECISION array of size <i>n</i> . Contains the initial approximation to the solution vector. Normally it is equal to 0 or to <i>b</i> .
<i>b</i>	DOUBLE PRECISION array of size <i>n</i> . Contains the right-hand side vector.

Output Parameters

<i>RCI_request</i>	INTEGER. Informs about result of work of the routine.
<i>ipar</i>	INTEGER array of size 128. Refer to the FGMRES Common Parameters .
<i>dpar</i>	DOUBLE PRECISION array of size 128. Refer to the FGMRES Common Parameters .
<i>tmp</i>	DOUBLE PRECISION array of size $((2 * ipar(15) + 1) * n + ipar(15) * ipar(15) + 9) / 2 + 1$. Refer to the FGMRES Common Parameters .

Return Values

<code>RCI_request= 0</code>	The routine completes task normally.
<code>RCI_request= -1100</code>	The routine is interrupted, errors occur.
<code>RCI_request= -1001</code>	The routine returns some warning messages.
<code>RCI_request= -1010</code>	The routine changes some parameters to make them consistent or correct.
<code>RCI_request= -1011</code>	The routine returns some warning messages and changes some parameters.

dfgmres

Makes the FGMRES iterations.

Syntax

```
dfgmres(n, x, b, RCI_request, ipar, dpar, tmp)
```

Description

The routine `dfgmres` performs the FGMRES iterations [Saad03]. The routine `dfgmres` uses the value that was in the vector x before the first call as an initial approximation to the solution. To update the current approximation to the solution, the `dfgmres_get` routine must be called. The RCI FGMRES iterations can be continued after the call to the `dfgmres_get` routine only if the value of the parameter `ipar(13)` is not equal to 0 (default value). Note that the updated solution overwrites the right hand side in the vector b if the parameter `ipar(13)` is positive, and the restarted version of the FGMRES method can not be run. If the right hand side should be kept, it must be saved in a different memory location before the first call to the `dfgmres_get` routine with a positive `ipar(13)`.

The parameter `RCI_request` informs the user about the task completion and requests results of certain operations that are needed to the solver.

Note that lengths of all the vectors are assumed to have been defined in a previous call to the `dfgmres_init` routine.

Input Parameters

`n` INTEGER. Contains the size of the problem, and the sizes of arrays x and b .

<i>x</i>	DOUBLE PRECISION array of size <i>n</i> . Contains the initial approximation to the solution vector.
<i>b</i>	DOUBLE PRECISION array of size <i>n</i> . Contains the right-hand side vector.
<i>tmp</i>	DOUBLE PRECISION array of size $((2*ipar(15)+1)*n+ipar(15)*ipar(15)+9)/2 + 1$. Refer to the FGMRES Common Parameters .

Output Parameters

<i>RCI_request</i>	INTEGER. Informs about result of work of the routine.
<i>ipar</i>	INTEGER array of size 128. Refer to the FGMRES Common Parameters .
<i>dpar</i>	DOUBLE PRECISION array of size 128. Refer to the FGMRES Common Parameters .
<i>tmp</i>	DOUBLE PRECISION array of size $((2*ipar(15)+1)*n+ipar(15)*ipar(15)+9)/2 + 1$. Refer to the FGMRES Common Parameters .

Return Values

<i>RCI_request</i> = 0	The routine completes task normally, the solution is found. This occurs only if the stopping tests are fully automatic. For the user defined stopping tests, see the comments to the <i>RCI_request</i> = 2 or 4.
<i>RCI_request</i> = -1	The routine is interrupted because the maximal number of iterations is reached, but the relative stopping criterion is not met. This situation occurs only if both tests are requested by the user.
<i>RCI_request</i> = -10	The routine is interrupted because attempt to divide by zero occurs. Normally it happens if the matrix is (almost) degenerate. However it may happen if the parameter <i>dpar</i> is altered by mistake, or if the method is not stopped when the solution is found.

`RCI_request= -11`

The routine is interrupted because it enters the infinite cycle. Probably, the values `ipar(8)`, `ipar(9)`, `ipar(10)` were altered outside of the routine, or the `dfgmres_check` routine was not called.

`RCI_request= -12`

The routine is interrupted because errors are found in the method parameters. Normally this happens if the parameters `ipar` and `dpar` are altered by mistake outside the routine.

`RCI_request= 1`

Requests the user to multiply the matrix by `tmp(ipar(22))`, put the result in the `tmp(ipar(23))`, and return the control back to the routine `dfgmres`.

`RCI_request= 2`

Requests the user to perform the stopping tests. If they fail, the control back must be returned to the `dfgmres` routine. Otherwise, the FGMRES solution is found, and the `fgmres_get` routine can be run to update the computed solution in the vector `x`.

`RCI_request= 3`

Requests the user to apply the inverse preconditioner to `ipar(22)`, put the result in the `ipar(23)`, and return the control back to the routine `dfgmres`.

`RCI_request= 4`

Requests the user to check the norm of the currently generated vector. If it is not zero within the computational/rounding errors, the control must be returned back to the `dfgmres` routine. Otherwise, the FGMRES solution is found, and the `dfgmres_get` routine can be run to update the computed solution in the vector `x`.

dfgmres_get

Retrieves the number of the current iteration and updates the solution.

Syntax

`dfgmres_get(n, x, b, RCI_request, ipar, dpar, tmp, itercount)`

Description

The routine `dfgmres_get` retrieves the current iteration number of the solution process and updates the solution according to the computations performed by the `dfgmres` routine. To retrieve the current iteration number only, set the parameter `ipar(13) = -1` beforehand. This is normally recommended to do to proceed further with the computations. If the intermediate solution is needed, the method parameters must be set properly, see for details [FGMRES Common Parameters](#) and [Iterative Sparse Solver Code Examples](#) section in the Appendix C.

Input Parameters

<i>n</i>	INTEGER. Contains the size of the problem, and the sizes of the arrays <i>x</i> and <i>b</i> .
<i>ipar</i>	INTEGER array of size 128. Refer to the FGMRES Common Parameters .
<i>dpar</i>	DOUBLE PRECISION array of size 128. Refer to the FGMRES Common Parameters .
<i>tmp</i>	DOUBLE PRECISION array of size $((2*ipar(15)+1)*n+ipar(15)*ipar(15)+9)/2 + 1$. Refer to the FGMRES Common Parameters .

Output Parameters

<i>x</i>	DOUBLE PRECISION array of size <i>n</i> . If <code>ipar(13) = 0</code> , it contains the updated approximation to the solution according to the computations done in <code>dfgmres</code> routine. Otherwise, it is not changed.
<i>b</i>	DOUBLE PRECISION array of size <i>n</i> . If <code>ipar(13) > 0</code> , it contains the updated approximation to the solution according to the computations done in <code>dfgmres</code> routine. Otherwise, it is not changed.
<i>RCI_request</i>	INTEGER. Informs about result of work of the routine.
<i>itercount</i>	INTEGER argument. Contains the value of the current iteration number.

Return Values

<i>RCI_request</i> = 0	The routine completes task normally.

`RCI_request= -12`

The routine is interrupted because errors are found in the method parameters. Normally this happens if the parameters *ipar* and *dpar* are altered by mistake outside the routine.

`RCI_request= -10000`

The routine fails to complete the task.

Implementation Details

Several aspects of the Intel MKL RCI ISS interface are platform-specific and language-specific. To promote portability across platforms and ease of use across different languages, include one of the Intel MKL RCI ISS language-specific header files. Currently, there is one language specific header file for C programs.

This language-specific header file defines function prototypes and they are the following:

```
void dcg_init(int *n, double *x, double *b, int *rci_request, int *ipar,
double dpar, double *tmp);
```

```
void dcg_check(int *n, double *x, double *b, int *rci_request, int *ipar,
double dpar, double *tmp);
```

```
void dcg(int *n, double *x, double *b, int *rci_request, int *ipar, double
dpar, double *tmp);
```

```
void dcg_get(int *n, double *x, double *b, int *rci_request, int *ipar,
double dpar, double *tmp, int *itercount);
```

```
void dcgmrhs_init(int *n, double *x, int *nRhs, double *b, int *method, int
*rci_request, int *ipar, double dpar, double *tmp);
```

```
void dcgmrhs_check(int *n, double *x, int *nRhs, double *b, int *rci_request,
int *ipar, double dpar, double *tmp);
```

```
void dcgmrhs(int *n, double *x, int *nRhs, double *b, int *rci_request, int
*ipar, double dpar, double *tmp);
```

```
void dcgmrhs_get(int *n, double *x, int *nRhs, double *b, int *rci_request,
int *ipar, double dpar, double *tmp, int *itercount);
```

```
void dfgmres_init(int *n, double *x, double *b, int *rci_request, int *ipar,
double dpar, double *tmp);
```

```
void dfgmres_check(int *n, double *x, double *b, int *rci_request, int *ipar,
double dpar, double *tmp);
```

```
void dfgmres(int *n, double *x, double *b, int *rci_request, int *ipar,
double dpar, double *tmp);
```

```
void dfgmres_get(int *n, double *x, double *b, int *rci_request, int *ipar,  
double dpar, double *tmp, int *itercount);
```



NOTE. Intel MKL does not support the RCI ISS interface without the language specific header file included.

Preconditioners based on Incomplete LU Factorization Technique

Usually, preconditioners, or accelerators are used to accelerate an iterative solution process. In some cases, their use can reduce the number of iterations dramatically and thus lead to better solver performance. Although the terms 'preconditioner' and 'accelerator' are synonyms, hereafter only 'preconditioner' is used.

Currently, Intel MKL provides two preconditioners, ILU0 and ILUT, for the sparse matrix presented only in the format accepted in the Intel MKL direct sparse solvers (three-array variation of the CSR storage format, see [Sparse Matrix Storage Format](#) section). The used algorithms are described in [Saad03].

The ILU0 preconditioner is based on a well-known factorization of the original matrix into a product of two triangular matrices: low triangular and upper triangular matrices. Usually, such decomposition leads to some fill-in in the resulting matrix structure in comparison with the original matrix. The distinctive feature of the ILU0 preconditioner is that it preserves the structure of the original matrix in the result.

Unlike the ILU0 preconditioner, the ILUT preconditioner preserves some resulting fill-in in the preconditioner matrix structure. The distinctive feature of the ILUT preconditioner is that it saves the resulting entry of the preconditioner if the entry satisfies two conditions simultaneously: the value of the entry is greater than the product of the given tolerance and matrix row norm, and the entry is in the given bandwidth of the resulting preconditioner matrix.

Both ILU0 and ILUT preconditioners can apply to any non-degenerate matrix. They can be used alone or together with the Intel MKL RCI FGMRES solver (see [Sparse Solver Routines](#)). Avoid using this preconditioners with MKL RCI CG solver because in general, they produce non-symmetric resulting matrix even if the original matrix is symmetric. Usually, an inverse of the preconditioner is required in this case. To do this the Intel MKL triangular solver routine [mkl_dcsrtrsv](#) must be applied twice: for the low triangular part of the preconditioner, and then for its upper triangular part.



NOTE. Although ILU0 and ILUT preconditioners apply to any non-degenerate matrix, in some cases the algorithm may fail to ensure successful termination and the required result. The result of the preconditioner routine can be checked only in practice.

A preconditioner may increase the number of iterations for an arbitrary case of the system and the initial guess, and even ruin the convergence. It is user's responsibility to carefully use a suitable preconditioner.

General Scheme of Using ILUT and RCI FGMRES Routines

The scheme is the same for both preconditioners. Some differences exist in the calling parameters of the preconditioners and in the subsequent call of two triangular solvers. All these differences can be seen in the example codes for both preconditioners in the [Preconditioner Code Examples](#) in the Appendix C.

The following pseudocode shows the general scheme of using the ILUT preconditioner in the RCI FGMRES context.

```
...
generate matrix A
generate preconditioner C (optional)
    call dfgmres_init(n, x, b, RCI_request, ipar, dpar, tmp)
    change parameters in ipar, dpar if necessary
    call dcsrilut(n, a, ia, ja, bilut, ibilut, jbilut, tol, maxfil, ipar, dpar,
ierr)
    call dfgmres_check(n, x, b, RCI_request, ipar, dpar, tmp)
1  call dfgmres(n, x, b, RCI_request, ipar, dpar, tmp)
    if (RCI_request.eq.1) then
        multiply the matrix A by tmp(ipar(22)) and put the result in tmp(ipar(23))
c  proceed with FGMRES iterations
    goto 1
endif
if (RCI_request.eq.2) then
    do the stopping test
```

```

        if (test not passed) then
c   proceed with FGMRES iterations
            go to 1
        else
c   stop FGMRES iterations.
            goto 2
        endif
    endif

    if (RCI_request.eq.3) then
c   Below, trvec is an intermediate vector of length at least n
c   Here is the recommended use of the result produced by the ILUT routine.
c   via standard Intel MKL Sparse Blas solver routine mkl_dcsrtrsv.
        call mkl_dcsrtrsv('L','N','U', n, bilut, ibilut, jbilut,
            tmp(ipar(22)), trvec)

        call mkl_dcsrtrsv('U','N','N', n, bilut, ibilut, jbilut, trvec,
            tmp(ipar(23)))
c   proceed with FGMRES iterations
            goto 1
        endif

    if (RCI_request.eq.4) then
        check the norm of the next orthogonal vector, it is contained in dpar(7)
        if (the norm is not zero up to rounding/computational errors) then
c   proceed with FGMRES iterations
            goto 1
        else
c   stop FGMRES iterations
            goto 2
        endif
    endif

```

```
endif
2 call dfgmres_get(n, x, b, RCI_request, ipar, dpar, tmp, itercount)
current iteration number is in itercount
the computed approximation is in the array x
```

ILU0 and ILUT Preconditioners Interface Description

The concepts required to understand the use of the Intel MKL preconditioner routines are discussed in the [Linear Solvers Basics](#).

In this section the FORTRAN style notations are used.

All types in this documentation refer to the standard Fortran types, `INTEGER`, and `DOUBLE PRECISION`.

C and C++ programmers should refer to the section [Calling Sparse Solver Routines From C/C++](#) for information on mapping Fortran types to C/C++ types.

User Data Arrays

The preconditioner routines take arrays of user data as input, for example, user arrays representing the original matrix. To minimize storage requirements and improve overall run-time efficiency, the Intel MKL preconditioner routines do not make copies of the user input arrays.

Common Parameters

The preconditioners have parameters for passing various options to the routine. The values for these parameters should be specified very carefully. These values can be changed during computations according to the user's needs.

Some parameters are common with the [FGMRES Common Parameters](#). Only the routine `dfgmres_init` specifies their default and initial values. However, some parameters can be redefined with other values. These parameters are listed below.

For the ILU0 preconditioner:

`ipar(2)` - specifies the destination of error messages generated by the ILU0 routine. The default value 6 means that all error messages are displayed on the screen. Otherwise the error messages are written to the newly created file `MKL_PREC_log.txt`. Note if the parameter `ipar(6)` is set to 0, then error messages are not generated at all.

ipar(6) - if its value is not equal to 0, the ILU0 routine returns error messages in accordance with the parameter *ipar(2)*. Otherwise, the routine does not generate error messages at all, but returns a negative value of the parameter *ierr*. The default value is 1.

For the ILUT preconditioner:

ipar(2) - specifies the destination of error messages generated by the ILUT routine. The default value 6 means that all messages are displayed on the screen. Otherwise the error messages are written to the newly created file `MKL_PREC_log.txt`. Note if the parameter *ipar(6)* is set to 0, error messages are not generated at all.

ipar(6) - if its value is not equal to 0, the ILUT routine returns error messages in accordance with the parameter *ipar(2)*. Otherwise, the routine does not generate error messages at all, but returns a negative value of the parameter *ierr*. The default value is 1.

ipar(7) - if its value is greater than 0, the ILUT routine generates warning messages in accordance with the parameter *ipar(2)* and continues calculations. Otherwise, if its value is equal to 0, the routine returns a positive value of the parameter *ierr*, if its value is less than 0, the routine generates a warning message in accordance with the parameter *ipar(2)* and returns a positive value of the parameter *ierr*. The default value is 1.



NOTE. Users must provide correct and consistent parameters to the routine to avoid fails or wrong results.

dcsrilu0

ILU0 preconditioner based on incomplete LU factorization of a sparse matrix in the CSR format (PARDISO variation).

Syntax

Fortran:

```
call dcsrilu0(n, a, ia, ja, bilu0, ipar, dpar, ierr)
```

C:

```
dcsrilu0(&n, a, ia, ja, bilu0, ipar, dpar, &ierr);
```

Input Parameters

n	INTEGER. Size (number of rows or columns) of the original square n -by- n -matrix A .
a	DOUBLE PRECISION. Array containing the set of elements of the matrix A . Its length is equal to the number of non-zero elements in the matrix A . Refer to values array description in the Sparse Matrix Storage Format section for more details.
ia	INTEGER. Array of size $(n+1)$ containing begin indices of rows of matrix A such that $ia(I)$ is the index in the array A of the first non-zero element from the row I . The value of the last element $ia(n+1)$ is equal to the number of non-zeros in the matrix A plus one. Refer to the <i>rowIndex</i> array description in the Sparse Matrix Storage Format section for more details.
ja	INTEGER. Array containing the column indices for each non-zero element of the matrix A . Its size is equal to the size of the array a . Refer to the <i>columns</i> array description in the Sparse Matrix Storage Format section for more details.



NOTE. Column indices should be put in increasing order for each row of matrix.

$ipar$ INTEGER array of size 128. This parameter is used to specify the integer set of data for both the ILU0 and RCI FGMRES computations. Refer to the $ipar$ array description in the [FGMRES Common Parameters](#) for more details on FGMRES parameter entries. The entries that are specific to ILU0 are listed below.

$ipar(31)$ - specifies how the routine operates if zero diagonal element occurs during calculation. If this parameter is set to 0 (default value set by the routine `dfgmres_init`) then that the calculations are stopped and the routine returns non-zero error value. Otherwise the value

of the diagonal element is set to the specified value and the calculations are continued.



NOTE. Advanced users can define this array in the code as follows:
`INTEGER ipar(31)`. However, to guarantee the compatibility with the future releases of the Intel MKL it is highly recommended to declare the array *ipar* of length 128.

dpar

DOUBLE PRECISION array of size 128. This parameter is used to specify the double precision set of data for both the ILU0 and RCI FGMRES computations. Refer to the *dpar* array description in the [FGMRES Common Parameters](#) for more details on FGMRES parameter entries. The entries that are specific to ILU0 are listed below.

dpar(31)

- specifies the small value that is compared with the diagonal elements during calculations; if the value of the diagonal element is smaller, then it is set to *dpar*(32), or the calculations are stopped, in accordance with *ipar*(31); the default value is 1.0D-16



NOTE. This parameter can be set to the negative value, because its absolute value is actually used in calculations.

If this parameter is set to 0, the comparison with the diagonal element is not performed.

`dpar(32)`

- specifies the value that is assigned to the diagonal element if its value is less than `dpar(31)` (see above); the default value is 1.0D-10



NOTE. Advanced users can define this array in the code as follows:

```
DOUBLE PRECISION dpar(32).
```

However, to guarantee the compatibility with the future releases of the Intel MKL it is highly recommended to declare the array `dpar` of length 128.

Output Parameters

`bilu0`

DOUBLE PRECISION. Array *B* stored in the PARDISO CSR format containing non-zero elements of the resulting preconditioning matrix. Its size is equal to the number of non-zero elements in the matrix *A*. Refer to the *values* array description in the [Sparse Matrix Storage Format](#) section for more details.

`ierr`

INTEGER. Error flag, informs about the routine completion status.



NOTE. To present the resulting preconditioning matrix in the CSR format the arrays *ia* (row indices) and *ja* (column indices) of the input matrix must be used.

Description

The routine `dcsrilu0` computes a preconditioner *B* [Saad03] of a given sparse matrix *A* stored in the CSR format accepted in PARDISO:

$A \sim B = L * U$, where *L* is a low triangular matrix with unit diagonal, *U* is an upper triangular matrix with non-unit diagonal, and the portrait of the original matrix *A* is used to store the incomplete factors *L* and *U*.

Return Values

`ierr=0`

The routine completed task normally.

`ierr=-101`

The routine is interrupted, the error occurs: at least one diagonal element is omitted from the matrix in CSR format (see [Sparse Matrix Storage Format](#)).

`ierr=-102`

The routine is interrupted because the matrix contains zero diagonal element, routine can not perform operations.

`ierr=-103`

The routine is interrupted as the matrix contains too small diagonal element, and an overflow may occur because of the division by its value required to complete the task, or a bad approximation to ILU0 with use of this element will be computed.

`ierr=-104`

The routine is interrupted because the memory is insufficient for the internal work array.

`ierr=-105`

The routine is interrupted because the input matrix size n is less than or equal to 0.

`ierr=-106`

The routine is interrupted because the column indices ja are placed in not increasing order.

Interfaces

FORTRAN 77 and Fortran 95:

```
SUBROUTINE dcsrilu0 (n, a, ia, ja, bilu0, ipar, dpar, ierr)
```

```
INTEGER n, ierr, ipar(128)
```

```
INTEGER ia(*), ja(*)
```

```
DOUBLE PRECISION a(*), bilu0(*), dpar(128)
```

C:

```
void dcsrilu0 (int *n, double *a, int *ia, int *ja, double *bilu0, int *ipar, double *dpar,
int *ierr);
```

dcsrilut

ILUT preconditioner based on the incomplete LU factorization with a threshold of a sparse matrix.

Syntax

Fortran:

```
call dcsrilut(n, a, ia, ja, bilut, bilut, ibilut, jbilut, tol, maxfil, ipar,
dpar, ierr)
```

C:

```
dcsrilut(&n, a, ia, ja, bilut, bilut, ibilut, jbilut, &tol, &maxfil, ipar,
dpar, &ierr);
```

Description

The routine `dcsrilut` computes a preconditioner B [Saad03] of a given sparse matrix A stored in the format accepted in the direct sparse solvers:

$A \sim B = L * U$, where L is a low triangular matrix with unit diagonal, U is an upper triangular matrix with non-unit diagonal.

The following threshold criteria are used to generate the incomplete factors L and U :

- 1) the resulting entry must be greater than the matrix current row norm multiplied by the parameter `tol`, and
- 2) the number of the non-zero elements in each row of the resulting L and U factors must not be greater than the value of the parameter `maxfil`.

Input Parameters

<i>n</i>	INTEGER. Size (number of rows or columns) of the original square n -by- n matrix A .
<i>a</i>	DOUBLE PRECISION. Array containing all non-zero elements of the matrix A . The length of the array is equal to their number. Refer to <i>values</i> array description in the Sparse Matrix Storage Format section for more details.
<i>ia</i>	INTEGER. Array of size $(n+1)$ containing indices of non-zero elements in the array A . $ia(I)$ is the index of the first non-zero element from the row I . The value of the last

element $ia(n+1)$ is equal to the number of non-zeros in the matrix A plus one. Refer to the *rowIndex* array description in the [Sparse Matrix Storage Format](#) section for more details.

ja

INTEGER. Array of size equal to the size of the array *a*. This array contains the column numbers for each non-zero element of the matrix A . Refer to the *columns* array description in the [Sparse Matrix Storage Format](#) section for more details.



NOTE. Column numbers must be in increasing order for each row of matrix.

tol

DOUBLE PRECISION. Tolerance for threshold criterion for the resulting entries of the preconditioner.

maxfil

INTEGER. Maximum fill-in, half of the preconditioner bandwidth. The number of non-zero elements in the rows of the preconditioner can not exceed $(2*maxfil+1)$.

ipar

INTEGER array of size 128. This parameter is used to specify the integer set of data for both the ILUT and RCI FGMRES computations. Refer to the *ipar* array description in the [FGMRES Common Parameters](#) for more details on FGMRES parameter entries. The entries specific to ILUT are listed below.

ipar(31)

- specifies how the routine operates if the value of the computed diagonal element is less than the current matrix row norm multiplied by the value of the parameter *tol*. If *ipar*(31) = 0, then the calculation is stopped and the routine returns non-zero error value. Otherwise, the value of the diagonal element is set to the certain value (see description of the parameter *dpar*(31) below), and the calculation continues.



NOTE. There is no default value for *ipar*(31) entry even if the preconditioner is used within the RCI ISS context. Always set the value of this entry.



NOTE. Advanced users can define this array in the code as `INTEGER ipar(31)`. However, to guarantee the compatibility with the future releases of the Intel MKL, declare the array *ipar* of length 128.

dpar

DOUBLE PRECISION array of size 128. This parameter specifies the double precision set of data for both ILUT and RCI FGMRES computations. Refer to the *dpar* array description in the [FGMRES Common Parameters](#) for more details on FGMRES parameter entries. The entries that are specific to ILUT are listed below.

dpar(31)

- specifies the value that being multiplied by the matrix row norm is assigned to those diagonal elements whose value is less than the matrix row norm multiplied by the value of the parameter *tol*, if the calculation is not stopped in accordance with *ipar*(31).



NOTE. There is no default value for *dpar*(31) entry even if the preconditioner is used within RCI ISS context. Always set the value of this entry.



NOTE. Advanced users can define this array in the code as `DOUBLE PRECISION dpar(31)`. However, to guarantee the compatibility with the future releases of the Intel MKL, declare the array `dpar` of length 128.

Output Parameters

bilut

DOUBLE PRECISION. Array *B* containing non-zero elements of the resulting preconditioning matrix, stored format accepted in the direct sparse solvers. Refer to the *values* array description in the [Sparse Matrix Storage Format](#) section for more details. The size of the array is equal to $(2*maxfil+1)*n-maxfil*(maxfil+1)+1$.



NOTE. Provide enough memory for this array before calling the routine. Otherwise, the routine may fail to complete successfully with a correct result.

ibilut

INTEGER. Array of size $(n+1)$ containing indices of non-zero elements in the array *B*. *ibilut(I)* is the index of the first non-zero element from the row *I*. The value of the last element *ibilut(n+1)* is equal to the number of non-zeros in the matrix *B* plus one. Refer to the *rowIndex* array description in the [Sparse Matrix Storage Format](#) section for more details..

jbilut

INTEGER. Array, its size is equal to the size of the array *bilut*. This array contains the column numbers for each non-zero element of the matrix *B*. Refer to the *columns* array description in the [Sparse Matrix Storage Format](#) section for more details.

ierr

INTEGER. Error flag, informs about the routine completion.

Return Values

ierr=0

The routine completes the task normally.

<code>ierr=-101</code>	The routine is interrupted, the error occurs: the number of elements in some matrix row specified in the sparse format is equal to or less than 0.
<code>ierr=-102</code>	The routine is interrupted because the value of the computed diagonal element is less than the product of the given tolerance and the current matrix row norm, and it cannot be replaced as <code>ipar(31)=0</code> .
<code>ierr=-103</code>	The routine is interrupted, the error occurs: the element <code>ia(I+1)</code> is less than or equal to the element <code>ia(I)</code> (see Sparse Matrix Storage Format section).
<code>ierr=-104</code>	The routine is interrupted because the memory is insufficient for the internal work arrays.
<code>ierr=-105</code>	The routine is interrupted because the input value of <code>maxfil</code> is less than 0.
<code>ierr=-106</code>	The routine is interrupted because the size <code>n</code> of the input matrix is less than 0.
<code>ierr=-107</code>	The routine is interrupted, the error occurs: an element of the array <code>ja</code> is less than 0, or greater than <code>n</code> (see Sparse Matrix Storage Format section).
<code>ierr=101</code>	The value of <code>maxfil</code> is greater than or equal to <code>n</code> . The calculation is performed with the value of <code>maxfil</code> set to <code>(n-1)</code> .
<code>ierr=102</code>	The value of <code>tol</code> is less than 0. The calculation is performed with the value of the parameter set to <code>(-tol)</code>
<code>ierr=103</code>	The absolute value of <code>tol</code> is greater than value of <code>dpar(31)</code> ; it can result in instability of the calculation.
<code>ierr=104</code>	The value of <code>dpar(31)</code> is equal to 0. It can cause the fail of the calculations.

Interfaces

FORTRAN 77 and Fortran 95:

```
SUBROUTINE dcsrilut (n, a, ia, ja, bilut, ibilut, jbilut, tol, maxfil, ipar, dpar, ierr)
  INTEGER n, ierr, ipar(*), maxfil
  INTEGER ia(*), ja(*), ibilut(*), jbilut(*)
  DOUBLE PRECISION a(*), bilut(*), dpar(*), tol
```

C:

```
void dcsrilut (int *n, double *a, int *ia, int *ja, double *bilut, int *ibilut, int *jbilut,
  double *tol, int *maxfil, int *ipar, double *dpar, int *ierr);
```

Calling Sparse Solver and Preconditioner Routines from C/C++

The calling interface for all of the Intel MKL sparse solver and preconditioner routines is designed to be used easily from FORTRAN 77 or Fortran 90. However, any of these routines can be invoked directly from C or C++ if users are familiar with the inter-language calling conventions of their platforms. These conventions include, but are not limited to, the argument passing mechanisms for the language, the data type mappings from Fortran to C/C++, and the platform specific method of decoration for Fortran external names.

To promote portability, the C header files provide a set of macros and type definitions intended to hide the inter-language calling conventions and provide an interface to the Intel MKL sparse solver routines that appears natural for C/C++.

For example, consider a hypothetical library routine `foo` that takes a real vector of length `n`, and returns an integer status. Fortran users would access such a function as:

```
INTEGER n, status, foo
REAL x(*)
status = foo(x, n)
```

As noted above, to invoke `foo`, C users need to know what C data types correspond to Fortran types `INTEGER` and `REAL`; what argument passing mechanism the Fortran compiler uses; and what, if any, name decoration the Fortran compiler performs when generating the external symbol `foo`.

However, by using the C specific header file, for example `mkl_solver.h`, the invocation of `foo`, within a C program would look as follows:

```
#include "mkl_solver.h"

_INTEGER_t i, status;

_REAL_t x[];

status = foo( x, i );
```

Note that in the above example, the header file `mkl_solver.h` provides definitions for the types `_INTEGER_t` and `_REAL_t` that correspond to the Fortran types `INTEGER` and `REAL`.

To simplify calling of the Intel MKL sparse solver routines from C and C++, the following approach of providing C definitions of Fortran types is used: if an argument or a result from a sparse solver is documented as having the Fortran language specific type `XXX`, then the C and C++ header files provide an appropriate C language type definitions for the name `_XXX_t`.

Caveat for C Users

One of the key differences between C/C++ and Fortran is the argument passing mechanisms for the languages: Fortran programs use pass-by-reference semantics and C/C++ programs use pass-by-value semantics. In the above example, the header file `mkl_solver.h` attempts to hide this difference, by defining a macro `foo`, which takes the address of the appropriate arguments. For example, on the Tru64 UNIX* operating system `mkl_solver.h` defines the macro as follows:

```
#define foo(a,b) foo_(a, &(b))
```

Note how constants are treated when using the macro form of `foo`. `foo( x, 10 )` is converted into `foo_( x, &10 )`. In a strictly ANSI compliant C compiler, taking the address of a constant is not permitted, so a strictly conforming program would look like:

```
INTEGER_t iTen = 10;

_REAL_t * x;

status = foo( x, iTen );
```

However, some C compilers in a non-ANSI compliant mode enable taking the address of a constant for ease of use with Fortran programs. The form `foo( x, 10 )` is acceptable for such compilers.

Vector Mathematical Functions

This chapter describes Vector Mathematical Functions Library (VML), which is designed to compute mathematical functions on vector arguments. VML is an integral part of the Intel® MKL Kernel Library and the VML terminology is used here for simplicity in discussing this group of functions.

VML includes a set of highly optimized implementations of certain computationally expensive core mathematical functions (power, trigonometric, exponential, hyperbolic etc.) that operate on vectors of real and complex numbers.

Application programs that might significantly improve performance with VML include nonlinear programming software, integrals computation, and many others.

VML functions are divided into the following groups according to the operations they perform:

- [VML Mathematical Functions](#) compute values of mathematical functions (such as sine, cosine, exponential, logarithm and so on) on vectors with unit increment indexing.
- [VML Pack/Unpack Functions](#) convert to and from vectors with positive increment indexing, vector indexing and mask indexing (see [Appendix B](#) for details on vector indexing methods).
- [VML Service Functions](#) allow the user to set/get the accuracy mode, and set/get the error code.

VML mathematical functions take an input vector as argument, compute values of the respective function element-wise, and return the results in an output vector.

Data Types and Accuracy Modes

Mathematical and pack/unpack vector functions in VML have been implemented for vector arguments of single and double precision real data. Both Fortran- and C-interfaces to all functions, including VML service functions, are provided in the library. The differences in naming and calling the functions for Fortran- and C-interfaces are detailed in the [Function Naming Conventions](#) section below.

Each vector function from VML (for each data format) can work in three modes: High Accuracy (HA), Low Accuracy (LA), and Enhanced Performance (EP). For many functions, using the LA version improves performance at the cost of slight reduction in accuracy (1-2 least significant bits). In contrast to the LA accuracy flavor, the EP flavor further enhances the performance at the cost of significant reduction in accuracy. In both single and double precision, about half bits of floating-point mantissa are correct. Moreover, subtle argument paths for certain functions (for example, large arguments in trigonometric functions) may be calculated with even less accuracy.

Despite the fact that default accuracy is HA, LA is more than sufficient in most cases. For certain applications that are not very demanding for accuracy (for example, media applications, some Monte Carlo simulations, etc.) you may find the EP accuracy flavor sufficient.

In addition, special value behavior may differ between the HA and LA versions of the functions. Any information on accuracy behavior can be found in the *Intel MKL Release Notes*.

Switching between the two modes (HA, LA, EP) is accomplished by using `vmlSetMode(mode)` (see [Table 9-14](#)). The function `vmlGetMode()` returns the currently used mode. The High Accuracy mode is used by default.

Function Naming Conventions

Full names of all VML functions include only lowercase letters for Fortran-interface, whereas for C-interface names the lowercase letters are mixed with uppercase.

VML mathematical and pack/unpack function full names have the following structure:

`v<p><name><mod>`

The initial letter `v` is a prefix indicating that a function belongs to VML.

The `<p>` field is a precision prefix that indicates the data type:

<code>s</code>	REAL for Fortran-interface, or <code>float</code> for C-interface
<code>d</code>	DOUBLE PRECISION for Fortran-interface, or <code>double</code> for C-interface.
<code>c</code>	COMPLEX for Fortran-interface, or <code>MKL_Complex8</code> for C-interface.
<code>z</code>	DOUBLE COMPLEX for Fortran-interface, or <code>MKL_Complex16</code> for C-interface.

The `<name>` field indicates the function short name, with some of its letters in uppercase for C-interface (see for example [Table 9-2](#) or [Table 9-13](#)).

The `<mod>` field (written in uppercase for C-interface) is present in pack/unpack functions only; it indicates the indexing method used:

<code>i</code>	indexing with positive increment
<code>v</code>	indexing with index vector
<code>m</code>	indexing with mask vector.

VML service function full names have the following structure:

`vml<name>`

where `vml` is a prefix indicating that a function belongs to VML, and `<name>` is the function short name, which includes some uppercase letters for C-interface (see [Table 9-13](#)). To call VML functions from an application program, use conventional function calls. For example, the VML exponential function for single precision real data can be called as

`call vsexp ( n, a, y )` for Fortran-interface, or

`vsExp ( n, a, y );` for C-interface.

Functions Interface

The interface to VML functions includes function full names and the arguments list. The Fortran- and C-interface descriptions for different groups of VML functions are given below. Note that some functions (`Div`, `Pow`, and `Atan2`) have two input vectors *a* and *b* as their arguments, while `SinCos` function has two output vectors *y* and *z*.

VML Mathematical Functions

Fortran:

```
call v<p><name>( n, a, y )
call v<p><name>( n, a, b, y )
call v<p><name>( n, a, y, z )
```

C:

```
v<p><name>( n, a, y );
v<p><name>( n, a, b, y );
v<p><name>( n, a, y, z );
```

Pack Functions

Fortran:

```
call v<p>packi( n, a, inca, y )
call v<p>packv( n, a, ia, y )
call v<p>packm( n, a, ma, y )
```

C:

```
v<p>PackI( n, a, inca, y );
v<p>PackV( n, a, ia, y );
v<p>PackM( n, a, ma, y );
```

Unpack Functions

Fortran:

```
call v<p>unpacki( n, a, y, incy )
call v<p>unpackv( n, a, y, iy )
call v<p>unpackm( n, a, y, my )
```

C:

```
v<p>UnpackI( n, a, y, incy );
v<p>UnpackV( n, a, y, iy );
v<p>UnpackM( n, a, y, my );
```

Service Functions

Fortran:

```
oldmode = vmlsetmode( mode )
mode = vmlgetmode( )
olderr = vmlseterrstatus ( err )
err = vmlgeterrstatus( )
olderr = vmlclearerrstatus( )
oldcallback = vmlseterrorcallback( callback )
callback = vmlgeterrorcallback( )
oldcallback = vmlclearerrorcallback( )
```

C:

```
oldmode = vmlSetMode( mode );
mode = vmlGetMode( void );
olderr = vmlSetErrStatus ( err );
err = vmlGetErrStatus( void );
olderr = vmlClearErrStatus( void );
oldcallback = vmlSetErrorCallBack( callback );
callback = vmlGetErrorCallBack( void );
oldcallback = vmlClearErrorCallBack( void );
```

Input Parameters

<i>n</i>	number of elements to be calculated
<i>a</i>	first input vector
<i>b</i>	second input vector

<i>inca</i>	vector increment for the input vector <i>a</i>
<i>ia</i>	index vector for the input vector <i>a</i>
<i>ma</i>	mask vector for the input vector <i>a</i>
<i>incy</i>	vector increment for the output vector <i>y</i>
<i>iy</i>	index vector for the output vector <i>y</i>
<i>my</i>	mask vector for the output vector <i>y</i>
<i>err</i>	error code
<i>mode</i>	VML mode
<i>callback</i>	address of the callback function

Output Parameters

<i>y</i>	first output vector
<i>z</i>	second output vector
<i>err</i>	error code
<i>mode</i>	VML mode
<i>olderr</i>	former error code
<i>oldmode</i>	former VML mode
<i>callback</i>	address of the callback function
<i>oldcallback</i>	address of the former callback function

The data types of the parameters used in each function are specified in the respective function description section. All VML mathematical functions can perform in-place operations, which means that the same vector can be used as both input and output parameter. This holds true for functions with two input vectors as well, in which case one of them may be overwritten with the output vector. For functions with two output vectors, one of them may coincide with the input vector. But partially overlapping input and output vectors could lead to unpredictable results.

Vector Indexing Methods

Current VML mathematical functions work only with unit increment. Arrays with other increments, or more complicated indexing, can be accommodated by gathering the elements into a contiguous vector and then scattering them after the computation is complete.

Three following indexing methods are used to gather/scatter the vector elements in VML:

- positive increment

- index vector
- mask vector.

The indexing method used in a particular function is indicated by the indexing modifier (see the description of the *<mod>* field in [Function Naming Conventions](#)). For more information on indexing methods see [Vector Arguments in VML](#) in Appendix B.

Error Diagnostics

The VML library has its own error handler. The only difference for C- and Fortran- interfaces is that the Intel MKL error reporting routine `XERBLA` can be called after the Fortran- interface VML function encounters an error, and this routine gets information on `VML_STATUS_BADSIZE` and `VML_STATUS_BADMEM` input errors (see [Table 9-16](#)).

The VML error handler has the following properties:

- The Error Status (`vmlErrStatus`) global variable is set after each VML function call. The possible values of this variable are shown in the [Table 9-16](#).
- Depending on the VML mode, the error handler function invokes:
 - `errno` variable setting. The possible values are shown in the [Table 9-1](#).
 - writing error text information to the `stderr` stream
 - raising the appropriate exception on error, if necessary
 - calling the additional error handler callback function.

Table 9-1. Set Values of the `errno` Variable

Value of <code>errno</code>	Description
0	No errors are detected.
<code>EINVAL</code>	The array dimension is not positive.
<code>EACCES</code>	NULL pointer is passed.
<code>EDOM</code>	At least one of array values is out of a range of definition.
<code>ERANGE</code>	At least one of array values caused a singularity, overflow or underflow.

VML Mathematical Functions

This section describes VML functions which compute values of mathematical functions on real and complex vector arguments with unit increment.

Each function group is introduced by its short name, a brief description of its purpose, and the calling sequence for each type of data both for Fortran- and C-interfaces, as well as a description of the input/output arguments.

For all VML mathematical functions, the input range of parameters is equal to the mathematical range of definition in the set of defined values for the respective data type. Several VML functions, specifically `Div`, `Exp`, `Sinh`, `Cosh`, and `Pow`, can result in an overflow. For these functions, the respective input threshold values that mark off the precision overflow are specified in the function description section. Note that in these specifications, `FLT_MAX` denotes the maximum number representable in single precision real data type, while `DBL_MAX` denotes the maximum number representable in double precision real data type.

Table 9-2 lists available mathematical functions and data types associated with them.

Table 9-2. VML Mathematical Functions

Function	Data Types	Description
Arithmetic Functions		
<code>Add</code>	<i>s, d, c, z</i>	Addition of the vector elements
<code>Sub</code>	<i>s, d, c, z</i>	Subtraction of the vector elements
<code>Sqr</code>	<i>s, d</i>	Squaring of the vector elements
<code>Mul</code>	<i>s, d, c, z</i>	Multiplication of the vector elements
<code>MulByConj</code>	<i>c, z</i>	Multiplication of elements of one vector by conjugated elements of the second vector
<code>Conj</code>	<i>c, z</i>	Conjugation of the vector elements
<code>Abs</code>	<i>s, d, c, z</i>	Absolute value of the vector elements
Power and Root Functions		
<code>Inv</code>	<i>s, d</i>	Inversion of the vector elements
<code>Div</code>	<i>s, d, c, z</i>	Divide elements of one vector by elements of second vector
<code>Sqrt</code>	<i>s, d, c, z</i>	Square root of vector elements
<code>InvSqrt</code>	<i>s, d</i>	Inverse square root of vector elements
<code>Cbrt</code>	<i>s, d</i>	Cube root of vector elements
<code>InvCbrt</code>	<i>s, d</i>	Inverse cube root of vector elements
<code>Pow2o3</code>	<i>s, d</i>	Each vector element raised to 2/3
<code>Pow3o2</code>	<i>s, d</i>	Each vector element raised to 3/2
<code>Pow</code>	<i>s, d, c, z</i>	Each vector element raised to the specified power
<code>Powx</code>	<i>s, d, c, z</i>	Each vector element raised to the constant power
<code>Hypot</code>	<i>s, d</i>	Square root of sum of squares
Exponential and Logarithmic Functions		
<code>Exp</code>	<i>s, d, c, z</i>	Exponential of vector elements
<code>Expml</code>	<i>s, d</i>	Exponential of vector elements decreased by 1
<code>Ln</code>	<i>s, d, c, z</i>	Natural logarithm of vector elements

Function	Data Types	Description
Logl0	<i>s, d, c, z</i>	Denary logarithm of vector elements
Loglp	<i>s, d</i>	Natural logarithm of vector elements that are increased by 1
Trigonometric Functions		
Cos	<i>s, d, c, z</i>	Cosine of vector elements
Sin	<i>s, d, c, z</i>	Sine of vector elements
SinCos	<i>s, d</i>	Sine and cosine of vector elements
CIS	<i>c, z</i>	Complex exponent of vector elements (cosine and sine combined to complex value)
Tan	<i>s, d, c, z</i>	Tangent of vector elements
Acos	<i>s, d, c, z</i>	Inverse cosine of vector elements
Asin	<i>s, d, c, z</i>	Inverse sine of vector elements
Atan	<i>s, d, c, z</i>	Inverse tangent of vector elements
Atan2	<i>s, d</i>	Four-quadrant inverse tangent of elements of two vectors
Hyperbolic Functions		
Cosh	<i>s, d, c, z</i>	Hyperbolic cosine of vector elements
Sinh	<i>s, d, c, z</i>	Hyperbolic sine of vector elements
Tanh	<i>s, d, c, z</i>	Hyperbolic tangent of vector elements
Acosh	<i>s, d, c, z</i>	Inverse hyperbolic cosine of vector elements
Asinh	<i>s, d, c, z</i>	Inverse hyperbolic sine of vector elements
Atanh	<i>s, d, c, z</i>	Inverse hyperbolic tangent of vector elements.
Special Functions		
Erf	<i>s, d</i>	Error function value of vector elements
Erfc	<i>s, d</i>	Complementary error function value of vector elements
ErfInv	<i>s, d</i>	Inverse error function value of vector elements
Rounding Functions		
Floor	<i>s, d</i>	Rounding towards minus infinity
Ceil	<i>s, d</i>	Rounding towards plus infinity
Trunc	<i>s, d</i>	Rounding towards zero infinity
Round	<i>s, d</i>	Rounding to nearest integer
NearbyInt	<i>s, d</i>	Rounding according to current mode
Rint	<i>s, d</i>	Rounding according to current mode and raising inexact result exception
Modf	<i>s, d</i>	Integer and fraction parts

Arithmetic Functions

Add

Performs element by element addition of vector a and vector b .

Syntax

Fortran:

```
call vsadd( n, a, b, y )
call vdadd( n, a, b, y )
call vcadd( n, a, b, y )
call vzadd( n, a, b, y )
```

C:

```
vsAdd( n, a, b, y );
vdAdd( n, a, b, y );
vcAdd( n, a, b, y );
vzAdd( n, a, b, y );
```

Input Parameters

Name	Type	Description
n	FORTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.
a, b	FORTRAN: REAL, INTENT (IN) for vsadd DOUBLE PRECISION, INTENT (IN) for vdadd COMPLEX, INTENT (IN) for vcadd	FORTRAN: Arrays, specify the input vectors a and b . C: Pointers to arrays that contain the input vectors a and b .

Name	Type	Description
	DOUBLE COMPLEX, INTENT(IN) for vzadd	
	C: const float* for vsAdd	
	const double* for vdAdd	
	const MKL_Complex8* for vcAdd	
	const MKL_Complex16* for vzAdd	

Output Parameters

Name	Type	Description
<i>y</i>	FORTTRAN: REAL for vsadd DOUBLE PRECISION for vdadd COMPLEX, for vcadd DOUBLE COMPLEX for vzadd C: float* for vsAdd double* for vdAdd MKL_Complex8* for vcAdd MKL_Complex16* for vzAdd	FORTTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Sub

Performs element by element subtraction of vector *b* from vector *a*.

Syntax

Fortran:

```
call vssub( n, a, b, y )
call vdsb( n, a, b, y )
call vcsub( n, a, b, y )
call vzsub( n, a, b, y )
```

C:

```
vsSub( n, a, b, y );
vdSub( n, a, b, y );
vcSub( n, a, b, y );
vzSub( n, a, b, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.
<i>a, b</i>	FORTRAN: REAL, INTENT (IN) for vssub DOUBLE PRECISION, INTENT (IN) for vdsb COMPLEX, INTENT (IN) for vcsub DOUBLE COMPLEX, INTENT (IN) for vzsub C: const float* for vsSub	FORTRAN: Arrays, specify the input vectors <i>a</i> and <i>b</i> . C: Pointers to arrays that contain the input vectors <i>a</i> and <i>b</i> .

Name	Type	Description
	const double* for vdSub	
	const MKL_Complex8* for vcSub	
	const MKL_Complex16* for vzSub	

Output Parameters

Name	Type	Description
<i>y</i>	FORTTRAN: REAL for vssub DOUBLE PRECISION for vdsusb COMPLEX, for vcsub DOUBLE COMPLEX for vzsub C: float* for vsSub double* for vdSub MKL_Complex8* for vcSub MKL_Complex16* for vzSub	FORTTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Sqr

Performs element by element squaring of the vector.

Syntax

Fortran:

```
call vssqr( n, a, y )
call vdsqr( n, a, y )
```

C:

```
vsSqr( n, a, y );
vdSqr( n, a, y );
```

Input Parameters

Name	Type	Description
n	FORTTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.
a	FORTTRAN: REAL, INTENT (IN) for vssqr DOUBLE PRECISION, INTENT (IN) for vdsqr C: const float* for vsSqr const double* for vdSqr	FORTTRAN: Array, specifies the input vector a . C: Pointer to an array that contains the input vector a .

Output Parameters

Name	Type	Description
y	FORTTRAN: REAL for vssqr DOUBLE PRECISION for vdsqr C: float* for vsSqr double* for vdSqr	FORTTRAN: Array, specifies the output vector y . C: Pointer to an array that contains the output vector y .

Mul

Performs element by element multiplication of vector a and vector b .

Syntax

Fortran:

```
call vsmul( n, a, b, y )
call vdmul( n, a, b, y )
call vcmul( n, a, b, y )
call vzmul( n, a, b, y )
```

C:

```
vsMul( n, a, b, y );
vdMul( n, a, b, y );
vcMul( n, a, b, y );
vzMul( n, a, b, y );
```

Input Parameters

Name	Type	Description
n	FORTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.
a, b	FORTRAN: REAL, INTENT (IN) for vsmul DOUBLE PRECISION, INTENT (IN) for vdmul COMPLEX, INTENT (IN) for vcmul DOUBLE COMPLEX, INTENT (IN) for vzmul C: const float* for vsMul	FORTRAN: Arrays, specify the input vectors a and b . C: Pointers to arrays that contain the input vectors a and b .

Name	Type	Description
	const double* for vdMul	
	const MKL_Complex8* for vcMul	
	const MKL_Complex16* for vzMul	

Output Parameters

Name	Type	Description
y	FORTTRAN: REAL for vsmul DOUBLE PRECISION for vdmul COMPLEX, for vcmul DOUBLE COMPLEX for vzmul C: float* for vsMul double* for vdMul MKL_Complex8* for vcMul MKL_Complex16* for vzMul	FORTTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

MulByConj

*Performs element by element multiplication of vector *a* element and conjugated vector *b* element.*

Syntax

Fortran:

```
call vcmulbyconj( n, a, b, y )  
call vzmulbyconj( n, a, b, y )
```

C:

```
vcMulByConj( n, a, b, y );  
vzMulByConj( n, a, b, y );
```

Input Parameters

Name	Type	Description
n	FORTTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.
a, b	FORTTRAN: COMPLEX, INTENT (IN) for vcmulbyconj DOUBLE COMPLEX, INTENT (IN) for vzmulbyconj C: const MKL_Complex8* for vcMulByConj const MKL_Complex16* for vzMulByConj	FORTTRAN: Arrays, specify the input vectors a and b . C: Pointers to arrays that contain the input vectors a and b .

Output Parameters

Name	Type	Description
y	FORTTRAN: COMPLEX, for vcmulbyconj DOUBLE COMPLEX for vzmulbyconj C: MKL_Complex8* for vcMulByConj MKL_Complex16* for vzMulByConj	FORTTRAN: Array, specifies the output vector y . C: Pointer to an array that contains the output vector y .

Conj

Performs element by element conjugation of the vector.

Syntax

Fortran:

```
call vcconj( n, a, y )
call vzconj( n, a, y )
```

C:

```
vcConj( n, a, y );
vzConj( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN: INTEGER, INTENT(IN) C: int	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN: COMPLEX, INTENT(IN) for vcconj DOUBLE COMPLEX, INTENT(IN) for vzconj C: const MKL_Complex8* for vcConj const MKL_Complex16* for vzConj	FORTRAN: Arrays, specify the input vector <i>a</i> . C: Pointers to arrays that contain the input vector <i>a</i> .

Output Parameters

Name	Type	Description
<i>y</i>	FORTTRAN: COMPLEX, for <code>vcconj</code> DOUBLE COMPLEX for <code>vzconj</code> C: MKL_Complex8* for <code>vcConj</code> MKL_Complex16* for <code>vzConj</code>	FORTTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Abs

Computes absolute value of vector elements.

Syntax

Fortran:

```
call vsabs( n, a, y )
call vdabs( n, a, y )
call vcabs( n, a, y )
call vzabs( n, a, y )
```

C:

```
vsAbs( n, a, y );
vdAbs( n, a, y );
vcAbs( n, a, y );
vzAbs( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.

Name	Type	Description
<i>a</i>	FORTTRAN: REAL, INTENT (IN) for vsabs DOUBLE PRECISION, INTENT (IN) for vdabs COMPLEX, INTENT (IN) for vcabs DOUBLE COMPLEX, INTENT (IN) for vzabs C: const float* for vsAbs const double* for vdAbs const MKL_Complex8* for vcAbs const MKL_Complex16* for vzAbs	FORTTRAN: Arrays, specify the input vector <i>a</i> . C: Pointers to arrays that contain the input vector <i>a</i> .

Output Parameters

Name	Type	Description
<i>y</i>	FORTTRAN: REAL for vsabs, vcabs DOUBLE PRECISION for vdabs, vzabs C: float* for vsAbs, vcAbs double* for vdAbs, vzAbs	FORTTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Power and Root Functions

Inv

Performs element by element inversion of the vector.

Syntax

Fortran:

```
call vsinv( n, a, y )
call vdinv( n, a, y )
```

C:

```
vsInv( n, a, y );
vdInv( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.
<i>a</i>	FORTTRAN: REAL, INTENT (IN) for vsinv DOUBLE PRECISION, INTENT (IN) for vdinv C: const float* for vsInv const double* for vdInv	FORTTRAN: Array, specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Output Parameters

Name	Type	Description
<i>y</i>	FORTTRAN: REAL for vsinv DOUBLE PRECISION for vdinv C: float* for vsInv double* for vdInv	FORTTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Div

Performs element by element division of vector a by vector b

Syntax

Fortran:

```
call vsdiv( n, a, b, y )  
call vddiv( n, a, b, y )  
call vcdiv( n, a, b, y )  
call vzdiv( n, a, b, y )
```

C:

```
vsDiv( n, a, b, y );  
vdDiv( n, a, b, y );  
vcDiv( n, a, b, y );  
vzDiv( n, a, b, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.

Name	Type	Description
<i>a, b</i>	FORTRAN: REAL, INTENT (IN) for vsdiv DOUBLE PRECISION, INTENT (IN) for vddiv COMPLEX, INTENT (IN) for vcdiv DOUBLE COMPLEX, INTENT (IN) for vzdiv C: const float* for vsDiv const double* for vdDiv const MKL_Complex8* for vcDiv const MKL_Complex16* for vzDiv	FORTRAN: Arrays, specify the input vectors <i>a</i> and <i>b</i> . C: Pointers to arrays that contain the input vectors <i>a</i> and <i>b</i> .

Table 9-3 Precision Overflow Thresholds for Div Real Function

Data Type	Threshold Limitations on Input Parameters
single precision	$\text{abs}(a[i]) < \text{abs}(b[i]) * \text{FLT\_MAX}$
double precision	$\text{abs}(a[i]) < \text{abs}(b[i]) * \text{DBL\_MAX}$



NOTE. Overflow can occur also in Div complex function, but the exact formula is beyond the scope of this document.

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN: REAL for vsdiv DOUBLE PRECISION for vddiv COMPLEX for vcdiv DOUBLE COMPLEX for vzdiv C: float* for vsDiv	FORTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Name	Type	Description
	double* for vdDiv	
	MKL_Complex8* for vcDiv	
	MKL_Complex16* for vzDiv	

Sqrt

Computes a square root of vector elements.

Syntax

Fortran:

```
call vssqrt( n, a, y )
call vdsqrt( n, a, y )
call vcsqrt( n, a, y )
call vzsqrt( n, a, y )
```

C:

```
vsSqrt( n, a, y );
vdSqrt( n, a, y );
vcSqrt( n, a, y );
vzSqrt( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN: INTEGER, INTENT(IN) C: int	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN: REAL, INTENT(IN) for vssqrt DOUBLE PRECISION, INTENT(IN) for vdsqrt C: Pointer to an array that contains the input vector <i>a</i> .	FORTRAN: Array, specifies the input vector <i>a</i> .

Name	Type	Description
	COMPLEX, INTENT(IN) for vcsqrt	
	DOUBLE COMPLEX, INTENT(IN) for vzsqrt	
	C: const float* for vsSqrt	
	const double* for vdSqrt	
	const MKL_Complex8* for vcSqrt	
	const MKL_Complex16* for vzSqrt	

Output Parameters

Name	Type	Description
<i>y</i>	FORTTRAN: REAL for vssqrt DOUBLE PRECISION for vdsqrt COMPLEX for vcsqrt DOUBLE COMPLEX for vzsqrt C: float* for vsSqrt double* for vdSqrt MKL_Complex8* for vcSqrt MKL_Complex16* for vzSqrt	FORTTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

InvSqrt

Computes an inverse square root of vector elements.

Syntax

Fortran:

```
call vsinvsqrt( n, a, y )
call vdinvsqrt( n, a, y )
```

C:

```
vsInvSqrt( n, a, y );
vdInvSqrt( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN: INTEGER, INTENT(IN) C: int	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN: REAL, INTENT(IN) for vsinvsqrt DOUBLE PRECISION, INTENT(IN) for vdinvsqrt C: const float* for vsInvSqrt const double* for vdInvSqrt	FORTRAN: Array, specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN: REAL for vsinvsqrt DOUBLE PRECISION for vdinvsqrt	FORTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Name	Type	Description
	C: float* for vsInvSqrt	
	double* for vdInvSqrt	

Cbrt

Computes a cube root of vector elements.

Syntax

Fortran:

```
call vscbrt( n, a, y )
call vdcbrt( n, a, y )
```

C:

```
vsCbrt( n, a, y );
vdCbrt( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.
<i>a</i>	FORTTRAN: REAL, INTENT (IN) for vscbrt DOUBLE PRECISION, INTENT (IN) for vdcbrt C: const float* for vsCbrt const double* for vdCbrt	FORTTRAN: Array, specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN: REAL for <code>vsqrt</code> DOUBLE PRECISION for <code>vdsqrt</code> C: <code>float*</code> for <code>vsqrt</code> <code>double*</code> for <code>vdsqrt</code>	FORTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

InvCbrt

Computes an inverse cube root of vector elements.

Syntax

Fortran:

```
call vsinvcbrrt( n, a, y )  
call vdsinvcbrrt( n, a, y )
```

C:

```
vsInvCbrrt( n, a, y );  
vdInvCbrrt( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN: INTEGER, INTENT(IN) C: <code>int</code>	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN: REAL, INTENT(IN) for <code>vsinvcbrrt</code> DOUBLE PRECISION, INTENT(IN) for <code>vdsinvcbrrt</code> C: <code>const float*</code> for <code>vsInvCbrrt</code>	FORTRAN: Array, specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Name	Type	Description
	<code>const double*</code> for <code>vdInvCbrt</code>	

Output Parameters

Name	Type	Description
<code>y</code>	FORTRAN: <code>REAL</code> for <code>vsinvcbtrt</code> <code>DOUBLE PRECISION</code> for <code>vdinvcbtrt</code> C: <code>float*</code> for <code>vsInvCbrt</code> <code>double*</code> for <code>vdInvCbrt</code>	FORTRAN: Array, specifies the output vector <code>y</code> . C: Pointer to an array that contains the output vector <code>y</code> .

Pow2o3

Raises each element of a vector to the constant power 2/3.

Syntax

Fortran:

```
call vspow2o3( n, a, y )
call vdpow2o3( n, a, y )
```

C:

```
vsPow2o3( n, a, y );
vdPow2o3( n, a, y );
```

Input Parameters

Name	Type	Description
<code>n</code>	FORTRAN: <code>INTEGER, INTENT (IN)</code> C: <code>int</code>	Specifies the number of elements to be calculated.

Name	Type	Description
<i>a, b</i>	FORTRAN: REAL, INTENT (IN) for vspow2o3 DOUBLE PRECISION, INTENT (IN) for vdpow2o3 C: const float* for vsPow2o3 const double* for vdPow2o3	FORTRAN: Arrays, specify the input vector <i>a</i> . C: Pointers to arrays that contain the input vector <i>a</i> .

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN: REAL for vspow2o3 DOUBLE PRECISION for vdpow2o3 C: float* for vsPow2o3 double* for vdPow2o3	FORTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Pow3o2

Raises each element of a vector to the constant power 3/2.

Syntax

Fortran:

```
call vspow3o2( n, a, y )  
call vdpow3o2( n, a, y )
```

C:

```
vsPow3o2( n, a, y );  
vdPow3o2( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.
<i>a, b</i>	FORTRAN: REAL, INTENT (IN) for vspow3o2 DOUBLE PRECISION, INTENT (IN) for vdpow3o2 C: const float* for vsPow3o2 const double* for vdPow3o2	FORTRAN: Arrays, specify the input vector <i>a</i> . C: Pointers to arrays that contain the input vector <i>a</i> .

Table 9-4 Precision Overflow Thresholds for Pow3o2 Function

Data Type	Threshold Limitations on Input Parameters
single precision	$\text{abs}(a[i]) < (\text{FLT\_MAX})^{2/3}$
double precision	$\text{abs}(a[i]) < (\text{DBL\_MAX})^{2/3}$

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN: REAL for vspow3o2 DOUBLE PRECISION for vdpow3o2 C: float* for vsPow3o2 double* for vdPow3o2	FORTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Pow

Computes a to the power b for elements of two vectors.

Syntax

Fortran:

```
call vspow( n, a, b, y )
call vdpow( n, a, b, y )
call vcpow( n, a, b, y )
call vzipow( n, a, b, y )
```

C:

```
vsPow( n, a, b, y );
vdPow( n, a, b, y );
vcPow( n, a, b, y );
vzPow( n, a, b, y );
```

Input Parameters

Name	Type	Description
n	FORTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.
a, b	FORTRAN: REAL, INTENT (IN) for vspow DOUBLE PRECISION, INTENT (IN) for vdpow COMPLEX, INTENT (IN) for vcpow DOUBLE COMPLEX, INTENT (IN) for vzipow C: const float* for vsPow	FORTRAN: Arrays, specify the input vectors a and b . C: Pointers to arrays that contain the input vectors a and b .

Name	Type	Description
	const double* for vdPow	
	const MKL_Complex8* for vcPow	
	const MKL_Complex16* for vzPow	

Table 9-5 Precision Overflow Thresholds for Pow Real Function

Data Type	Threshold Limitations on Input Parameters
single precision	$\text{abs}(a[i]) < (\text{FLT\_MAX})^{1/b[i]}$
double precision	$\text{abs}(a[i]) < (\text{DBL\_MAX})^{1/b[i]}$



NOTE. Overflow can occur also in `Pow` complex function, but the exact formula is beyond the scope of this document.

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN: REAL for vspow DOUBLE PRECISION for vdpow COMPLEX for vcpow DOUBLE COMPLEX for vzpow C: float* for vsPow double* for vdPow MKL_Complex8* for vcPow MKL_Complex16* for vzPow	FORTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Description

The real function `Pow` has certain limitations on the input range of *a* and *b* parameters. Specifically, if *a*[*i*] is positive, then *b*[*i*] may be arbitrary. For negative *a*[*i*], the value of *b*[*i*] must be integer (either positive or negative).

The complex function `POW` has no such input range limitations.

Powx

Raises each element of a vector to the constant power.

Syntax

Fortran:

```
call vspowx( n, a, b, y )
call vdpowx( n, a, b, y )
call vcpowx( n, a, b, y )
call vzipowx( n, a, b, y )
```

C:

```
vsPowx( n, a, b, y );
vdPowx( n, a, b, y );
vcPowx( n, a, b, y );
vzPowx( n, a, b, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Number of elements to be calculated.
<i>a</i>	FORTTRAN: REAL, INTENT (IN) for vspowx DOUBLE PRECISION, INTENT (IN) for vdpowx COMPLEX, INTENT (IN) for vcpowx DOUBLE COMPLEX, INTENT (IN) for vzipowx	FORTTRAN: Array <i>a</i> that specifies the input vector C: Pointer to an array that contains the input vector <i>a</i> .

Name	Type	Description
	C: <code>const float*</code> for <code>vsPowx</code> <code>const double*</code> for <code>vdPowx</code> <code>const MKL_Complex8*</code> for <code>vcPowx</code> <code>const MKL_Complex16*</code> for <code>vzPowx</code>	
b	FORTTRAN: <code>REAL, INTENT (IN)</code> for <code>vspowx</code> <code>DOUBLE PRECISION, INTENT (IN)</code> for <code>vdpowx</code> <code>COMPLEX, INTENT (IN)</code> for <code>vcpowx</code> <code>DOUBLE COMPLEX, INTENT (IN)</code> for <code>vzpowx</code> C: <code>const float</code> for <code>vsPowx</code> <code>const double</code> for <code>vdPowx</code> <code>const MKL_Complex8*</code> for <code>vcPowx</code> <code>const MKL_Complex16*</code> for <code>vzPowx</code>	FORTTRAN: Scalar value b that is the constant power. C: Constant value for power b .

Table 9-6 Precision Overflow Thresholds for Powx Real Function

Data Type	Threshold Limitations on Input Parameters
single precision	$\text{abs}(a[i]) < (\text{FLT\_MAX})^{1/b}$
double precision	$\text{abs}(a[i]) < (\text{DBL\_MAX})^{1/b}$



NOTE. Overflow can occur also in `Powx` complex function, but the exact formula is beyond the scope of this document.

Output Parameters

Name	Type	Description
y	FORTRAN: REAL for vspowx DOUBLE PRECISION for vdpowx COMPLEX for vcpowx DOUBLE COMPLEX for vzpowx C: float* for vsPowx double* for vdPowx MKL_Complex8* for vcPowx MKL_Complex16* for vzPowx	FORTRAN: Array, specifies the output vector y . C: Pointer to an array that contains the output vector y .

Description

The real function `Powx` has certain limitations on the input range of a and b parameters. Specifically, if $a[i]$ is positive, then b may be arbitrary. For negative $a[i]$, the value of b must be integer (either positive or negative).

The complex function `Powx` has no such input range limitations.

Hypot

Computes a square root of sum of two squared elements.

Syntax

Fortran:

```
call vshypot( n, a, b, y )  
call vdhypot( n, a, b, y )
```

C:

```
vsHypot( n, a, b, y );  
vdHypot( n, a, b, y );
```

Input Parameters

Name	Type	Description
n	FORTTRAN: INTEGER, INTENT (IN) C: int	Number of elements to be calculated.
a, b	FORTTRAN: REAL, INTENT (IN) for vshypot DOUBLE PRECISION, INTENT (IN) for vdhypot C: const float* for vsHypot const double* for vdHypot	FORTTRAN: Arrays that specify the input vectors a and b C: Pointers to arrays that contain the input vectors a and b .

Table 9-7 Precision Overflow Thresholds for Hypot Function

Data Type	Threshold Limitations on Input Parameters
single precision	$\text{abs}(a[i]) < \text{sqrt}(\text{FLT\_MAX})$ $\text{abs}(b[i]) < \text{sqrt}(\text{FLT\_MAX})$
double precision	$\text{abs}(a[i]) < \text{sqrt}(\text{DBL\_MAX})$ $\text{abs}(b[i]) < \text{sqrt}(\text{DBL\_MAX})$

Output Parameters

Name	Type	Description
y	FORTTRAN: REAL for vshypot DOUBLE PRECISION for vdhypot C: float* for vsHypot double* for vdHypot	FORTTRAN: Array, specifies the output vector y . C: Pointer to an array that contains the output vector y .

Exponential and Logarithmic Functions

Exp

Computes an exponential of vector elements.

Syntax

Fortran:

```
call vsexp( n, a, y )
call vdexp( n, a, y )
call vcexp( n, a, y )
call vzexp( n, a, y )
```

C:

```
vsExp( n, a, y );
vdExp( n, a, y );
vcExp( n, a, y );
vzExp( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN: REAL, INTENT (IN) for vsexp DOUBLE PRECISION, INTENT (IN) for vdexp COMPLEX, INTENT (IN) for vcexp	FORTRAN: Array, specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Name	Type	Description
	DOUBLE COMPLEX, INTENT (IN) for vzexp	
	C: const float* for vsExp	
	const double* for vdExp	
	const MKL_Complex8* for vcExp	
	const MKL_Complex16* for vzExp	

Table 9-8 Precision Overflow Thresholds for Exp Real Function

Data Type	Threshold Limitations on Input Parameters
single precision	$a[i] < \text{Ln}(\text{FLT\_MAX})$
double precision	$a[i] < \text{Ln}(\text{DBL\_MAX})$



NOTE. Overflow can occur also in `Exp` complex function, but the exact formula is beyond the scope of this document.

Output Parameters

Name	Type	Description
<i>y</i>	FORTTRAN: REAL for vsexp DOUBLE PRECISION for vdexp COMPLEX for vcexp DOUBLE COMPLEX for vzexp C: float* for vsExp double* for vdExp MKL_Complex8* for vcExp MKL_Complex16* for vzExp	FORTTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Expm1

Computes an exponential of vector elements decreased by 1.

Syntax

Fortran:

```
call vsexpm1( n, a, y )
call vdexpm1( n, a, y )
```

C:

```
vsExpm1( n, a, y );
vdExpm1( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN: REAL, INTENT (IN) for vsexpm1 DOUBLE PRECISION, INTENT (IN) for vdexpm1 C: const float* for vsExpm1 const double* for vdExpm1	FORTRAN: Array, specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Table 9-9 Precision Overflow Thresholds for Expm1 Function

Data Type	Threshold Limitations on Input Parameters
single precision	$a[i] < \text{Ln}(\text{FLT\_MAX})$
double precision	$a[i] < \text{Ln}(\text{DBL\_MAX})$

Output Parameters

Name	Type	Description
<i>y</i>	FORTTRAN: REAL for vsexpm1 DOUBLE PRECISION for vdexpm1 C: float* for vsExpml double* for vdExpml	FORTTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Ln

Computes natural logarithm of vector elements.

Syntax

Fortran:

```
call vsln( n, a, y )
call vdln( n, a, y )
call vcLn( n, a, y )
call vzln( n, a, y )
```

C:

```
vsLn( n, a, y );
vdLn( n, a, y );
vcLn( n, a, y );
vzLn( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.

Name	Type	Description
<i>a</i>	FORTTRAN: REAL, INTENT (IN) for vsln	FORTTRAN: Array, specifies the input vector <i>a</i> .
	DOUBLE PRECISION, INTENT (IN) for vdl n	
	COMPLEX, INTENT (IN) for vcln	C: Pointer to an array that contains the input vector <i>a</i> .
	DOUBLE COMPLEX, INTENT (IN) for vzln	
	C: const float* for vsLn	
	const double* for vdLn	
	const MKL_Complex8* for vcLn	
	const MKL_Complex16* for vzLn	

Output Parameters

Name	Type	Description
<i>y</i>	FORTTRAN: REAL for vsln	FORTTRAN: Array, specifies the output vector <i>y</i> .
	DOUBLE PRECISION for vdl n	
	COMPLEX for vcln	C: Pointer to an array that contains the output vector <i>y</i> .
	DOUBLE COMPLEX for vzln	
	C: float* for vsLn	
	double* for vdLn	
	MKL_Complex8* for vcLn	
	MKL_Complex16* for vzLn	

Log10

Computes denary logarithm of vector elements.

Syntax

Fortran:

```
call vslog10( n, a, y )
call vdlog10( n, a, y )
call vclog10( n, a, y )
call vzlog10( n, a, y )
```

C:

```
vsLog10( n, a, y );
vdLog10( n, a, y );
vcLog10( n, a, y );
vzLog10( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN: INTEGER, INTENT(IN) C: int	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN: REAL, INTENT(IN) for vslog10 DOUBLE PRECISION, INTENT(IN) for vdlog10 COMPLEX, INTENT(IN) for vclog10 DOUBLE COMPLEX, INTENT(IN) for vzlog10 C: const float* for vsLog10	FORTRAN: Array, specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Name	Type	Description
	const double* for vdLog10	
	const MKL_Complex8* for vcLog10	
	const MKL_Complex16* for vzLog10	

Output Parameters

Name	Type	Description
y	FORTTRAN: REAL for vslog10 DOUBLE PRECISION for vdlog10 COMPLEX for vclog10 DOUBLE COMPLEX for vzlog10 C: float* for vsLog10 double* for vdLog10 MKL_Complex8* for vcLog10 MKL_Complex16* for vzLog10	FORTTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Log1p

Computes natural logarithm of vector elements that are increased by 1.

Syntax

Fortran:

```
call vslog1p( n, a, y )  
call vdlog1p( n, a, y )
```

C:

```
vsLog1p( n, a, y );
```

```
vdLog1p( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.
<i>a</i>	FORTTRAN: REAL, INTENT (IN) for vslog1p DOUBLE PRECISION, INTENT (IN) for vdlog1p C: const float* for vsLog1p const double* for vdLog1p	FORTTRAN: Array, specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Output Parameters

Name	Type	Description
<i>y</i>	FORTTRAN: REAL for vslog1p DOUBLE PRECISION for vdlog1p C: float* for vsLog1p double* for vdLog1p	FORTTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Trigonometric Functions

Cos

Computes cosine of vector elements.

Syntax

Fortran:

```
call vscos( n, a, y )
call vdcos( n, a, y )
call vccos( n, a, y )
call vzcoss( n, a, y )
```

C:

```
vsCos( n, a, y );
vdCos( n, a, y );
vcCos( n, a, y );
vzCos( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.
<i>a</i>	FORTTRAN: REAL, INTENT (IN) for vscos DOUBLE PRECISION, INTENT (IN) for vdcos COMPLEX, INTENT (IN) for vccos	FORTTRAN: Array, specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Name	Type	Description
	DOUBLE PRECISION, INTENT(IN) for vzcos	
	C: const float* for vsCos	
	const double* for vdCos	
	const MKL_Complex8* for vcCos	
	const MKL_Complex16* for vzCos	

Output Parameters

Name	Type	Description
<i>y</i>	FORTTRAN: REAL for vscos DOUBLE PRECISION for vdcos COMPLEX for vccos DOUBLE COMPLEX for vzcos C: float* for vsCos double* for vdCos MKL_Complex8* for vcCos MKL_Complex16* for vzCos	FORTTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Description

The function computes sine of vector elements.

Note that arguments $\text{abs}(a[i]) \leq 2^{13}$ and $\text{abs}(a[i]) \leq 2^{16}$ for single and double precisions respectively are called fast computational path. These are trig function arguments for which VML provides the best possible performance. For performance reasons, avoid arguments that do not belong to the fast computational path in the VML HA and LA functions. Alternatively, you can use VML EP functions that are fast on the entire function domain at the cost of accuracy.

Sin

Computes sine of vector elements.

Syntax

Fortran:

```
call vssin( n, a, y )
call vdsin( n, a, y )
call vcsin( n, a, y )
call vzsine( n, a, y )
```

C:

```
vsSin( n, a, y );
vdSin( n, a, y );
vcSin( n, a, y );
vzSin( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN: INTEGER, INTENT (IN) . C: int.	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN: REAL, INTENT (IN) for vssin DOUBLE PRECISION, INTENT (IN) for vdsin COMPLEX, INTENT (IN) for vcsin DOUBLE PRECISION, INTENT (IN) for vzsine C: const float* for vsSin	FORTRAN: Array, specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Name	Type	Description
	const double* for vdSin	
	const MKL_Complex8* for vcSin	
	const MKL_Complex16* for vzSin	

Output Parameters

Name	Type	Description
<i>y</i>	FORTTRAN: REAL for vssin DOUBLE PRECISION for vdsin COMPLEX for vcsin DOUBLE COMPLEX for vzsine C: float* for vsSin double* for vdSin MKL_Complex8* for vcSin MKL_Complex16* for vzSin	FORTTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Description

The function computes sine of vector elements.

Note that arguments $\text{abs}(a[i]) \leq 2^{13}$ and $\text{abs}(a[i]) \leq 2^{16}$ for single and double precisions respectively are called fast computational path. These are trig function arguments for which VML provides the best possible performance. For performance reasons, avoid arguments that do not belong to the fast computational path in the VML HA and LA functions. Alternatively, you can use VML EP functions that are fast on the entire function domain at the cost of accuracy.

SinCos

Computes sine and cosine of vector elements.

Syntax

Fortran:

```
call vssincos( n, a, y, z )
call vdsincos( n, a, y, z )
```

C:

```
vsSinCos( n, a, y, z );
vdSinCos( n, a, y, z );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTTRAN: INTEGER, INTENT (IN) . C: int.	Specifies the number of elements to be calculated.
<i>a</i>	FORTTRAN: REAL, INTENT (IN) for vssincos DOUBLE PRECISION, INTENT (IN) for vdsincos C: const float* for vsSinCos const double* for vdSinCos	FORTTRAN: Array, specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Output Parameters

Name	Type	Description
<i>y, z</i>	FORTTRAN: REAL for vssincos DOUBLE PRECISION for vdsincos C: float* for vsSinCos	FORTTRAN: Arrays, specify the output vectors <i>y</i> (for sine values) and <i>z</i> (for cosine values).

Name	Type	Description
	double* for vdSinCos	C: Pointers to arrays that contain the output vectors <i>y</i> (for sinevalues) and <i>z</i> (for cosine values).

Description

The function computes sine of vector elements.

Note that arguments $\text{abs}(a[i]) \leq 2^{13}$ and $\text{abs}(a[i]) \leq 2^{16}$ for single and double precisions respectively are called fast computational path. These are trig function arguments for which VML provides the best possible performance. For performance reasons, avoid arguments that do not belong to the fast computational path in the VML HA and LA functions. Alternatively, you can use VML EP functions that are fast on the entire function domain at the cost of accuracy.

CIS

Computes complex exponent of real vector elements (cosine and sine of real vector elements combined to complex value).

Syntax

Fortran:

```
call vccis( n, a, y )
call vzcis( n, a, y )
```

C:

```
vcCIS( n, a, y );
vzCIS( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.

Name	Type	Description
<i>a</i>	FORTRAN: REAL, INTENT (IN) for <code>vccis</code> DOUBLE PRECISION, INTENT (IN) for <code>vzcis</code> C: <code>const float*</code> for <code>vcCIS</code> <code>const double*</code> for <code>vzCIS</code>	FORTRAN: Arrays, specify the input vector <i>a</i> . C: Pointers to arrays that contain the input vector <i>a</i> .

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN: COMPLEX for <code>vccis</code> DOUBLE COMPLEX for <code>vzcis</code> C: <code>MKL_Complex8*</code> for <code>vcCIS</code> <code>MKL_Complex16*</code> for <code>vzCIS</code>	FORTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Tan

Computes tangent of vector elements.

Syntax

Fortran:

```
call vstan( n, a, y )  
call vdtan( n, a, y )  
call vctan( n, a, y )  
call vztan( n, a, y )
```

C:

```
vsTan( n, a, y );
vdTan( n, a, y );
vcTan( n, a, y );
vzTan( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN: REAL, INTENT (IN) for vstan DOUBLE PRECISION, INTENT (IN) for vdtan COMPLEX, INTENT (IN) for vctan DOUBLE COMPLEX, INTENT (IN) for vztan C: const float* for vsTan const double* for vdTan const MKL_Complex8* for vcTan const MKL_Complex16* for vzTan	FORTRAN: Array, specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN: REAL for vstan DOUBLE PRECISION for vdtan COMPLEX for vctan	FORTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Name	Type	Description
	DOUBLE COMPLEX for vzTan	
	C: float* for vsTan	
	double* for vdTan	
	MKL_Complex8* for vcTan	
	MKL_Complex16* for vzTan	

Description

The function computes sine of vector elements.

Note that arguments $\text{abs}(a[i]) \leq 2^{13}$ and $\text{abs}(a[i]) \leq 2^{16}$ for single and double precisions respectively are called fast computational path. These are trig function arguments for which VML provides the best possible performance. For performance reasons, avoid arguments that do not belong to the fast computational path in the VML HA and LA functions. Alternatively, you can use VML EP functions that are fast on the entire function domain at the cost of accuracy.

Acos

Computes inverse cosine of vector elements.

Syntax

Fortran:

```
call vsacos( n, a, y )
call vdacos( n, a, y )
call vcacos( n, a, y )
call vzacos( n, a, y )
```

C:

```
vsAcos( n, a, y );
vdAcos( n, a, y );
vcAcos( n, a, y );
vzAcos( n, a, y );
```

Input Parameters

Name	Type	Description
n	FORTTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.
a	FORTTRAN: REAL, INTENT (IN) for vsacos DOUBLE PRECISION, INTENT (IN) for vdacos COMPLEX, INTENT (IN) for vcacos DOUBLE COMPLEX, INTENT (IN) for vzaeos C: const float* for vsAcos const double* for vdAcos const MKL_Complex8* for vcAcos const MKL_Complex16* for vzAcos	FORTTRAN: Array, specifies the input vector a . C: Pointer to an array that contains the input vector a .

Output Parameters

Name	Type	Description
y	FORTTRAN: REAL for vsacos DOUBLE PRECISION for vdacos COMPLEX for vcacos DOUBLE COMPLEX for vzaeos C: float* for vsAcos double* for vdAcos MKL_Complex8* for vcAcos	FORTTRAN: Array, specifies the output vector y . C: Pointer to an array that contains the output vector y .

Name	Type	Description
	MKL_Complex16* for vzAcos	

Asin

Computes inverse sine of vector elements.

Syntax

Fortran:

```
call vsasin( n, a, y )
call vdasin( n, a, y )
call vcasin( n, a, y )
call vzasin( n, a, y )
```

C:

```
vsAsin( n, a, y );
vdAsin( n, a, y );
vcAsin( n, a, y );
vzAsin( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTTRAN: INTEGER, INTENT(IN) C: int	Specifies the number of elements to be calculated.
<i>a</i>	FORTTRAN: REAL, INTENT(IN) for vsasin DOUBLE PRECISION, INTENT(IN) for vdasin COMPLEX, INTENT(IN) for vcasin	FORTTRAN: Array, specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Name	Type	Description
	DOUBLE COMPLEX, INTENT(IN) for vzasin	
	C: const float* for vsAsin const double* for vdAsin const MKL_Complex8* for vcAsin const MKL_Complex16* for vzAsin	

Output Parameters

Name	Type	Description
<i>y</i>	FORTTRAN: REAL for vsasin DOUBLE PRECISION for vdasin COMPLEX for vcasin DOUBLE COMPLEX for vzasin C: float* for vsAsin double* for vdAsin MKL_Complex8* for vcAsin MKL_Complex16* for vzAsin	FORTTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Atan

Computes inverse tangent of vector elements.

Syntax

Fortran:

```
call vsatan( n, a, y )
call vdatan( n, a, y )
call vcatan( n, a, y )
call vzatan( n, a, y )
```

C:

```
vsAtan( n, a, y );
vdAtan( n, a, y );
vcAtan( n, a, y );
vzAtan( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN: REAL, INTENT (IN) for vsatan DOUBLE PRECISION, INTENT (IN) for vdatan COMPLEX, INTENT (IN) for vcatan DOUBLE COMPLEX, INTENT (IN) for vzatan C: const float* for vsAtan const double* for vdAsin	FORTRAN: Array, specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Name	Type	Description
	const MKL_Complex8* for vcAtan	
	const MKL_Complex16* for vzAsin	

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN: REAL for vsatan DOUBLE PRECISION for vdatan COMPLEX for vcatan DOUBLE COMPLEX for vzatan C: float* for vsAtan double* for vdAtan MKL_Complex8* for vcAtan MKL_Complex16* for vzAtan	FORTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Atan2

Computes four-quadrant inverse tangent of elements of two vectors.

Syntax

Fortran:

```
call vsatan2( n, a, b, y )
call vdatan2( n, a, b, y )
```

C:

```
vsAtan2( n, a, b, y );
vdAtan2( n, a, b, y );
```

Input Parameters

Name	Type	Description
n	FORTRAN: INTEGER, INTENT(IN) C: int	Specifies the number of elements to be calculated.
a, b	FORTRAN: REAL, INTENT(IN) for vsatan2 DOUBLE PRECISION, INTENT(IN) for vdatan2 C: const float* for vsAtan2 const double* for vdAtan2	FORTRAN: Arrays, specify the input vectors a and b . C: Pointers to arrays that contain the input vectors a and b .

Output Parameters

Name	Type	Description
y	FORTRAN: REAL for vsatan2 DOUBLE PRECISION for vdatan2 C: float* for vsAtan2 double* for vdAtan2	FORTRAN: Array, specifies the output vector y . C: Pointer to an array that contains the output vector y .

Description

The elements of the output vector y are computed as the four-quadrant arctangent of $a[i]/b[i]$.

Hyperbolic Functions

Cosh

Computes hyperbolic cosine of vector elements.

Syntax

Fortran:

```
call vscosh( n, a, y )
call vdcosh( n, a, y )
call vccosh( n, a, y )
call vzcosh( n, a, y )
```

C:

```
vsCosh( n, a, y );
vdCosh( n, a, y );
vcCosh( n, a, y );
vzCosh( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN: REAL, INTENT (IN) for vscosh DOUBLE PRECISION, INTENT (IN) for vdcosh COMPLEX, INTENT (IN) for vccosh	FORTRAN: Array, specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Name	Type	Description
	DOUBLE COMPLEX, INTENT(IN) for vzcosh	
	C: const float* for vsCosh	
	const double* for vdCosh	
	const MKL_Complex8* for vcCosh	
	const MKL_Complex16* for vzCosh	

Table 9-10 Precision Overflow Thresholds for Cosh Real Function

Data Type	Threshold Limitations on Input Parameters
single precision	$-\ln(\text{FLT\_MAX}) - \ln 2 < a[i] < \ln(\text{FLT\_MAX}) + \ln 2$
double precision	$-\ln(\text{DBL\_MAX}) - \ln 2 < a[i] < \ln(\text{DBL\_MAX}) + \ln 2$



NOTE. Overflow can occur also in Cosh complex function, but the exact formula is beyond the scope of this document.

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN: REAL for vscosh DOUBLE PRECISION for vdcosh COMPLEX for vccosh DOUBLE COMPLEX for vzcosh C: float* for vsCosh double* for vdCosh MKL_Complex8* for vcCosh MKL_Complex16* for vzCosh	FORTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Sinh

Computes hyperbolic sine of vector elements.

Syntax

Fortran:

```
call vssinh( n, a, y )
call vdsinh( n, a, y )
call vcsinh( n, a, y )
call vzsinh( n, a, y )
```

C:

```
vsSinh( n, a, y );
vdSinh( n, a, y );
vcSinh( n, a, y );
vzSinh( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN: INTEGER, INTENT(IN) C: int	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN: REAL, INTENT(IN) for vssinh DOUBLE PRECISION, INTENT(IN) for vdsinh COMPLEX, INTENT(IN) for vcsinh DOUBLE COMPLEX, INTENT(IN) for vzsinh C: const float* for vsSinh const double* for vdSinh	FORTRAN: Array, specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Name	Type	Description
	const MKL_Complex8* for vcSinh	
	const MKL_Complex16* for vzSinh	

Table 9-11 Precision Overflow Thresholds for Sinh Real Function

Data Type	Threshold Limitations on Input Parameters
single precision	$-\ln(\text{FLT\_MAX}) - \ln 2 < a[i] < \ln(\text{FLT\_MAX}) + \ln 2$
double precision	$-\ln(\text{DBL\_MAX}) - \ln 2 < a[i] < \ln(\text{DBL\_MAX}) + \ln 2$



NOTE. Overflow can occur also in `Sinh` complex function, but the exact formula is beyond the scope of this document.

Output Parameters

Name	Type	Description
y	FORTTRAN: REAL for vssinh DOUBLE PRECISION for vdsinh COMPLEX for vcsinh DOUBLE COMPLEX for vzsinh C: float* for vsSinh double* for vdSinh MKL_Complex8* for vcSinh MKL_Complex16* for vzSinh	FORTTRAN: Array, specifies the output vector y . C: Pointer to an array that contains the output vector y .

Tanh

Computes hyperbolic tangent of vector elements.

Syntax

Fortran:

```
call vstanh( n, a, y )
call vdtanh( n, a, y )
call vctanh( n, a, y )
call vztanh( n, a, y )
```

C:

```
vsTanh( n, a, y );
vdTanh( n, a, y );
vcTanh( n, a, y );
vzTanh( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN: REAL, INTENT (IN) for vstanh DOUBLE PRECISION, INTENT (IN) for vdtanh COMPLEX, INTENT (IN) for vctanh DOUBLE COMPLEX, INTENT (IN) for vztanh C: const float* for vsTanh const double* for vdTanh	FORTRAN: Array, specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Name	Type	Description
	const MKL_Complex8* for vcTanh	
	const MKL_Complex16* for vzTanh	

Output Parameters

Name	Type	Description
<i>y</i>	FORTTRAN: REAL for vstanh DOUBLE PRECISION for vdtanh COMPLEX for cstanh DOUBLE COMPLEX for zdtanh C: float* for vsTanh double* for vdTanh MKL_Complex8* for vcTanh MKL_Complex16* for vzTanh	FORTTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Acosh

Computes inverse hyperbolic cosine (nonnegative) of vector elements.

Syntax

Fortran:

```
call vsacosh( n, a, y )  
call vdacosh( n, a, y )  
call vcacosh( n, a, y )  
call vzacosh( n, a, y )
```

C:

`vsAcosh( n, a, y );`

`vdAcosh( n, a, y );`

`vcAcosh( n, a, y );`

`vzAcosh( n, a, y );`

Input Parameters

Name	Type	Description
<i>n</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.
<i>a</i>	FORTTRAN: REAL, INTENT (IN) for <code>vsacosh</code> DOUBLE PRECISION, INTENT (IN) for <code>vdacosh</code> COMPLEX, INTENT (IN) for <code>vcacosh</code> DOUBLE COMPLEX, INTENT (IN) for <code>vzacosh</code> C: const float* for <code>vsAcosh</code> const double* for <code>vdAcosh</code> const MKL_Complex8* for <code>vcAcosh</code> const MKL_Complex16* for <code>vzAcosh</code>	FORTTRAN: Array, specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Output Parameters

Name	Type	Description
<i>y</i>	FORTTRAN: REAL for <code>vsacosh</code> DOUBLE PRECISION for <code>vdacosh</code>	FORTTRAN: Array, specifies the output vector <i>y</i> .

Name	Type	Description
	COMPLEX for <code>vcacosh</code>	C: Pointer to an array that contains the output vector <i>y</i> .
	DOUBLE COMPLEX for <code>vzacosh</code>	
	C: float* for <code>vsAcosh</code>	
	double* for <code>vdAcosh</code>	
	MKL_Complex8* for <code>vcAcosh</code>	
	MKL_Complex16* for <code>vzAcosh</code>	

Asinh

Computes inverse hyperbolic sine of vector elements.

Syntax

Fortran:

```
call vsasinh( n, a, y )
call vdasinh( n, a, y )
call vcasinh( n, a, y )
call vzasinh( n, a, y )
```

C:

```
vsAsinh( n, a, y );
vdAsinh( n, a, y );
vcAsinh( n, a, y );
vzAsinh( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTTRAN: INTEGER, INTENT(IN) C: int	Specifies the number of elements to be calculated.

Name	Type	Description
<i>a</i>	FORTTRAN: REAL, INTENT (IN) for vsasinh DOUBLE PRECISION, INTENT (IN) for vdasinh COMPLEX, INTENT (IN) for vcasinh DOUBLE COMPLEX, INTENT (IN) for vzasinh C: const float* for vsAsinh const double* for vdAsinh const MKL_Complex8* for vcAsinh const MKL_Complex16* for vzAsinh	FORTTRAN: Array, specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Output Parameters

Name	Type	Description
<i>y</i>	FORTTRAN: REAL for vsasinh DOUBLE PRECISION for vdasinh COMPLEX for vcasinh DOUBLE COMPLEX for vzasinh C: float* for vsAsinh double* for vdAsinh MKL_Complex8* for vcAsinh MKL_Complex16* for vzAsinh	FORTTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Atanh

Computes inverse hyperbolic tangent of vector elements.

Syntax

Fortran:

```
call vsatanh( n, a, y )
call vdatanh( n, a, y )
call vcatanh( n, a, y )
call vzatanh( n, a, y )
```

C:

```
vsAtanh( n, a, y );
vdAtanh( n, a, y );
vcAtanh( n, a, y );
vzAtanh( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN: REAL, INTENT (IN) for vsatanh DOUBLE PRECISION, INTENT (IN) for vdatanh COMPLEX, INTENT (IN) for vcatanh DOUBLE COMPLEX, INTENT (IN) for vzatanh C: const float* for vsAtanh	FORTRAN: Array, specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Name	Type	Description
	const double* for vdAtanh	
	const MKL_Complex8* for vcAtanh	
	const MKL_Complex16* for vzAtanh	

Output Parameters

Name	Type	Description
<i>y</i>	FORTTRAN: REAL for vsatanh DOUBLE PRECISION for vdatanh COMPLEX for vcatanh DOUBLE COMPLEX for vzatanh C: float* for vsAtanh double* for vdAtanh MKL_Complex8* for vcAtanh MKL_Complex16* for vzAtanh	FORTTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Special Functions

Erf

Computes the error function value of vector elements.

Syntax

Fortran:

```
call vserf( n, a, y )
call vderf( n, a, y )
```


C:

```
vsErf( n, a, y );
```

```
vdErf( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN: REAL, INTENT (IN) for vserf DOUBLE PRECISION, INTENT (IN) for vderf C: const float* for vsErf const double* for vdErf	FORTRAN: Array, specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN: REAL for vserf DOUBLE PRECISION for vderf C: float* for vsErf double* for vdErf	FORTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Description

The function `Erf` computes the error function values for elements of the input vector *a* and writes them to the output vector *y*.

The error function is defined as given by:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt .$$

Useful relations:

1. $\text{erfc}(x) = 1 - \text{erf}(x) ,$

where erfc is the complementary error function.

2. $\Phi(x) = \frac{1}{2} \text{erf} (x/\sqrt{2}) ,$

where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \exp (- t^2/2) dt$$

is the cumulative normal distribution function.

3. $\Phi^{-1}(x) = \sqrt{2} \text{erf}^{-1}(2x - 1) ,$

where $\Phi^{-1}(x)$ and $\text{erf}^{-1}(x)$ are the inverses to $\Phi(x)$ and $\text{erf}(x)$ respectively.

Erfc

Computes the complementary error function value of vector elements.

Syntax

Fortran:

```
call vserfc( n, a, y )
```

```
call vderfc( n, a, y )
```

C:

```
vsErfc( n, a, y );
```

```
vdErfc( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.
<i>a</i>	FORTTRAN: REAL, INTENT (IN) for vserfc DOUBLE PRECISION, INTENT (IN) for vderfc C: const float* for vsErfc const double* for vdErfc	FORTTRAN: Array, specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Output Parameters

Name	Type	Description
<i>y</i>	FORTTRAN: REAL for vserfc DOUBLE PRECISION for vderfc C: float* for vsErfc double* for vdErfc	FORTTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Description

The function `Erfc` computes the complementary error function values for elements of the input vector *a* and writes them to the output vector *y*.

The complementary error function is defined as given by:

$$\operatorname{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt .$$

Useful relations:

1. $\operatorname{erfc}(x) = 1 - \operatorname{erf}(x)$.

2. $\Phi(x) = \frac{1}{2} \operatorname{erf}(x/\sqrt{2})$,

where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \exp(-t^2/2) dt$$

is the cumulative normal distribution function.

3. $\Phi^{-1}(x) = \sqrt{2} \operatorname{erf}^{-1}(2x - 1)$,

where $\Phi^{-1}(x)$ and $\operatorname{erf}^{-1}(x)$ are the inverses to $\Phi(x)$ and $\operatorname{erf}(x)$ respectively.

ErfInv

Computes inverse error function value of vector elements.

Syntax

Fortran:

```
call vserfinv( n, a, y )
call vderfinv( n, a, y )
```

C:

```
vsErfInv( n, a, y );
vdErfInv( n, a, y );
```

Input Parameters

Name	Type	Description
n	FORTRAN: INTEGER, INTENT(IN) C: int	Specifies the number of elements to be calculated.
a	FORTRAN: REAL, INTENT(IN) for vserfinv DOUBLE PRECISION, INTENT(IN) for vderfinv C: const float* for vsErfInv const double* for vdErfInv	FORTRAN: Array, specifies the input vector a . C: Pointer to an array that contains the input vector a .

Output Parameters

Name	Type	Description
y	FORTRAN: REAL for vserfinv DOUBLE PRECISION for vderfinv C: float* for vsErfInv double* for vdErfInv	FORTRAN: Array, specifies the output vector y . C: Pointer to an array that contains the output vector y .

Description

The function `ErfInv` computes the inverse error function values for elements of the input vector a and writes them to the output vector y

$$y = \operatorname{erf}^{-1}(a),$$

where $\operatorname{erf}(x)$ is the error function defined as given by:

$$\operatorname{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt .$$

Useful relations:

$$1. \operatorname{erf}^{-1}(x) = \operatorname{erfc}^{-1}(1 - x),$$

where `erfc` is the complementary error function.

$$2. \Phi(x) = \frac{1}{2} \operatorname{erf}(x/\sqrt{2}),$$

where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \exp(-t^2/2) dt$$

is the cumulative normal distribution function.

$$3. \Phi^{-1}(x) = \sqrt{2} \operatorname{erf}^{-1}(2x - 1),$$

where $\Phi^{-1}(x)$ and $\operatorname{erf}^{-1}(x)$ are the inverses to $\Phi(x)$ and $\operatorname{erf}(x)$ respectively.

Rounding Functions

Floor

Computes a rounded towards minus infinity integer value for each vector element.

Syntax

Fortran:

```
call vsfloor( n, a, y )
```

```
call vdfloor( n, a, y )
```

C:

```
vsFloor( n, a, y );
```

```
vdFloor( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTTRAN: INTEGER, INTENT(IN) C: int	Specifies the number of elements to be calculated.
<i>a</i>	FORTTRAN: REAL, INTENT(IN) for vsfloor DOUBLE PRECISION, INTENT(IN) for vdfloor C: const float* for vsFloor const double* for vdFloor	FORTTRAN: Array, specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Output Parameters

Name	Type	Description
<i>y</i>	FORTTRAN: REAL for vsfloor DOUBLE PRECISION for vdfloor C: float* for vsFloor double* for vdFloor	FORTTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Ceil

Computes a rounded towards plus infinity integer value for each vector element.

Syntax

Fortran:

```
call vsceil( n, a, y )  
call vdceil( n, a, y )
```

C:

```
vsCeil( n, a, y );
```

```
vdCeil( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN: REAL, INTENT (IN) for vsceil DOUBLE PRECISION, INTENT (IN) for vdceil C: const float* for vsCeil const double* for vdCeil	FORTRAN: Array, specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN: REAL for vsceil DOUBLE PRECISION for vdceil C: float* for vsCeil double* for vdCeil	FORTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Trunc

Computes a rounded towards zero integer value for each vector element.

Syntax

Fortran:

```
call vstrunc( n, a, y )
call vdtrunc( n, a, y )
```

C:

```
vsTrunc( n, a, y );
vdTrunc( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN: INTEGER, INTENT(IN) C: int	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN: REAL, INTENT(IN) for vstrunc DOUBLE PRECISION, INTENT(IN) for vdtrunc C: const float* for vsTrunc const double* for vdTrunc	FORTRAN: Array, specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN: REAL for vstrunc DOUBLE PRECISION for vdtrunc C: float* for vsTrunc	FORTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Name	Type	Description
	double* for vdTrunc	

Round

Computes a rounded to nearest integer value for each vector element.

Syntax

Fortran:

```
call vsround( n, a, y )
call vdround( n, a, y )
```

C:

```
vsRound( n, a, y );
vdRound( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN: REAL, INTENT (IN) for vsround DOUBLE PRECISION, INTENT (IN) for vdround C: const float* for vsRound const double* for vdRound	FORTRAN: Array, specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN: REAL for vsround DOUBLE PRECISION for vdround C: float* for vsRound double* for vdRound	FORTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Description

Halfway values, that is, 0.5, -1.5, and the like, are rounded off away from zero. That is, 0.5 -> 1, -1.5 -> -2, etc.

NearbyInt

Computes a rounded integer value in a current rounding mode for each vector element.

Syntax

Fortran:

```
call vsnearbyint( n, a, y )
```

```
call vdnearbyint( n, a, y )
```

C:

```
vsNearbyInt( n, a, y );
```

```
vdNearbyInt( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN: INTEGER, INTENT(IN) C: int	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN: REAL, INTENT(IN) for vsnearbyint	FORTRAN: Array, specifies the input vector <i>a</i> .

Name	Type	Description
	DOUBLE PRECISION, INTENT(IN) for vdnearbyint	C: Pointer to an array that contains the input vector <i>a</i> .
	C: const float* for vsNearbyInt	
	const double* for vdNearbyInt	

Output Parameters

Name	Type	Description
<i>y</i>	FORTTRAN: REAL for vsnearbyint	FORTTRAN: Array, specifies the output vector <i>y</i> .
	DOUBLE PRECISION for vdnearbyint	C: Pointer to an array that contains the output vector <i>y</i> .
	C: float* for vsNearbyInt	
	double* for vdNearbyInt	

Description

Halfway values, that is, 0.5, -1.5, and the like, are rounded off towards even values.

Rint

Computes a rounded integer value in a current rounding mode for each vector element with inexact result exception raised for each changed value.

Syntax

Fortran:

```
call vsrint( n, a, y )
call vdrint( n, a, y )
```

C:

```
vsRint( n, a, y );  
vdRint( n, a, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN: REAL, INTENT (IN) for vsrint DOUBLE PRECISION, INTENT (IN) for vdrint C: const float* for vsRint const double* for vdRint	FORTRAN: Array, specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN: REAL for vsrint DOUBLE PRECISION for vdrint C: float* for vsRint double* for vdRint	FORTRAN: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Description

Halfway values, that is, 0.5, -1.5, and the like, are rounded off towards even values. For each changed value, inexact result exception is raised.

Modf

Computes a truncated integer value and remaining fraction part for each vector element.

Syntax

Fortran:

```
call vsmodf( n, a, y, z )
call vdmodf( n, a, y, z )
```

C:

```
vsModf( n, a, y, z );
vdModf( n, a, y, z );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN: REAL, INTENT (IN) for vsmodf DOUBLE PRECISION, INTENT (IN) for vdmodf C: const float* for vsModf const double* for vdModf	FORTRAN: Array, specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Output Parameters

Name	Type	Description
<i>y, z</i>	FORTRAN: REAL for vsmodf DOUBLE PRECISION for vdmodf C: float* for vsModf	FORTRAN: Array, specifies the output vector <i>y</i> and <i>z</i> . C: Pointer to an array that contains the output vector <i>y</i> and <i>z</i> .

Name	Type	Description
	double* for vdModf	

VML Pack/Unpack Functions

This section describes VML functions which convert vectors with unit increment to and from vectors with positive increment indexing, vector indexing and mask indexing (see [Appendix B](#) for details on vector indexing methods).

[Table 9-12](#) lists available VML Pack/Unpack functions, together with data types and indexing methods associated with them.

Table 9-12 VML Pack/Unpack Functions

Function	Short Name	Data Types	Indexing Methods	Description
Pack		s, d	I, V, M	Gathers elements of arrays, indexed by different methods.
Unpack		s, d	I, V, M	Scatters vector elements to arrays with different indexing.

Pack

Copies elements of an array with specified indexing to a vector with unit increment.

Syntax

Fortran:

```
call vsPackI( n, a, inca, y )
call vsPackV( n, a, ia, y )
call vsPackM( n, a, ma, y )
call vdPackI( n, a, inca, y )
call vdPackV( n, a, ia, y )
call vdPackM( n, a, ma, y )
```

C:

```
vsPackI( n, a, inca, y );
vsPackV( n, a, ia, y );
vsPackM( n, a, ma, y );
vdPackI( n, a, inca, y );
vdPackV( n, a, ia, y );
vdPackM( n, a, ma, y );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTTRAN: INTEGER, INTENT (IN). C: int.	Specifies the number of elements to be calculated.
<i>a</i>	FORTTRAN: REAL, INTENT (IN) for vspacki, vspackv, vspackm DOUBLE PRECISION, INTENT (IN) for vdpacki, vdpackv, vdpackm. C: const float* for vsPackI, vsPackV, vsPackM const double* for vdPackI, vdPackV, vdPackM	FORTTRAN: Array, DIMENSION at least (1 + (n-1)*inca) for vspacki/vdpacki, Array, DIMENSION at least max (n,max(ia[j])), j=0, ..., n-1 for vspackv/vdpackv , Array, DIMENSION at least n for vspackm/vdpackm. Specifies the input vector <i>a</i> . C: Specifies pointer to an array that contains the input vector <i>a</i> . Size of the array must be: at least (1 + (n-1)*inca) for vsPackI/vdPackI, at least max(n,max(ia[j])), j=0, ..., n-1 for vsPackV/vdPackV , at least n for vsPackM/vdPackM.

Name	Type	Description
<i>inca</i>	FORTTRAN: INTEGER, INTENT (IN) for vspacki, vdpacki. C: int for vsPackI, vdPackI.	Specifies the increment for the elements of <i>a</i> .
<i>ia</i>	FORTTRAN: INTEGER, INTENT (IN) for vspackv, vdpackv. C: const int* for vsPackV, vdPackV.	FORTTRAN: Array, DIMENSION at least <i>n</i> . Specifies the index vector for the elements of <i>a</i> . C: Specifies the pointer to an array of size at least <i>n</i> that contains the index vector for the elements of <i>a</i> .
<i>ma</i>	FORTTRAN: INTEGER, INTENT (IN) for vspackm, vdpackm. C: const int* for vsPackM, vdPackM.	FORTTRAN: Array, DIMENSION at least <i>n</i> , Specifies the mask vector for the elements of <i>a</i> . C: Specifies the pointer to an array of size at least <i>n</i> that contains the mask vector for the elements of <i>a</i> .

Output Parameters

Name	Type	Description
<i>y</i>	FORTTRAN: REAL for vspacki, vspackv, vspackm DOUBLE PRECISION for vdpacki, vdpackv, vdpackm. C: float* for vsPackI, vsPackV, vsPackM double* for vdPackI, vdPackV, vdPackM	FORTTRAN: Array, DIMENSION at least <i>n</i> . Specifies the output vector <i>y</i> . C: Pointer to an array of size at least <i>n</i> that contains the output vector <i>y</i> .

Unpack

Copies elements of a vector with unit increment to an array with specified indexing.

Syntax

Fortran:

```
call vsunpacki( n, a, y, incy )
call vsunpackv( n, a, y, iy )
call vsunpackm( n, a, y, my )
call vdunpacki( n, a, y, incy )
call vdunpackv( n, a, y, iy )
call vdunpackm( n, a, y, my )
```

C:

```
vsUnpackI( n, a, y, incy );
vsUnpackV( n, a, y, iy );
vsUnpackM( n, a, y, my );
vdUnpackI( n, a, y, incy );
vdUnpackV( n, a, y, iy );
vdUnpackM( n, a, y, my );
```

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN: INTEGER, INTENT (IN). C: int.	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN: REAL, INTENT (IN) for vsunpacki, vsunpackv, vsunpackm	FORTRAN: Array, DIMENSION at least <i>n</i> . Specifies the input vector <i>a</i> .

Name	Type	Description
	DOUBLE PRECISION, INTENT(IN) for vdunpacki, vdunpackv, vdunpackm. C: const float* for vsUnpackI, vsUnpackV, vsUnpackM const double* for vdUnpackI, vdUnpackV, vdUnpackM	C: Specifies the pointer to an array of size at least n that contains the input vector a .
<i>incy</i>	FORTRAN: INTEGER, INTENT(IN) for vsunpacki, vdunpacki. C: int for vsUnpackI, vdUnpackI.	Specifies the increment for the elements of y .
<i>iy</i>	FORTRAN: INTEGER, INTENT(IN) for vsunpackv, vdunpackv. C: const int* for vsUnpackV, vdUnpackV.	FORTRAN: Array, DIMENSION at least n . Specifies the index vector for the elements of y . C: Specifies the pointer to an array of size at least n that contains the index vector for the elements of a .
<i>my</i>	FORTRAN: INTEGER, INTENT(IN) for vsunpackm, vdunpackm. C: const int* for vsUnpackM, vdUnpackM.	FORTRAN: Array, DIMENSION at least n , Specifies the mask vector for the elements of y . C: Specifies the pointer to an array of size at least n that contains the mask vector for the elements of a .

Output Parameters

Name	Type	Description
y	FORTRAN: REAL for vsunpacki, vsunpackv, vsunpackm	FORTRAN: Array, DIMENSION at least $(1 + (n-1)*incy)$ for vsunpacki,

Name	Type	Description
	DOUBLE PRECISION for vdunpacki, vdunpackv, vdunpackm.	at least $\max(n, \max(iy[j]))$, $j=0, \dots, n-1$, for vsunpackv, at least n for vsunpackm/vdunpackm
	C: float* for vsUnpackI, vsUnpackV, vsUnpackM double* for vdUnpackI, vdUnpackV, vdUnpackM	C: Specifies the pointer to an array that contains the output vector y . Size of the array must be: at least $(1 + (n-1)*incy)$ for vsUnPackI, at least $\max(n, \max(ia[j]))$, $j=0, \dots, n-1$, for vsUnPackV, at least n for vsUnPackM.

VML Service Functions

VML Service functions allow the user to set /get the accuracy mode, and set/get the error code. All these functions are available both in Fortran- and C- interfaces. [Table 9-13](#) lists available VML Service functions and their short description.

Table 9-13 VML Service Functions

Function Short Name	Description
SetMode	Sets the VML mode
GetMode	Gets the VML mode
SetErrStatus	Sets the VML error status
GetErrStatus	Gets the VML error status
ClearErrStatus	Clears the VML error status
SetErrorCallBack	Sets the additional error handler callback function
GetErrorCallBack	Gets the additional error handler callback function
ClearErrorCallBack	Deletes the additional error handler callback function

SetMode

Sets a new mode for VML functions according to *mode* parameter and stores the previous VML mode to *oldmode*.

Syntax

Fortran:

```
oldmode = vmlsetmode( mode )
```

C:

```
oldmode = vmlSetMode( mode );
```

Input Parameters

Name	Type	Description
<i>mode</i>	FORTTRAN: INTEGER, INTENT(IN) C: int	Specifies the VML mode to be set.

Output Parameters

Name	Type	Description
<i>oldmode</i>	FORTTRAN: INTEGER C: int	Specifies the former VML mode.

Description

The *mode* parameter is designed to control accuracy, FPU, error handling and threading options. [Table 9-14](#) lists values of the *mode* parameter. All other possible values of the *mode* parameter may be obtained from these values by using bitwise OR (|) operation to combine one value for accuracy, one for FPU, and one for error control options. The default value of the *mode* parameter is `VML_HA | VML_ERRMODE_DEFAULT | VML_NUM_THREADES_OMP_AUTO`. Thus, the current FPU control word (FPU precision and the rounding method) is used by default.

If any VML mathematical function requires different FPU precision, or rounding method, it changes these options automatically and then restores the former values. The *mode* parameter enables you to minimize switching the internal FPU mode inside each VML mathematical function

that works with similar precision and accuracy settings. To accomplish this, set the *mode* parameter to `VML_FLOAT_CONSISTENT` for single precision real and complex functions, or to `VML_DOUBLE_CONSISTENT` for double precision real and complex functions. These values of the *mode* parameter are the optimal choice for the respective function groups, as they are required for most of the VML mathematical functions. After the execution is over, set the *mode* to `VML_RESTORE` if you need to restore the previous FPU mode.

Table 9-14 Values of the *mode* Parameter

Value of <i>mode</i>	Description
Accuracy Control	
<code>VML_HA</code>	High accuracy versions of VML functions will be used.
<code>VML_LA</code>	Low accuracy versions of VML functions will be used.
<code>VML_EP</code>	Enhanced Performance accuracy versions of VML functions will be used.
Additional FPU Mode Control	
<code>VML_FLOAT_CONSISTENT</code>	The optimal FPU mode (control word) for single precision functions is set, and the previous FPU mode is saved.
<code>VML_DOUBLE_CONSISTENT</code>	The optimal FPU mode (control word) for double precision functions is set, and the previous FPU mode is saved.
<code>VML_RESTORE</code>	The previously saved FPU mode is restored.
Error Mode Control	
<code>VML_ERRMODE_IGNORE</code>	No action is set for computation errors.
<code>VML_ERRMODE_ERRNO</code>	On error, the <i>errno</i> variable is set.
<code>VML_ERRMODE_STDERR</code>	On error, the error text information is written to <i>stderr</i> .
<code>VML_ERRMODE_EXCEPT</code>	On error, an exception is raised.
<code>VML_ERRMODE_CALLBACK</code>	On error, an additional error handler function is called.
<code>VML_ERRMODE_DEFAULT</code>	On error, the <i>errno</i> variable is set, an exception is raised, and an additional error handler function is called.
Treading Mode Control	
<code>VML_NUM_THREADS_OMP_AUTO</code>	This is default behavior. Maximum number of threads is determined by environmental variable <code>OMP_NUM_THREADS</code> and can be overridden by OpenMP* function <code>omp_set_num_threads()</code> . For performance reasons VML threading logic can use fewer number of threads.

Value of <i>mode</i>	Description
VML_NUM_THREADS_OMP_FIXED	Number of threads is determined by environmental variable OMP_NUM_THREADS and can be overridden by OpenMP* function <code>omp_set_num_threads()</code> . Use this mode to disable VML threading logic.

Examples

Several examples of calling the function `vmlSetMode()` with different values of the *mode* parameter are given below:

Fortran:

```
oldmode = vmlsetmode( VML_LA )  
  
call vmlsetmode( IOR(VML_LA, IOR(VML_FLOAT_CONSISTENT,  
VML_ERRMODE_IGNORE )))  
  
call vmlsetmode( VML_RESTORE)  
  
call vmlsetmode( VML_NUM_THREADS_OMP_FIXED)
```

C:

```
vmlSetMode( VML_LA );  
  
vmlSetMode( VML_LA | VML_FLOAT_CONSISTENT | VML_ERRMODE_IGNORE );  
  
vmlSetMode( VML_RESTORE);  
  
vmlSetMode( VML_NUM_THREADS_OMP_FIXED);
```

GetMode

Gets the VML mode.

Syntax

Fortran:

```
mod = vmlgetmode()
```

C:

```
mod = vmlGetMode( void );
```

Output Parameters

Name	Type	Description
<i>mod</i>	FORTTRAN: INTEGER C: int	Specifies the packed <i>mode</i> parameter.

Description

The function `vmlGetMode()` returns the VML *mode* parameter that controls accuracy, FPU, and error handling options. The *mod* variable value is some combination of the values listed in the [Table 9-14](#). You can obtain some of these values using the respective mask from the [Table 9-15](#).

Table 9-15 Values of Mask for the *mode* Parameter

Value of mask	Description
VML_ACCURACY_MASK	Specifies mask for accuracy <i>mode</i> selection.
VML_FPUMODE_MASK	Specifies mask for FPU <i>mode</i> selection.
VML_ERRMODE_MASK	Specifies mask for error <i>mode</i> selection.

See example below:

Examples

Fortran:

```
mod = vmlgetmode()
accm = IAND(mod, VML_ACCURACY_MASK)
fpum = IAND(mod, VML_FPUMODE_MASK)
errm = IAND(mod, VML_ERRMODE_MASK)
```

C:

```
accm = vmlGetMode(void ) & VML_ACCURACY_MASK;
fpum = vmlGetMode(void ) & VML_FPUMODE_MASK;
errm = vmlGetMode(void ) & VML_ERRMODE_MASK;
```


SetErrStatus

Sets the new VML error status according to *err* and stores the previous VML error status to *olderr*.

Syntax

Fortran:

```
olderr = vmlseterrstatus( err )
```

C:

```
olderr = vmlSetErrStatus( err );
```

Input Parameters

Name	Type	Description
<i>err</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Specifies the VML error status to be set.

Output Parameters

Name	Type	Description
<i>olderr</i>	FORTTRAN: INTEGER C: int	Specifies the former VML error status.

Description

Table 9-16 lists possible values of the *err* parameter.

Table 9-16 Values of the VML Error Status

Error Status	Description
VML_STATUS_OK	The execution was completed successfully.
VML_STATUS_BADSIZE	The array dimension is not positive.
VML_STATUS_BADMEM	NULL pointer is passed.
VML_STATUS_ERRDOM	At least one of array values is out of a range of definition.
VML_STATUS_SING	At least one of array values caused a singularity.

Error Status	Description
VML_STATUS_OVERFLOW	An overflow has happened during the calculation process.
VML_STATUS_UNDERFLOW	An underflow has happened during the calculation process.

Examples

```
vmlSetErrStatus( VML_STATUS_OK );
vmlSetErrStatus( VML_STATUS_ERRDOM );
vmlSetErrStatus( VML_STATUS_UNDERFLOW );
```

GetErrStatus

Gets the VML error status.

Syntax

Fortran:

```
err = vmlgeterrstatus( )
```

C:

```
err = vmlGetErrStatus( void );
```

Output Parameters

Name	Type	Description
<i>err</i>	FORTTRAN: INTEGER C: int	Specifies the VML error status.

ClearErrStatus

Sets the VML error status to `VML_STATUS_OK` and stores the previous VML error status to `olderr`.

Syntax

Fortran:

```
olderr = vmlclearerrstatus( )
```

C:

```
olderr = vmlClearErrStatus( void );
```

Output Parameters

Name	Type	Description
<code>olderr</code>	FORTTRAN: INTEGER C: int	Specifies the former VML error status.

SetErrorCallBack

Sets the additional error handler callback function and gets the old callback function.

Syntax

Fortran:

```
oldcallback = vmlseterrorcallback( callback )
```

C:

```
oldcallback = vmlSetErrorCallBack( callback );
```

Input Parameters

Name	Description
FORTTRAN: Address of the callback function. <i>callback</i>	The callback function has the following format: <pre> INTEGER FUNCTION ERRFUNC(par) TYPE (ERROR_STRUCTURE) par ! ... ! user error processing ! ... ERRFUNC = 0 ! if ERRFUNC= 0 - standard VML error handler ! is called after the callback ! if ERRFUNC != 0 - standard VML error handler ! is not called END </pre> The passed error structure is defined as follows: <pre> TYPE ERROR_STRUCTURE SEQUENCE INTEGER*4 ICODE INTEGER*4 IINDEX REAL*8 DBA1 REAL*8 DBA2 REAL*8 DBR1 REAL*8 DBR2 CHARACTER(64) CFUNCNAME INTEGER*4 IFUNCNAMELEN END TYPE ERROR_STRUCTURE </pre>

Name	Description
C: <i>callback</i> Pointer to the callback function.	The callback function has the following format: <pre data-bbox="867 392 1235 623">static int __stdcall MyHandler(DefVmlErrorContext* pContext) { /* Handler body */ };</pre> The passed error structure is defined as follows: <pre data-bbox="867 722 1360 1362">typedef struct _DefVmlErrorContext { int iCode; /* Error status value */ int iIndex; /* Index for bad array element, or bad array dimension, or bad array pointer */ double dbA1; /* Error argument 1 */ double dbA2; /* Error argument 2 */ double dbR1; /* Error result 1 */ double dbR2; /* Error result 2 */ char cFuncName[64]; /* Function name */ int iFuncNameLen; /* Length of functionname*/ } DefVmlErrorContext;</pre>

Output Parameters

Name	Type	Description
<code>oldcallback</code>	FORTRAN: INTEGER C: int	FORTRAN: Address of the former callback function. C: Pointer to the former callback function.

Description

The callback function is called on each VML mathematical function error if VML_ERRMODE_CALLBACK error mode is set (see [Table 9-14](#)).

Use the `vm1SetErrorCallBack()` function if you need to define your own callback function instead of default empty callback function.

The input structure for a callback function contains the following information about the error encountered:

- the input value that caused an error
- location (array index) of this value
- the computed result value
- error code
- name of the function in which the error occurred.

You can insert your own error processing into the callback function. This may include correcting the passed result values in order to pass them back and resume computation. The standard error handler is called after the callback function only if it returns 0.

GetErrorCallBack

Gets the additional error handler callback function.

Syntax

Fortran:

```
callback = vm1geterrorcallback( )
```

C:

```
callback = vm1GetErrorCallBack( void );
```

Output Parameters

Name

callback

Description

FORTRAN: Address of the callback function

C: Pointer to the callback function

ClearErrorCallBack

Deletes the additional error handler callback function and retrieves the former callback function.

Syntax

Fortran:

```
oldcallback = vmlclearerrorcallback( )
```

C:

```
oldcallback = vmlClearErrorCallBack( void );
```

Output Parameters

Name

Type

oldcallback **FORTRAN:** INTEGER

C: int

Description

FORTRAN: Address of the former callback function

C: Pointer to the former callback function

Statistical Functions

Statistical functions in Intel® MKL are known as Vector Statistical Library (VSL) that is designed for the purpose of

- generating vectors of pseudorandom and quasi-random numbers
- performing mathematical operations of convolution and correlation.

The corresponding functionality is described in the respective [Random Number Generators](#) and [Convolution and Correlation](#) sections.

Random Number Generators

VSL provides a set of routines implementing commonly used pseudo- or quasi-random number generators with continuous and discrete distribution. To speed up performance, all these routines were developed using the calls to the highly optimized *Basic Random Number Generators* (BRNGs) and the library of vector mathematical functions (VML, see [Chapter 9, “Vector Mathematical Functions”](#)).

VSL provides interfaces both for FORTRAN and C languages. For users of the C and C++ languages the `mkl_vsl.h` header file is provided. For users of the Fortran 90 or Fortran 95 language the `mkl_vsl.fi` header file is provided. Both header files are found in the following directory:

```
${MKL}/include
```

The `mkl_vsl.fi` header is intended for using via the FORTRAN `include` clause and is compatible with both standard forms of F90/F95 sources — the free and 72-columns fixed forms. If you need to use the VSL interface with 80- or 132-columns fixed form sources, you may add a new file to your project. That file is formatted as a 72-columns fixed-form source and consists of a single `include` clause as follows:

```
include 'mkl_vsl.fi'
```

This `include` clause causes the compiler to generate the module files `mkl_vsl.mod` and `mkl_vsl_type.mod`, which are used to process the FORTRAN use clauses referencing to the VSL interface:

```
use mkl_vsl_type
use mkl_vsl
```

Because of this specific feature, you do not need to include the `mkl_vsl.fi` header into each source of your project. You only need to include the header into some of the sources. In any case, make sure that the sources that depend on the VSL interface are compiled after those that include the header so that the module files `mkl_vsl.mod` and `mkl_vsl_type.mod` are generated prior to using them.



NOTE. For FORTRAN interface, VSL provides both subroutine-style interface and function-style interface. Default interface in this case is a function-style interface. Subroutine-style interface is provided for backward compatibility only. To use subroutine-style interface, manually include `mkl_vsl_subroutine.fi` file instead of `mkl_vsl.fi` by changing the line `include 'mkl_vsl.fi'` in `include\mkl.fi` with the line `include 'mkl_vsl_subroutine.fi'`.

Function-style interface, unlike subroutine-style interface, allows user to get error status of each routine.

All VSL routines can be classified into three major categories:

- Transformation routines for different types of statistical distributions, for example, uniform, normal (Gaussian), binomial, etc. These routines indirectly call basic random number generators, which are either pseudorandom number generators or quasi-random number generators. Detailed description of the generators can be found in [Distribution Generators](#) section.
- Service routines to handle random number streams: create, initialize, delete, copy, save to a binary file, load from a binary file, get the index of a basic generator. The description of these routines can be found in [Service Subroutines](#) section.
- Registration routines for basic pseudorandom generators and routines that obtain properties of the registered generators (see [Advanced Service Subroutines](#) section).

The last two categories are referred to as service routines.

Conventions

This document makes no specific differentiation between random, pseudorandom, and quasi-random numbers, nor between random, pseudorandom, and quasi-random number generators unless the context requires otherwise. For details, refer to 'Random Numbers' section in [VSL Notes](#) document provided with Intel® MKL.

All generators of nonuniform distributions, both discrete and continuous, are built on the basis of the uniform distribution generators, called Basic Random Number Generators (BRNGs). The pseudorandom numbers with nonuniform distribution are obtained through an appropriate

transformation of the uniformly distributed pseudorandom numbers. Such transformations are referred to as *generation methods*. For a given distribution, several generation methods can be used. See [VSL Notes](#) for the description of methods available for each generator.

The *stream descriptor* specifies which BRNG should be used in a given transformation method. See 'Random Streams and RNGs in Parallel Computation' section of [VSL Notes](#).

The term *computational node* means a logical or physical unit that can process data in parallel.

Mathematical Notation

The following notation is used throughout the text:

N	The set of natural numbers $N = \{1, 2, 3 \dots\}$.
Z	The set of integers $Z = \{\dots -3, -2, -1, 0, 1, 2, 3 \dots\}$.
R	The set of real numbers.
$\lfloor a \rfloor$	The floor of a (the largest integer less than or equal to a).

$\oplus$ or **xor**

Bitwise exclusive OR.

$$C_{\alpha}^k \text{ or } \binom{\alpha}{k}$$

Binomial coefficient or combination ($\alpha \in R, \alpha \geq 0; k \in N \cup \{0\}$).

$$C_{\alpha}^k = 1$$

For $\alpha \geq k$ binomial coefficient is defined as

$$C_{\alpha}^k = \frac{\alpha(\alpha - 1) \dots (\alpha - k + 1)}{k!}$$

If $\alpha < k$, then

$$C_{\alpha}^k = 0$$

$\Phi(x)$

Cumulative Gaussian distribution function

$$\Phi(x) = \int_{-\infty}^x \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{y^2}{2}\right) dy$$

	defined over $-\infty < x < +\infty$.
	$\Phi(-\infty) = 0, \Phi(+\infty) = 1$.
$\Gamma(\alpha)$	The complete gamma function
	$\Gamma(\alpha) = \int_0^{\infty} t^{\alpha-1} e^{-t} dt$
	where $\alpha > 0$.
$B(p, q)$	The complete beta function
	$B(p, q) = \int_0^1 t^{p-1} (1-t)^{q-1} dt$
	where $p > 0$ and $q > 0$.
$LCG(a, c, m)$	Linear Congruential Generator $x_{n+1} = (ax_n + c) \bmod m$, where a is called the <i>multiplier</i> , c is called the <i>increment</i> , and m is called the <i>modulus</i> of the generator.
$MCG(a, m)$	Multiplicative Congruential Generator $x_{n+1} = (ax_n) \bmod m$ is a special case of Linear Congruential Generator, where the increment c is taken to be 0.
$GFSR(p, q)$	Generalized Feedback Shift Register Generator
	$x_n = x_{n-p} \oplus x_{n-q}$.

Naming Conventions

The names of all VSL functions in FORTRAN are lowercase; names in C may contain both lowercase and uppercase letters.

The names of generator routines have the following structure:

`v<type of result>rng<distribution>` for FORTRAN-interface

`v<type of result>Rng<distribution>` for C-interface,

where `v` is the prefix of a VSL vector function, and the field `<type of result>` is either `s`, `d`, or `i` and specifies one of the following types:

<code>s</code>	REAL for FORTRAN-interface float for C-interface
<code>d</code>	DOUBLE PRECISION for FORTRAN-interface double for C-interface
<code>i</code>	INTEGER for FORTRAN-interface int for C-interface

Prefixes `s` and `d` apply to continuous distributions only, prefix `i` applies only to discrete case. The prefix `rng` indicates that the routine is a random generator, and the `<distribution>` field specifies the type of statistical distribution.

Names of service routines follow the template below:

`vsl<name>`,

where `vsl` is the prefix of a VSL service function. The field `<name>` contains a short function name. For a more detailed description of service routines refer to [Service Routines](#) and [Advanced Service Routines](#) sections.

Prototype of each generator routine corresponding to a given probability distribution fits the following structure:

`<function name>( method, stream, n, r, [<distribution parameters >] ),`

where

- `method` is the number specifying the method of generation. A detailed description of this parameter can be found in [Distribution Generators](#) section. See the next page for `method` name structure definition.
- `stream` defines the random stream descriptor and must have a nonzero value. Random streams and their usage are discussed further in [Random Streams](#) and [Service Routines](#).
- `n` defines the number of random values to be generated. If `n` is less than or equal to zero, no values are generated. Furthermore, if `n` is negative, an error condition is set.
- `r` defines the destination array for the generated numbers. The dimension of the array must be large enough to store at least `n` random numbers.

Additional parameters included into `<distribution parameters>` field are individual for each generator routine and are described in detail in [Distribution Generators](#) section.

To invoke a distribution generator, use a call to the respective VSL routine. For example, to obtain a vector `r`, composed of `n` independent and identically distributed random numbers with normal (Gaussian) distribution, that have the mean value `a` and standard deviation `sigma`, write the following:

for FORTRAN-interface

```
status = vsrnggaussian( method, stream, n, r, a, sigma )
```

for C-interface

```
status = vsRngGaussian( method,
stream, n, r, a, sigma )
```

The name of a *method* parameter has the following structure:

```
VSL_METHOD_<precision><distribution>_<method>,
```

```
VSL_METHOD_<precision><distribution>_<method>_ACCURATE,
```

where

<code><precision></code>	<i>s</i>	for single precision continuous distribution
	<i>D</i>	for double precision continuous distribution
	<i>I</i>	for discrete distribution
<code><distribution></code>	probability distribution	
<code><method></code>	method name.	

Type of name structure for *method* parameter corresponds to fast and accurate modes of random number generation (see “Distribution Generators” section and [VSL Notes](#) for details).

Method names `VSL_METHOD_<precision><distribution>_<method>` and `VSL_METHOD_<precision><distribution>_<method>_ACCURATE` should be used with `vsl<precision>Rng<distribution>` function only, where

<code><precision></code>	<i>s</i>	for single precision continuous distribution
	<i>d</i>	for double precision continuous distribution
	<i>i</i>	for discrete distribution
<code><distribution></code>	probability distribution.	

[Table 10-1](#) provides specific predefined values of the *method* name. The third column contains names of the functions that use the given method.

Table 10-1 Values of `<method>` in *method* parameter

Method	Short Description	Functions
STD	Standard method. Currently there is only one method for these functions.	Uniform (continuous), Uniform (discrete), UniformBits

Method	Short Description	Functions
BOXMULLER	BOXMULLER generates normally distributed random number x thru the pair of uniformly distributed numbers u_1 and u_2 according to the formula: $x = \sqrt{-2 \ln u_1} \sin 2\pi u_2$	Gaussian , GaussianMV
BOXMULLER2	BOXMULLER2 generates normally distributed random numbers x_1 and x_2 thru the pair of uniformly distributed numbers u_1 and u_2 according to the formulas: $x_1 = \sqrt{-2 \ln u_1} \sin 2\pi u_2$ $x_2 = \sqrt{-2 \ln u_1} \cos 2\pi u_2$	Gaussian , GaussianMV , Lognormal
ICDF	Inverse cumulative distribution function method.	Exponential , Laplace , Weibull , Cauchy , Rayleigh , Gumbel , Bernoulli , Geometric , Gaussian , GaussianMV
GNORM	For $\alpha > 1$, a gamma distributed random number is generated as a cube of properly scaled normal random number; for $0.6 \leq \alpha < 1$, a gamma distributed random number is generated using rejection from Weibull distribution; for $\alpha < 0.6$, a gamma distributed random number is obtained using transformation of exponential power distribution; for $\alpha = 1$, gamma distribution is reduced to exponential distribution.	Gamma

Method	Short Description	Functions
CJA	For $\min(p, q) > 1$, Cheng method is used; for $\min(p, q) < 1$, Jöhnk method is used, if $q + K \cdot p^2 + C \leq 0$ ($K = 0.852...$, $C = -0.956...$) otherwise, Atkinson switching algorithm is used; for $\max(p, q) < 1$, method of Jöhnk is used; for $\min(p, q) < 1$, $\max(p, q) > 1$, Atkinson switching algorithm is used (CJA stands for the first letters of Cheng, Jöhnk, Atkinson); for $p = 1$ or $q = 1$, inverse cumulative distribution function method is used; for $p = 1$ and $q = 1$, beta distribution is reduced to uniform distribution.	Beta
BTPE	Acceptance/rejection method for $ntrial \cdot \min(p, 1 - p) \geq 30$ with decomposition into 4 regions: – 2 parallelograms – triangle – left exponential tail – right exponential tail	Binomial
H2PE	Acceptance/rejection method for large mode of distribution with decomposition into 3 regions: – rectangular – left exponential tail – right exponential tail	Hypergeometric
PTPE	Acceptance/rejection method for $\lambda \geq 27$ with decomposition into 4 regions: – 2 parallelograms – triangle – left exponential tail – right exponential tail; otherwise, table lookup method is used.	Poisson

Method	Short Description	Functions
POISNORM	for $\lambda \geq 1$, method based on Poisson inverse CDF approximation by Gaussian inverse CDF; for $\lambda < 1$, table lookup method is used.	<a>Poisson , <a>PoissonV
NBAR	Acceptance/rejection method for ,	<a>NegBinomial

$$\frac{(a-1) \cdot (1-p)}{p} \geq 100$$

- with decomposition into 5 regions:
- rectangular
 - 2 trapezoid
 - left exponential tail
 - right exponential tail

Basic Generators

VSL provides the following BRNGs, which differ in speed and other properties:

- the 32-bit multiplicative congruential pseudorandom number generator $MCG(1132489760, 2^{31}-1)$ [L'Ecuyer99]
- the 32-bit generalized feedback shift register pseudorandom number generator $GFSR(250,103)$ [Kirkpatrick81]
- the combined multiple recursive pseudorandom number generator $MRG-32k3a$ [L'Ecuyer99a]
- the 59-bit multiplicative congruential pseudorandom number generator $MCG(13^{13}, 2^{59})$ from NAG Numerical Libraries [NAG]
- Wichmann-Hill pseudorandom number generator (a set of 273 basic generators) from NAG Numerical Libraries [NAG]
- Mersenne Twister pseudorandom number generator $MT19937$ [Matsumoto98] with period length $2^{19937}-1$ of the produced sequence

- Set of 1024 Mersenne Twister pseudorandom number generators *MT2203* [Matsumoto98], [Matsumoto00]. Each of them generates a sequence of period length equal to $2^{2203}-1$. Parameters of the generators provide mutual independence of the corresponding sequences.

Besides these pseudorandom number generators, VSL provides two basic quasi-random number generators:

- Sobol quasi-number generator [Sobol76], [Bratley88], which works in arbitrary dimension. For dimensions greater than 40 the user should supply initialization parameters (initial direction numbers and primitive polynomials or direction numbers) by using `vslNewStreamEx` function. See additional details on interface for registration of the parameters in the library in [VSL Notes](#).
- Niederreiter quasi-random number generator [Bratley92], which works in arbitrary dimension. For dimensions greater than 318 the user should supply initialization parameters (irreducible polynomials or direction numbers) by using `vslNewStreamEx` function. See additional details on interface for registration of the parameters in the library in [VSL Notes](#).

Also see some testing results for the generators in [VSL Notes](#) and comparative performance data at http://www.intel.com/software/products/mkl/data/vsl/vsl_performance_data.htm.

VSL provides means of registration of such user-designed generators through the steps described in [Advanced Service Subroutines](#) section.

For some basic generators, VSL provides two methods of creating independent random streams in multiprocessor computations, which are the leapfrog method and the block-splitting method. These sequence splitting methods are also useful in sequential Monte Carlo.

In addition, MT2203 pseudorandom number generator is a set of 1024 generators designed to create up to 1024 independent random sequences, which might be used in parallel Monte Carlo simulations. Another generator that has the same feature is Wichmann-Hill. It allows creating up to 273 independent random streams. The properties of the generators designed for parallel computations are discussed in detail in [Coddington94].

You may want to design and use your own basic generators. VSL provides means of registration of such user-designed generators through the steps described in [Advanced Service Subroutines](#) section.

There is also an option to utilize externally generated random numbers in VSL distribution generator routines. For this purpose VSL provides three additional basic random number generators:

- for external random data packed in 32-bit integer array
- for external random data stored in double precision floating-point array; data is supposed to be uniformly distributed over (a,b) interval
- for external random data stored in single precision floating-point array; data is supposed to be uniformly distributed over (a,b) interval.

Such basic generators are called the abstract basic random number generators.
See [VSL Notes](#) for a more detailed description of the generator properties.

BRNG Parameter Definition

Predefined values for the *brng* input parameter are as follows:

Table 10-2 Values of *brng* parameter

Value	Short Description
VSL_BRNG_MCG31	A 31-bit multiplicative congruential generator.
VSL_BRNG_R250	A generalized feedback shift register generator.
VSL_BRNG_MRG32K3A	A combined multiple recursive generator with two components of order 3.
VSL_BRNG_MCG59	A 59-bit multiplicative congruential generator.
VSL_BRNG_WH	A set of 273 Wichmann-Hill combined multiplicative congruential generators.
VSL_BRNG_MT19937	A Mersenne Twister pseudorandom number generator.
VSL_BRNG_MT2203	A set of 1024 Mersenne Twister pseudorandom number generators.
VSL_BRNG_SOBOL	A 32-bit Gray code-based generator producing low-discrepancy sequences for dimensions $1 \leq s \leq 40$; user-defined dimensions are also available.
VSL_BRNG_NIEDERR	A 32-bit Gray code-based generator producing low-discrepancy sequences for dimensions $1 \leq s \leq 318$; user-defined dimensions are also available.
VSL_BRNG_IABSTRACT	An abstract random number generator for integer arrays.
VSL_BRNG_DABSTRACT	An abstract random number generator for double precision floating-point arrays.
VSL_BRNG_SABSTRACT	An abstract random number generator for single precision floating-point arrays.

See [VSL Notes](#) for detailed description.

Random Streams

Random stream (or *stream*) is an abstract source of pseudo- and quasi-random sequences of uniform distribution. Users have no direct access to these sequences and operate with stream state descriptors only. A stream state descriptor, which holds state descriptive information for a particular BRNG, is a necessary parameter in each routine of a distribution generator. Only the distribution generator routines operate with random streams directly. See [VSL Notes](#) for details.



NOTE. Random streams associated with abstract basic random number generator are called the abstract random streams. See [VSL Notes](#) for detailed description of abstract streams and their use.

User can create unlimited number of random streams by VSL [Service Routines](#) like [NewStream](#) and utilize them in any distribution generator to get the sequence of numbers of given probability distribution. When they are no longer needed, the streams should be deleted calling service routine [DeleteStream](#).

VSL provides service functions [SaveStreamF](#) and [LoadStreamF](#) to save random stream descriptive data to a binary file and to read this data from a binary file respectively. See [VSL Notes](#) for detailed description.

Data Types

FORTRAN:

```
TYPE VSL_STREAM_STATE
  INTEGER*4 descriptor1
  INTEGER*4 descriptor2
ENDTYPE VSL_STREAM_STATE
```

C:

```
typedef (void*) VSLStreamStatePtr;
```

See [Advanced Service Routines](#) for the format of the stream state structure for user-designed generators.

Error Reporting

VSL routines return status codes of the performed operation to report errors and warnings to the calling program. Thus, it is up to the application to perform error-related actions and/or recover from the error. The status codes are of integer type and have the following format:

VSL_ERROR_<ERROR_NAME> - indicates VSL errors

VSL_WARNING_<WARNING_NAME> - indicates VSL warnings.

VSL errors are of negative values while warnings are of positive values. The status code of zero value indicates that the operation is completed successfully: VSL_ERROR_OK (or synonymic VSL_STATUS_OK).

Table 10-3 Status Codes and Messages

Status Code	Message
VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_BAD_ARG	Input argument value is not valid.
VSL_ERROR_NULL_PTR	Input pointer argument is NULL.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory.
VSL_ERROR_INVALID_BRNG_INDEX	BRNG index is not valid.
VSL_ERROR_BRNGS_INCOMPATIBLE	Two BRNGs are not compatible for the operation.
VSL_ERROR_LEAPFROG_UNSUPPORTED	BRNG does not support Leapfrog method.
VSL_ERROR_SKIPAHEAD_UNSUPPORTED	BRNG does not support Skip-Ahead method.
VSL_ERROR_BAD_STREAM	The random stream is invalid.
VSL_ERROR_FILE_OPEN	Indicates an error in opening the file.
VSL_ERROR_FILE_READ	Indicates an error in reading the file.
VSL_ERROR_FILE_WRITE	Indicates an error in writing the file.
VSL_ERROR_FILE_CLOSE	Indicates an error in closing the file.
VSL_ERROR_BAD_FILE_FORMAT	File format is unknown.

Status Code	Message
VSL_ERROR_UNSUPPORTED_FILE_VER	File format version is not supported.
VSL_ERROR_BRNG_TABLE_FULL	Registration cannot be completed due to lack of free entries in the table of registered BRNGs.
VSL_ERROR_BAD_STREAM_STATE_SIZE	The value in <code>StreamStateSize</code> field is bad.
VSL_ERROR_BAD_WORD_SIZE	The value in <code>WordSize</code> field is bad.
VSL_ERROR_BAD_NSEEDS	The value in <code>NSeeds</code> field is bad.
VSL_ERROR_BAD_NBITS	The value in <code>NBits</code> field is bad.
VSL_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> n_{\max}$.
VSL_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns zero as the number of updated entries in a buffer.
VSL_ERROR_INVALID_ABSTRACT_STREAM	The abstract random stream is invalid.

Service Routines

Stream handling comprises routines for creating, deleting, or copying the streams and getting the index of a basic generator. A random stream can also be saved to and then read from a binary file. [Table 10-4](#) lists all available service routines

Table 10-4 Service Routines

Routine	Short Description
NewStream	Creates and initializes a random stream.
NewStreamEx	Creates and initializes a random stream for the generators with multiple initial conditions.
iNewAbstractStream	Creates and initializes an abstract random stream for integer arrays.
dNewAbstractStream	Creates and initializes an abstract random stream for double precision floating-point arrays.

Routine	Short Description
<code>sNewAbstractStream</code>	Creates and initializes an abstract random stream for single precision floating-point arrays.
<code>DeleteStream</code>	Deletes previously created stream.
<code>CopyStream</code>	Copies a stream to another stream.
<code>CopyStreamState</code>	Creates a copy of a random stream state.
<code>SaveStreamF</code>	Writes a stream to a binary file.
<code>LoadStreamF</code>	Reads a stream from a binary file.
<code>LeapfrogStream</code>	Initializes the stream by the leapfrog method to generate a subsequence of the original sequence.
<code>SkipAheadStream</code>	Initializes the stream by the skip-ahead method.
<code>GetStreamStateBrng</code>	Obtains the index of the basic generator responsible for the generation of a given random stream.
<code>GetNumRegBrngs</code>	Obtains the number of currently registered basic generators.



NOTE. In the above table, the `vs1` prefix in the function names is omitted. In the function reference this prefix is always used in function prototypes and code examples.

Most of the generator-based work comprises three basic steps:

1. Creating and initializing a stream (`NewStream`, `NewStreamEx`, `CopyStream`, `CopyStreamState`, `LeapfrogStream`, `SkipAheadStream`).
2. Generating random numbers with given distribution, see [Distribution Generators](#).
3. Deleting the stream (`DeleteStream`).

Note that you can concurrently create multiple streams and obtain random data from one or several generators by using the stream state. You must use the `DeleteStream` function to delete all the streams afterwards.

NewStream

Creates and initializes a random stream.

Syntax

Fortran:

```
status = vslnewstream( stream, brng, seed )
```

C:

```
status = vslNewStream( &stream, brng, seed );
```

Description

For a basic generator with number *brng*, this function creates a new stream and initializes it with a 32-bit seed. The seed is an initial value used to select a particular sequence generated by the basic generator *brng*. The function is also applicable for generators with multiple initial conditions. See [VSL Notes](#) for a more detailed description of stream initialization for different basic generators.



NOTE. This function is not applicable for abstract basic random number generators. Please use [vslNewAbstractStream](#), [vslsNewAbstractStream](#) or [vslldNewAbstractStream](#) to utilize integer, single-precision or double-precision external random data respectively.

Input Parameters

Name	Type	Description
<i>brng</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Index of the basic generator to initialize the stream. See Table 10-2 for specific value.
<i>seed</i>	FORTRAN: INTEGER, INTENT (IN) C: unsigned int	Initial condition of the stream. In the case of a quasi-random number generator seed parameter is used to set the dimension. If the dimension is

Name	Type	Description
		greater than the dimension that <code>brng</code> can support or is less than 1, then the dimension is assumed to be equal to 1.

Output Parameters

Name	Type	Description
<i>stream</i>	FORTRAN: TYPE (VSL_STREAM_STATE), INTENT (OUT) C: VSLStreamStatePtr*	Stream state descriptor

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_INVALID_BRNG_INDEX	BRNG index is invalid.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for <i>stream</i> .

NewStreamEx

Creates and initializes a random stream for generators with multiple initial conditions.

Syntax

Fortran:

```
status = vslnewstreamex( stream, brng, n, params )
```

C:

```
status = vslNewStreamEx( &stream, brng, n, params );
```

Description

This function provides an advanced tool to set the initial conditions for a basic generator if its input arguments imply several initialization parameters. Initial values are used to select a particular sequence generated by the basic generator *brng*. Whenever possible, use [NewStream](#),

which is analogous to `vslNewStreamEx` except that it takes only one 32-bit initial condition. In particular, `vslNewStreamEx` may be used to initialize the state table in Generalized Feedback Shift Register Generators (GFSRs). A more detailed description of this issue can be found in [VSL Notes](#).

This function is also used to pass user-defined initialization parameters of quasi-random number generators into the library. See [VSL Notes](#) for the format for their passing and registration in VSL.



NOTE. This function is not applicable for abstract basic random number generators. Please use `vsliNewAbstractStream`, `vslsNewAbstractStream` or `vsldNewAbstractStream` to utilize integer, single-precision or double-precision external random data respectively.

Input Parameters

Name	Type	Description
<i>brng</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Index of the basic generator to initialize the stream. See Table 10-2 for specific value.
<i>n</i>	FORTTRAN: INTEGER, INTENT (IN) C: unsigned int	Number of initial conditions contained in <i>params</i>
<i>params</i>	FORTTRAN: INTEGER, INTENT (IN) C: const unsigned int	Array of initial conditions necessary for the basic generator <i>brng</i> to initialize the stream. In the case of a quasi-random number generator only the first element in <i>params</i> parameter is used to set the dimension. If the dimension is greater than the dimension that <i>brng</i> can support or is less than 1, then the dimension is assumed to be equal to 1.

Output Parameters

Name	Type	Description
<i>stream</i>	FORTRAN: TYPE (VSL_STREAM_STATE), INTENT (OUT) C: VSLStreamStatePtr*	Stream state descriptor

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_INVALID_BRNG_INDEX	BRNG index is invalid.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for <i>stream</i> .

iNewAbstractStream

Creates and initializes an abstract random stream for integer arrays.

Syntax

Fortran:

```
status = vslinewabstractstream( stream, n, ibuf, icallback )
```

C:

```
status = vsliNewAbstractStream( &stream, n, ibuf, icallback );
```

Description

This function creates a new abstract stream and associates it with an integer array *ibuf* and user's callback function *icallback* that is intended for updating of *ibuf* content.

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN: INTEGER, INTENT (IN)	Size of the array <i>ibuf</i>

Name	Type	Description
	C: <code>int</code>	
<i>ibuf</i>	FORTRAN: <code>INTEGER, INTENT (IN)</code> C: <code>unsigned int*</code>	Array of <i>n</i> 32-bit integers
<i>icallback</i>	FORTRAN: See <i>Note</i> below C: See <i>Note</i> below	FORTRAN: Address of the callback function used for <i>ibuf</i> update C: Pointer to the callback function used for <i>ibuf</i> update

Output Parameters

Name	Type	Description
<i>stream</i>	FORTRAN: <code>TYPE (VSL_STREAM_STATE), TINTENT (OUT)</code> C: <code>VSLStreamStatePtr*</code>	Descriptor of the stream state structure



NOTE. Format of the callback function in Fortran:

```

INTEGER FUNCTION IUPDATEFUNC(C) ( stream, n, ibuf, nmin, nmax, idx )
TYPE (VSL_STREAM_STATE), POINTER :: stream[reference]
INTEGER (KIND=4), INTENT (IN)      :: n[reference]
INTEGER (KIND=4), INTENT (OUT)     :: ibuf[reference] (0:n-1)
INTEGER (KIND=4), INTENT (IN)      :: nmin[reference]
INTEGER (KIND=4), INTENT (IN)      :: nmax[reference]
INTEGER (KIND=4), INTENT (IN)      :: idx[reference]

```

Format of the callback function in C:

```

int iUpdateFunc( VSLStreamStatePtrstream, int* n, unsigned int ibuf[], int* nmin,
int* nmax, int* idx );

```

The callback function returns the number of elements in the array actually updated by the function. [Table 10-5](#) gives the description of the callback function parameters.

Table 10-5 icallback Callback Function Parameters

Parameters	Short Description
<i>stream</i>	Abstract stream descriptor
<i>n</i>	Size of <i>ibuf</i>
<i>ibuf</i>	Array of random numbers associated with the stream <i>stream</i>
<i>nmin</i>	Minimal quantity of numbers to update
<i>nmax</i>	Maximal quantity of numbers that can be updated
<i>idx</i>	Position in cyclic buffer <i>ibuf</i> to start update $0 \leq idx < n$.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_BAD_ARG	Parameter <i>n</i> is not positive.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for <i>stream</i> .
VSL_ERROR_NULL_PTR	Either buffer or callback function parameter is a NULL pointer.

dNewAbstractStream

Creates and initializes an abstract random stream for double precision floating-point arrays.

Syntax

Fortran:

```
status = vsldnewabstractstream( stream, n, dbuf, a, b, dcallback )
```

C:

```
status = vsldNewAbstractStream( &stream, n, dbuf, a, b, dcallback );
```

Description

This function creates a new abstract stream for double precision floating-point arrays with random numbers of the uniform distribution over interval (a,b) . The function associates the stream with a double precision array *dbuf* and user's callback function *dcallback* that is intended for updating of *dbuf* content.

Input Parameters

Name	Type	Description
<i>n</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Size of the array <i>dbuf</i>
<i>dbuf</i>	FORTTRAN: DOUBLE PRECISION, INTENT (IN) C: double*	Array of <i>n</i> double precision floating-point random numbers with uniform distribution over interval (a,b)
<i>a</i>	FORTTRAN: DOUBLE PRECISION, INTENT (IN) C: double	Left boundary <i>a</i>
<i>b</i>	FORTTRAN: DOUBLE PRECISION, INTENT (IN) C: double	Right boundary <i>b</i>
<i>dcallback</i>	FORTTRAN: See <i>Note</i> below C: See <i>Note</i> below	FORTTRAN: Address of the callback function used for update of the array <i>dbuf</i> C: Pointer to the callback function used for update of the array <i>dbuf</i>

Output Parameters

Name	Type	Description
<i>stream</i>	FORTTRAN: TYPE (VSL_STREAM_STATE), INTENT (OUT) C: VSLStreamStatePtr*	Descriptor of the stream state structure



NOTE. Format of the callback function in Fortran:

```
INTEGER FUNCTION DUPDATEFUNC[C]( stream, n, dbuf, nmin, nmax, idx )  
TYPE (VSL_STREAM_STATE),POINTER :: stream[reference]  
  
INTEGER (KIND=4), INTENT (IN)      :: n[reference]  
REAL (KIND=8),   INTENT (OUT)     :: dbuf[reference] (0:n-1)  
INTEGER (KIND=4), INTENT (IN)      :: nmin[reference]  
INTEGER (KIND=4), INTENT (IN)      :: nmax[reference]  
INTEGER (KIND=4), INTENT (IN)      :: idx[reference]
```

Format of the callback function in C:

```
int dUpdateFunc( VSLStreamStatePtr stream, int* n, double dbuf[], int* nmin, int*  
nmax, int* idx );
```

The callback function returns the number of elements in the array actually updated by the function. [Table 10-6](#) gives the description of the callback function parameters.

dcallback Callback Function Parameters

Parameters	Short Description
<i>stream</i>	Abstract stream descriptor
<i>n</i>	Size of <i>dbuf</i>
<i>dbuf</i>	Array of random numbers associated with the stream <i>stream</i>
<i>nmin</i>	Minimal quantity of numbers to update

Parameters	Short Description
<i>nmax</i>	Maximal quantity of numbers that can be updated
<i>idx</i>	Position in cyclic buffer <i>dbuf</i> to start update $0 \leq idx < n$.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_BAD_ARG	Parameter <i>n</i> is not positive.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for <i>stream</i> .
VSL_ERROR_NULL_PTR	Either buffer or callback function parameter is a NULL pointer.

sNewAbstractStream

Creates and initializes an abstract random stream for single precision floating-point arrays.

Syntax

Fortran:

```
status = vslnsnewabstractstream( stream, n, sbuf, a, b, scallback )
```

C:

```
status = vslnsNewAbstractStream( &stream, n, sbuf, a, b, scallback );
```

Description

This function creates a new abstract stream for single precision floating-point arrays with random numbers of the uniform distribution over interval (a,b). The function associates the stream with a single precision array *sbuf* and user's callback function *scallback* that is intended for updating of *sbuf* content.

Input Parameters

Name	Type	Description
<i>n</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Size of the array <i>sbuf</i>
<i>sbuf</i>	FORTTRAN: REAL, INTENT (IN) C: float*	Array of <i>n</i> single precision floating-point random numbers with uniform distribution over interval (<i>a</i> , <i>b</i>)
<i>a</i>	FORTTRAN: REAL, INTENT (IN) C: float	Left boundary <i>a</i>
<i>b</i>	FORTTRAN: REAL, INTENT (IN) C: float	Right boundary <i>b</i>
<i>scallback</i>	FORTTRAN: See <i>Note</i> below C: See <i>Note</i> below	FORTTRAN: Address of the callback function used for update of the array <i>sbuf</i> C: Pointer to the callback function used for update of the array <i>sbuf</i>

Output Parameters

Name	Type	Description
<i>stream</i>	FORTTRAN: TYPE (VSL_STREAM_STATE), INTENT (OUT) C: VSLStreamStatePtr*	Descriptor of the stream state structure

**NOTE.** Format of the callback function in Fortran:

```

INTEGER FUNCTION SUPDATEFUNC(C)( stream, n, sbuf, nmin, nmax, idx )
TYPE(VSL_STREAM_STATE), POINTER :: stream[reference]
INTEGER(KIND=4), INTENT(IN)      :: n[reference]
REAL(KIND=4), INTENT(OUT)       :: sbuf[reference](0:n-1)
INTEGER(KIND=4), INTENT(IN)      :: nmin[reference]
INTEGER(KIND=4), INTENT(IN)      :: nmax[reference]
INTEGER(KIND=4), INTENT(IN)      :: idx[reference]

```

Format of the callback function in C:

```

int sUpdateFunc( VSLStreamStatePtr stream, int* n, float sbuf[], int* nmin, int*
nmax, int* idx );

```

The callback function returns the number of elements in the array actually updated by the function. [Table 10-7](#) gives the description of the callback function parameters.

Table 10-7 scallback Callback Function Parameters

Parameters	Short Description
<i>stream</i>	Abstract stream descriptor
<i>n</i>	Size of <i>sbuf</i>
<i>sbuf</i>	Array of random numbers associated with the stream <i>stream</i>
<i>nmin</i>	Minimal quantity of numbers to update
<i>nmax</i>	Maximal quantity of numbers that can be updated
<i>idx</i>	Position in cyclic buffer <i>sbuf</i> to start update $0 \leq idx < n$.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_BAD_ARG	Parameter <i>n</i> is not positive.

VSL_ERROR_MEM_FAILURE	System cannot allocate memory for <i>stream</i> .
VSL_ERROR_NULL_PTR	Either buffer or callback function parameter is a NULL pointer.

DeleteStream

Deletes a random stream.

Syntax

Fortran:

```
status = vsldeletestream( stream )
```

C:

```
status = vslDeleteStream( &stream );
```

Description

This function deletes the random stream created by one of the initialization functions.

Input/Output Parameters

Name	Type	Description
<i>stream</i>	FORTTRAN: TYPE (VSL_STREAM_STATE), INTENT (OUT) C: VSLStreamStatePtr*	FORTTRAN: Stream state descriptor. Must have non-zero value. After the stream is successfully deleted, the descriptor becomes invalid. C: Stream state descriptor. Must have non-zero value. After the stream is successfully deleted, the pointer is set to NULL.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>srcstream</i> parameter is a NULL pointer.
VSL_ERROR_BAD_STREAM	<i>srcstream</i> is not a valid random stream.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for <i>newstream</i> .

CopyStream

Creates a copy of a random stream.

Syntax

Fortran:

```
status = vslcopystream( newstream, srcstream )
```

C:

```
status = vslCopyStream( &newstream, srcstream );
```

Description

The function creates an exact copy of *srcstream* and stores its descriptor to *newstream*.

Input Parameters

Name	Type	Description
<i>srcstream</i>	FORTTRAN: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	FORTTRAN: Descriptor of the stream to be copied C: Pointer to the stream state structure to be copied

Output Parameters

Name	Type	Description
<i>newstream</i>	FORTTRAN: TYPE (VSL_STREAM_STATE), INTENT (OUT) C: VSLStreamStatePtr*	Copied stream descriptor

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>srcstream</i> parameter is a NULL pointer.
VSL_ERROR_BAD_STREAM	<i>srcstream</i> is not a valid random stream.

VSL_ERROR_MEM_FAILURE

System cannot allocate memory for *newstream*.

CopyStreamState

Creates a copy of a random stream state.

Syntax

Fortran:

```
status = vslcopystreamstate( deststream, srcstream )
```

C:

```
status = vslCopyStreamState( deststream, srcstream );
```

Description

The function copies a stream state from *srcstream* to the existing *deststream* stream. Both the streams should be generated by the same basic generator. En error message is generated when the index of the BRNG that produced *deststream* stream differs from the index of the BRNG that generated *srcstream* stream.

Unlike [CopyStream](#) function, which creates a new stream and copies both the stream state and other data from *srcstream*, the function `CopyStreamState` copies only *srcstream* stream state data to the generated *deststream* stream.

Input Parameters

Name	Type	Description
<i>srcstream</i>	FORTTRAN:	FORTTRAN: Descriptor of the destination stream where the state of <i>srcstream</i> stream is copied
	TYPE (VSL_STREAM_STATE),	
	INTENT (IN)	C: Pointer to the stream state structure, from which the state structure is copied
	C: VSLStreamStatePtr	

Output Parameters

Name	Type	Description
<i>deststream</i>	FORTRAN: TYPE (VSL_STREAM_STATE), INTENT (OUT) C: VSLStreamStatePtr	FORTRAN: Descriptor of the stream with the state to be copied C: Pointer to the stream state structure where the stream state is copied

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	Either <i>srcstream</i> or <i>deststream</i> is a NULL pointer.
VSL_ERROR_BAD_STREAM	Either <i>srcstream</i> or <i>deststream</i> is not a valid random stream.
VSL_ERROR_BRNGS_INCOMPATIBLE	BRNG associated with <i>srcstream</i> is not compatible with BRNG associated with <i>deststream</i> .

SaveStreamF

Writes random stream descriptive data to binary file.

Syntax

Fortran:

```
errstatus = vslsavestreamf( stream, fname )
```

C:

```
errstatus = vslSaveStreamF( stream, fname );
```

Description

This function writes the random stream descriptive data to the binary file. Random stream descriptive data is saved to the binary file with the name *fname*. Random stream *stream* must be a valid stream created by [NewStream](#)-like or [CopyStream](#)-like service routines. If the stream cannot be saved to the file, *errstatus* has a non-zero value. Random stream can be read from the binary file using [LoadStreamF](#) function.

Input Parameters

Name	Type	Description
<i>stream</i>	FORTTRAN: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	Random stream to be written to the file
<i>fname</i>	FORTTRAN: CHARACTER (*), INTENT (IN) C: char*	FORTTRAN: File name specified as a C-style null-terminated string C: File name specified as a Fortran-style character string

Output Parameters

Name	Type	Description
<i>errstatus</i>	FORTTRAN: INTEGER C: int	Error status of the operation

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	Either <i>fname</i> or <i>stream</i> is a NULL pointer.
VSL_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_ERROR_FILE_OPEN	Indicates an error in opening the file.
VSL_ERROR_FILE_WRITE	Indicates an error in writing the file.
VSL_ERROR_FILE_CLOSE	Indicates an error in closing the file.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for internal needs.

LoadStreamF

Creates new stream and reads stream descriptive data from binary file.

Syntax

Fortran:

```
errstatus = vslloadstreamf( stream, fname )
```

C:

```
errstatus = vslLoadStreamF( &stream, fname );
```

Description

This function creates a new stream and reads stream descriptive data from the binary file. A new random stream is created using the stream descriptive data from the binary file with the name *fname*. If the stream cannot be read (for example, I/O error occurs or the file format is invalid), *errstatus* has a non-zero value. To save random stream to the file, use [SaveStreamF](#) function.

Input Parameters

Name	Type	Description
<i>fname</i>	FORTTRAN: CHARACTER (*), INTENT (IN)	FORTTRAN: File name specified as a C-style null-terminated string
	C: char*	C: File name specified as a Fortran-style character string

Output Parameters

Name	Type	Description
<i>stream</i>	FORTTRAN: TYPE (VSL_STREAM_STATE), INTENT (OUT)	FORTTRAN: Descriptor of a new random stream
	C: VSLStreamStatePtr*	C: Pointer to a new random stream

Name	Type	Description
<i>errstatus</i>	FORTRAN: INTEGER C: int	Error status of the operation
Return Values		
VSL_ERROR_OK, VSL_STATUS_OK		Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR		<i>fname</i> is a NULL pointer.
VSL_ERROR_FILE_OPEN		Indicates an error in opening the file.
VSL_ERROR_FILE_WRITE		Indicates an error in writing the file.
VSL_ERROR_FILE_CLOSE		Indicates an error in closing the file.
VSL_ERROR_MEM_FAILURE		System cannot allocate memory for internal needs.
VSL_ERROR_BAD_FILE_FORMAT		Unknown file format.
VSL_ERROR_UNSUPPORTED_FILE_VER		File format version is unsupported.

LeapfrogStream

Initializes a stream using the leapfrog method.

Syntax

Fortran:

```
status = vslleapfrogstream( stream, k, nstreams )
```

C:

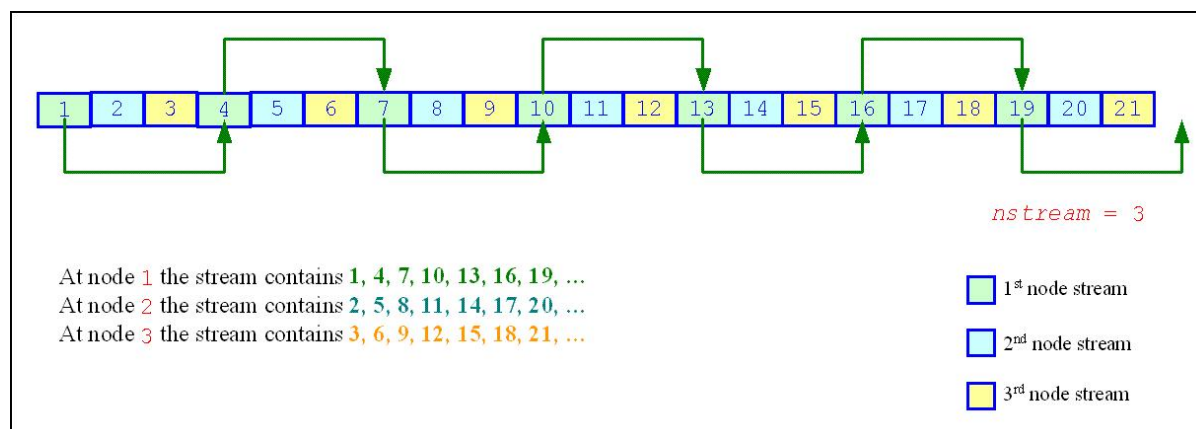
```
status = vslLeapfrogStream( stream, k, nstreams );
```

Description

The function allows generating random numbers in a random stream with non-unit stride. This feature is particularly useful in distributing random numbers from original stream across *nstreams* buffers without generating the original random sequence with subsequent manual distribution. One of the important applications of the leapfrog method is splitting the original sequence into non-overlapping subsequences across *nstreams* computational nodes. The

function initializes the original random stream (see [Figure 10-1](#)) to generate random numbers for the computational node k , $0 \leq k < nstreams$, where $nstreams$ is the largest number of computational nodes used.

Figure 10-1 Leapfrog Method



The leapfrog method is supported only for those basic generators that allow splitting elements by the leapfrog method, which is more efficient than simply generating them by a generator with subsequent manual distribution across computational nodes. See [VSL Notes](#) for details.

For quasi-random basic generators the leapfrog method allows generating individual components of quasi-random vectors instead of whole quasi-random vectors. In this case $nstreams$ parameter should be equal to the dimension of the quasi-random vector while k parameter should be the index of a component to be generated ($0 \leq k < nstreams$). Other parameters values are not allowed.

The following code examples illustrate the initialization of three independent streams using the leapfrog method:

Example 10-1 FORTRAN Code for Leapfrog Method

```
...
TYPE(VSL_STREAM_STATE)  ::stream1
TYPE(VSL_STREAM_STATE)  ::stream2
TYPE(VSL_STREAM_STATE)  ::stream3
! Creating 3 identical streams
status = vslnewstream(stream1, VSL_BRNG_MCG31, 174)
status = vslcopystream(stream2, stream1)
status = vslcopystream(stream3, stream1)
! Leapfrogging the streams

status = vslleapfrogstream(stream1, 0, 3)
status = vslleapfrogstream(stream2, 1, 3)
status = vslleapfrogstream(stream3, 2, 3)
! Generating random numbers

...
! Deleting the streams

status = vsldeletestream(stream1)
status = vsldeletestream(stream2)
status = vsldeletestream(stream3)
...
```

Example 10-2 C Code for Leapfrog Method

```
...
VSLStreamStatePtr stream1;
VSLStreamStatePtr stream2;
VSLStreamStatePtr stream3;
/* Creating 3 identical streams */
status = vslNewStream(&stream1, VSL_BRNG_MCG31, 174);
status = vslCopyStream(&stream2, stream1);
status = vslCopyStream(&stream3, stream1);
/* Leapfrogging the streams
*/
status = vslLeapfrogStream(stream1, 0, 3);
status = vslLeapfrogStream(stream2, 1, 3);
status = vslLeapfrogStream(stream3, 2, 3);
/* Generating random numbers
*/
...
/* Deleting the streams
*/
status = vslDeleteStream(&stream1);
status =
vslDeleteStream(&stream2);
status = vslDeleteStream(&stream3);
...
```

Input Parameters

Name	Type	Description
<i>stream</i>	FORTTRAN: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	FORTTRAN: Descriptor of the stream to which leapfrog method is applied C: Pointer to the stream state structure to which leapfrog method is applied
<i>k</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Index of the computational node, or stream number
<i>nstreams</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Largest number of computational nodes, or stride

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_ERROR_LEAPFROG_UNSUPPORTED	BRNG does not support Leapfrog method.

SkipAheadStream

Initializes a stream using the block-splitting method.

Syntax

Fortran:

```
status = vslskipaheadstream( stream, nskip )
```

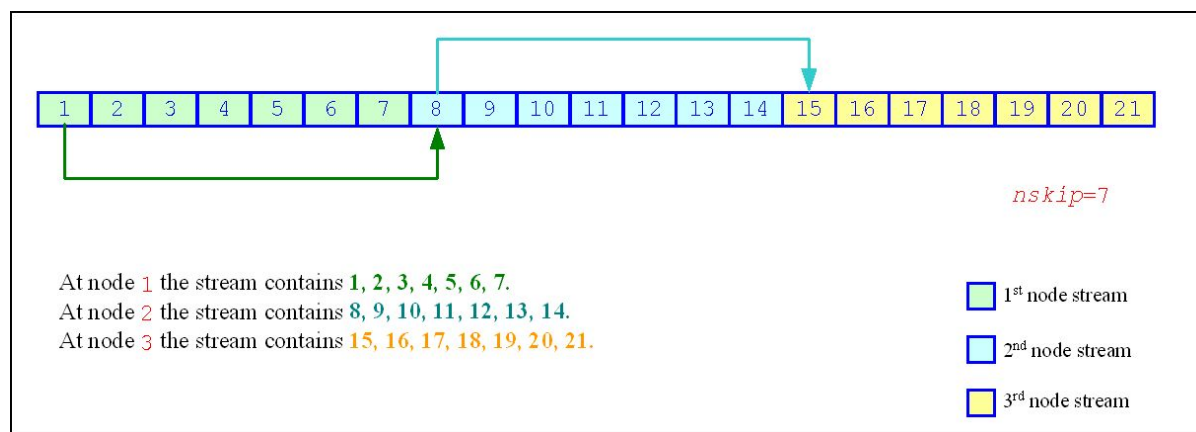
C:

```
status = vslSkipAheadStream( stream, nskip);
```

Description

This function skips a given number of elements in a random stream. This feature is particularly useful in distributing random numbers from original random stream across different computational nodes. If the largest number of random numbers used by a computational node is *nskip*, then the original random sequence may be split by `SkipAheadStream` into non-overlapping blocks of *nskip* size so that each block corresponds to the respective computational node. The number of computational nodes is unlimited. This method is known as the block-splitting method or as the skip-ahead method. (see [Figure 10-2](#)).

Figure 10-2 Block-Splitting Method



The skip-ahead method is supported only for those basic generators that allow skipping elements by the skip-ahead method, which is more efficient than simply generating them by generator with subsequent manual skipping. See [VSL Notes](#) for details.

Please note that for quasi-random basic generators the skip-ahead method works with components of quasi-random vectors rather than with whole quasi-random vectors. Thus to skip *NS* quasi-random vectors, set *nskip* parameter equal to the *NS*DIMEN*, where *DIMEN* is the dimension of quasi-random vector.

The following code examples illustrate how to initialize three independent streams using `SkipAheadStream` function:

Example 10-3 FORTRAN Code for Block-Splitting Method

```
...
type(VSL_STREAM_STATE)  ::stream1
type(VSL_STREAM_STATE)  ::stream2
type(VSL_STREAM_STATE)  ::stream3
! Creating the 1st stream
status = vslnewstream(stream1, VSL_BRNG_MCG31, 174)
! Skipping ahead by 7 elements the 2nd stream

status = vslcopystream(stream2, stream1);
status = vslskipaheadstream(stream2, 7);
! Skipping ahead by 7 elements the 3rd stream

status = vslcopystream(stream3, stream2);
status = vslskipaheadstream(stream3, 7);
! Generating random numbers

...
! Deleting the streams

status = vsldeletestream(stream1)
status = vsldeletestream(stream2)
status = vsldeletestream(stream3)
...
```

Example 10-4 C Code for Block-Splitting Method

```
VSLStreamStatePtr stream1;
VSLStreamStatePtr stream2;
VSLStreamStatePtr stream3;
/* Creating the 1st stream
*/
status = vslNewStream(&stream1, VSL_BRNG_MCG31, 174);
/* Skipping ahead by 7 elements the 2nd stream */
status = vslCopyStream(&stream2, stream1);
status = vslSkipAheadStream(stream2, 7);
/* Skipping ahead by 7 elements the 3rd stream */
status = vslCopyStream(&stream3, stream2);
status = vslSkipAheadStream(stream3, 7);
/* Generating random numbers
*/
...
/* Deleting the streams
*/
status = vslDeleteStream(&stream1);
status = vslDeleteStream(&stream2);
status = vslDeleteStream(&stream3);
...
```

Input Parameters

Name	Type	Description
<i>stream</i>	FORTTRAN: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	FORTTRAN: Descriptor of the stream to which block-splitting method is applied C: Pointer to the stream state structure to which block-splitting method is applied

Name	Type	Description
<i>nskip</i>	FORTTRAN: INTEGER(KIND=8), INTENT(IN) C: long long	Number of skipped elements

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_ERROR_SKIPAHEAD_UNSUPPORTED	BRNG does not support Skip-Ahead method.

GetStreamStateBrng

Returns index of a basic generator used for generation of a given random stream.

Syntax

Fortran:

```
brng = vslgetstreamstatebrng( stream )
```

C:

```
brng = vslGetStreamStateBrng( stream );
```

Description

This function retrieves the index of a basic generator used for generation of a given random stream.

Input Parameters

Name	Type	Description
<i>stream</i>	FORTTRAN: TYPE(VSL_STREAM_STATE), TINTENT(IN) C: long long	FORTTRAN: Descriptor of the stream state C: Pointer to the stream state structure

Name	Type	Description
	C: <code>VSLStreamStatePtr</code>	

Output Parameters

Name	Type	Description
<i>brng</i>	FORTTRAN: <code>INTEGER</code> C: <code>int</code>	Index of the basic generator assigned for the generation of <i>stream</i> ; negative in case of an error

Return Values

<code>VSL_ERROR_NULL_PTR</code>	<i>stream</i> is a <code>NULL</code> pointer.
<code>VSL_ERROR_BAD_STREAM</code>	<i>stream</i> is not a valid random stream.

GetNumRegBrngs

Obtains the number of currently registered basic generators.

Syntax

Fortran:

```
nregbrngs = vslgetnumregbrngs( )
```

C:

```
nregbrngs = vslGetNumRegBrngs( void );
```

Description

This function obtains the number of currently registered basic generators. Whenever user registers a user-designed basic generator, the number of registered basic generators is incremented. The maximum number of basic generators that can be registered is determined by `VSL_MAX_REG_BRNGS` parameter.

Output Parameters

Name	Type	Description
<i>nregbrngs</i>	FORTRAN: INTEGER C: int	Number of basic generators registered at the moment of the function call

Distribution Generators

VSL routines are used to generate random numbers with different types of distribution. Each function group is introduced below by the type of underlying distribution and contains a short description of its functionality, as well as specifications of the call sequence for both FORTRAN and C-interface and the explanation of input and output parameters. [Table 10-8](#) and [Table 10-9](#) list the random number generator routines, together with used data types, output distributions, and sets correspondence between data types of the generator routines and called basic random number generators.

Table 10-8 Continuous Distribution Generators

Type of Distribution	Data Types	BRNG Type	Data Description
Uniform	s, d	s, d	Uniform continuous distribution on the interval $[a, b]$.
Gaussian	s, d	s, d	Normal (Gaussian) distribution.
GaussianMV	s, d	s, d	Multivariate normal (Gaussian) distribution.
Exponential	s, d	s, d	Exponential distribution.
Laplace	s, d	s, d	Laplace distribution (double exponential distribution).
Weibull	s, d	s, d	Weibull distribution.
Cauchy	s, d	s, d	Cauchy distribution.
Rayleigh	s, d	s, d	Rayleigh distribution.
Lognormal	s, d	s, d	Lognormal distribution.
Gumbel	s, d	s, d	Gumbel (extreme value) distribution.

Type of Distribution	Data Types	BRNG Data Type	Data Description
Gamma	s, d	s, d	Gamma distribution.
Beta	s, d	s, d	Beta distribution.

Table 10-9 Discrete Distribution Generators

Type of Distribution	Data Types	BRNG Data Type	Description
Uniform	i	d	Uniform discrete distribution on the interval $[a, b]$.
UniformBits	i	i	Generator of integer random values with uniform bit distribution.
Bernoulli	i	s	Bernoulli distribution.
Geometric	i	s	Geometric distribution.
Binomial	i	d	Binomial distribution.
Hypergeometric	i	d	Hypergeometric distribution.
Poisson	i	s (for VSL_METHOD_IPOISSON_POISNORM) s (for distribution parameter $\lambda \geq 27$) and d (for $\lambda < 27$) (for VSL_METHOD_IPOISSON_PTPE)	Poisson distribution.
PoissonV	i	s	Poisson distribution with varying mean.
NegBinomial	i	d	Negative binomial distribution, or Pascal distribution.

The library provides two modes of random number generation, accurate and fast. Accurate generation mode is intended for the applications that are highly demanding to accuracy of calculations. When used in this mode, the generators produce random numbers lying completely within definitional domain for all values of the distribution parameters. For example, random numbers obtained from the generator of continuous distribution that is uniform on interval $[a,b]$ belong to this interval irrespective of what a and b values may be. Fast mode provides high performance of generation and also guaranties that generated random numbers belong to the definitional domain except for some specific values of distribution parameters. The generation mode is set by specifying relevant value of the method parameter in generator routines. List of distributions that support accurate mode of generation is given in the table below.

Table 10-10 Distribution Generators Supporting Accurate Mode

Type of Distribution	Data Types
Uniform	s, d
Exponential	s, d
Weibull	s, d
Rayleigh	s, d
Lognormal	s, d
Gamma	s, d
Beta	s, d

See additional details about accurate and fast mode of random number generation in [VSL Notes](#).

Continuous Distributions

This section describes routines for generating random numbers with continuous distribution.

Uniform

Generates random numbers with uniform distribution.

Syntax

Fortran:

```
status = vsrnguniform( method, stream, n, r, a, b )
```

```
status = vdrnguniform( method, stream, n, r, a, b )
```

C:

```
status = vsRngUniform( method, stream, n, r, a, b );
```

```
status = vdRngUniform( method, stream, n, r, a, b );
```

Description

This function generates random numbers uniformly distributed over the interval $[a, b]$, where a, b are the left and right bounds of the interval, respectively, and $a, b \in \mathbb{R}$; $a > b$.

The probability density function is given by:

$$f_{a,b}(x) = \begin{cases} \frac{1}{b-a}, & x \in [a, b] \\ 0, & x \notin [a, b] \end{cases}, \quad -\infty < x < +\infty$$

The cumulative distribution function is as follows:

$$f_{a,b}(x) = \begin{cases} 0, & x < a \\ \frac{x-a}{b-a}, & a \leq x \leq b \\ 1, & x \geq b \end{cases}, \quad -\infty < x < +\infty.$$

Input Parameters

Name	Type	Description
<i>method</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Generation method; the specific values are as follows: VSL_METHOD_SUNIFORM_STD VSL_METHOD_DUNIFORM_STD VSL_METHOD_SUNIFORM_STD_ACCURATE VSL_METHOD_DUNIFORM_STD_ACCURATE Standard method.
<i>stream</i>	FORTTRAN: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	FORTTRAN: Descriptor of the stream state structure. C: Pointer to the stream state structure
<i>n</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Number of random values to be generated
<i>a</i>	FORTTRAN: REAL, INTENT (IN) for vsrnguniform DOUBLE PRECISION, INTENT (IN) for vdrnguniform C: float for vsRngUniform double for vdRngUniform	Left bound a
<i>b</i>	FORTTRAN: REAL, INTENT (IN) for vsrnguniform	Right bound b

Name	Type	Description
	DOUBLE PRECISION, INTENT(IN) for vdrnguniform	
	C: float for vsRngUniform double for vdRngUniform	

Output Parameters

Name	Type	Description
<i>r</i>	FORTTRAN: REAL, INTENT(OUT) for vsrnguniform DOUBLE PRECISION, INTENT(OUT) for vdrnguniform C: float* for vsRngUniform double* for vdRngUniform	Vector of <i>n</i> random numbers uniformly distributed over the interval $[a, b]$

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.
VSL_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Gaussian

Generates normally distributed random numbers.

Syntax

Fortran:

```
status = vsrnggaussian( method, stream, n, r, a, sigma )
```

```
status = vdrnggaussian( method, stream, n, r, a, sigma )
```

C:

```
status = vsRngGaussian( method, stream, n, r, a, sigma );
```

```
status = vdRngGaussian( method, stream, n, r, a, sigma );
```

Description

This function generates random numbers with normal (Gaussian) distribution with mean value a and standard deviation σ , where

$a, \sigma \in \mathbb{R}$; $\sigma > 0$.

The probability density function is given by:

$$f_{a, \sigma}(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-a)^2}{2\sigma^2}\right), \quad -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a, \sigma}(x) = \int_{-\infty}^x \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y-a)^2}{2\sigma^2}\right) dy, \quad -\infty < x < +\infty.$$

The cumulative distribution function $F_{a,\sigma}(x)$ can be expressed in terms of standard normal distribution $\Phi(x)$ as

$$F_{a,\sigma}(x) = \Phi((x - a)/\sigma)$$

Input Parameters

Name	Type	Description
<i>method</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Generation method. The specific values are as follows: VSL_METHOD_SGAUSSIAN_BOXMULLER VSL_METHOD_SGAUSSIAN_BOXMULLER2 VSL_METHOD_SGAUSSIAN_ICDF VSL_METHOD_DGAUSSIAN_BOXMULLER VSL_METHOD_DGAUSSIAN_BOXMULLER2 VSL_METHOD_DGAUSSIAN_ICDF See brief description of the methods BOXMULLER, BOXMULLER2, and ICDF in Table 10-1
<i>stream</i>	FORTTRAN: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	FORTTRAN: Descriptor of the stream state structure. C: Pointer to the stream state structure
<i>n</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Number of random values to be generated
<i>a</i>	FORTTRAN: REAL, INTENT (IN) for vsrnggaussian DOUBLE PRECISION, INTENT (IN) for vdrnggaussian	Mean value <i>a</i> .

Name	Type	Description
	C: float for vsRngGaussian double for vdRngGaussian	
<i>sigma</i>	FORTTRAN: REAL, INTENT (IN) for vsrnggaussian DOUBLE PRECISION, INTENT (IN) for vdrnggaussian C: float for vsRngGaussian double for vdRngGaussian	Standard deviation σ .

Output Parameters

Name	Type	Description
<i>r</i>	FORTTRAN: REAL, INTENT (OUT) for vsrnggaussian DOUBLE PRECISION, INTENT (OUT) for vdrnggaussian C: float* for vsRngGaussian double* for vdRngGaussian	Vector of n normally distributed random numbers

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.

VSL_ERROR_BAD_UPDATE

Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> n_{\max}$.

VSL_ERROR_NO_NUMBERS

Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

GaussianMV

Generates random numbers from multivariate normal distribution.

Syntax

Fortran:

```
status = vsrnggaussianmv( method, stream, n, r, dimen, mstorage, a, t )
status = vdrnggaussianmv( method, stream, n, r, dimen, mstorage, a, t )
```

C:

```
status = vsRngGaussianMV( method, stream, n, r, dimen, mstorage, a, t );
status = vdRngGaussianMV( method, stream, n, r, dimen, mstorage, a, t );
```

Description

This function generates random numbers with d -variate normal (Gaussian) distribution with mean value a and variance-covariance matrix C , where $a \in R^d$; C is a $d \times d$ symmetric positive-definite matrix.

The probability density function is given by:

$$f_{a,C}(x) = \frac{1}{\sqrt{\det(2\pi C)}} \exp(-1/2(x - a)^T C^{-1}(x - a)),$$

where $x \in R^d$.

Matrix C can be represented as $C = TT^T$, where T is a lower triangular matrix - Cholesky factor of C .

Instead of variance-covariance matrix c the generation routines require Cholesky factor of c in input. To compute Cholesky factor of matrix c , the user may call MKL LAPACK routines for matrix factorization: `?potrf` or `?pptrf` for `v?RngGaussianMV/v?rnggaussianmv` routines (`?` means either `s` or `d` for single and double precision respectively). See [Application Notes](#) for more details.

Input Parameters

Name	Type	Description
<i>method</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Generation method. The specific values are as follows: VSL_METHOD_SGAUSSIANMV_BOXMULLER VSL_METHOD_SGAUSSIANMV_BOXMULLER2 VSL_METHOD_SGAUSSIANMV_ICDF VSL_METHOD_DGAUSSIANMV_BOXMULLER VSL_METHOD_DGAUSSIANMV_BOXMULLER2 VSL_METHOD_DGAUSSIANMV_ICDF See brief description of the methods BOXMULLER, BOXMULLER2, and ICDF in Table 10-1
<i>stream</i>	FORTTRAN: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	FORTTRAN: Descriptor of the stream state structure. C: Pointer to the stream state structure
<i>n</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Number of random values to be generated
<i>dimen</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Dimension d ($d \geq 1$) of output random vectors

Name	Type	Description
<i>mstorage</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	FORTTRAN: Matrix storage scheme for upper triangular matrix T^T . The routine supports three matrix storage schemes: • VSL_MATRIX_STORAGE_FULL— all $d \times d$ elements of the matrix T^T are passed, however, only the upper triangle part is actually used in the routine. • VSL_MATRIX_STORAGE_PACKED— upper triangle elements of T^T are packed by rows into a one-dimensional array. • VSL_MATRIX_STORAGE_DIAGONAL— only diagonal elements of T^T are passed. C: Matrix storage scheme for lower triangular matrix T . The routine supports three matrix storage schemes: • VSL_MATRIX_STORAGE_FULL— all $d \times d$ elements of the matrix T are passed, however, only the lower triangle part is actually used in the routine. • VSL_MATRIX_STORAGE_PACKED— lower triangle elements of T are packed by rows into a one-dimensional array. • VSL_MATRIX_STORAGE_DIAGONAL— only diagonal elements of T are passed.
<i>a</i>	FORTTRAN: REAL, INTENT (IN) for vsrnggaussianmv DOUBLE PRECISION, INTENT (IN) for vdrnggaussianmv	Mean vector a of dimension d

Name	Type	Description
	C: float* for vsRngGaussianMV double* for vdRngGaussianMV	
t	FORTTRAN: REAL, INTENT (IN) for vsrnggaussianmv DOUBLE PRECISION, INTENT (IN) for vdrnggaussianmv C: float* for vsRngGaussianMV double* for vdRngGaussianMV	FORTTRAN: Elements of the upper triangular matrix passed according to the matrix T^T storage scheme $mstorage$. C: Elements of the lower triangular matrix passed according to the matrix T storage scheme $mstorage$.

Output Parameters

Name	Type	Description
r	FORTTRAN: REAL, INTENT (OUT) for vsrnggaussianmv DOUBLE PRECISION, INTENT (OUT) for vdrnggaussianmv C: float* for vsRngGaussianMV double* for vdRngGaussianMV	Array of n random vectors of dimension $dimen$

Application Notes

Since matrices are stored in Fortran by columns, while in C they are stored by rows, the usage of MKL factorization routines (assuming Fortran matrices storage) in combination with multivariate normal RNG (assuming C matrix storage) is slightly different in C and Fortran. The following tables help in using these routines in C and Fortran. For further information please refer to the appropriate VSL example file.

Table 10-11 Using Cholesky Factorization Routines in Fortran

Matrix Storage Scheme	Variance-Covariance Matrix Argument	Factorization Routine	UPLO Parameter in Factorization Routine	Result of Factorization as Input Argument for RNG
VSL_MATRIX_STORAGE_FULL	C in Fortran two-dimensional array	spotrf for vsrnggaussianmv dpotrf for vdrnggaussianmv	'U'	Upper triangle of T^T . Lower triangle is not used.
VSL_MATRIX_STORAGE_PACKED	Lower triangle of C packed by columns into one-dimensional array	spptrf for vsrnggaussianmv dpptrf for vdrnggaussianmv	'L'	Upper triangle of T^T packed by rows into one-dimensional array.

Table 10-12 Using Cholesky Factorization Routines in C

Matrix Storage Scheme	Variance-Covariance Matrix Argument	Factorization Routine	UPLO Parameter in Factorization Routine	Result of Factorization as Input Argument for RNG
VSL_MATRIX_STORAGE_FULL	C in C two-dimensional array	spotrf for vsRngGaussianMV dpotrf for vdRngGaussianMV	'U'	Upper triangle of T^T . Lower triangle is not used.

Matrix Storage Scheme	Variance-CovarianceFactorization Matrix Argument Routine	UPLO Parameter in FactorizationArgument Routine	Result of Factorization as Input Argument for RNG
VSL_MATRIX_STORAGE_PACKED	Lower triangle of C packed by columns into one-dimensional array	spptrf for vsRngGaussianMV dpptrf for vdRngGaussianMV	'L' Upper triangle of T^T packed by rows into one-dimensional array.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> n_{\max}$.
VSL_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Exponential

Generates exponentially distributed random numbers.

Syntax

Fortran:

```
status = vsrngexponential( method, stream, n, r, a, beta )  
status = vdrngexponential( method, stream, n, r, a, beta )
```

C:

```
status = vsRngExponential( method, stream, n, r, a, beta );
status = vdRngExponential( method, stream, n, r, a, beta );
```

Description

This function generates random numbers with exponential distribution that has displacement a and scalefactor β , where $a, \beta \in \mathbb{R}$; $\beta > 0$.

The probability density function is given by:

$$f_{a,\beta}(x) = \begin{cases} \frac{1}{\beta} \exp (-(x - a)) / \beta , & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \begin{cases} 1 - \exp (-(x - a)) / \beta , & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

Input Parameters

Name	Type	Description
<i>method</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Generation method. The specific values are as follows: VSL_METHOD_SEXPONENTIAL_ICDF VSL_METHOD_DEXPONENTIAL_ICDF VSL_METHOD_SEXPONENTIAL_ICDF_ACCURATE VSL_METHOD_DEXPONENTIAL_ICDF_ACCURATE Inverse cumulative distribution function method

Name	Type	Description
<i>stream</i>	FORTTRAN: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	FORTTRAN: Descriptor of the stream state structure. C: Pointer to the stream state structure
<i>n</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Number of random values to be generated
<i>a</i>	FORTTRAN: REAL, INTENT (IN) for vsrngexponential DOUBLE PRECISION, INTENT (IN) for vdrngexponential C: float for vsRngExponential C: double for vdRngExponential	Displacement <i>a</i>
<i>beta</i>	FORTTRAN: REAL, INTENT (IN) for vsrngexponential DOUBLE PRECISION, INTENT (IN) for vdrngexponential C: float for vsRngExponential double for vdRngExponential	Scalefactor β .

Output Parameters

Name	Type	Description
<i>r</i>	FORTTRAN: REAL, INTENT (OUT) for vsrngexponential DOUBLE PRECISION, INTENT (OUT) for vdrngexponential C: float* for vsRngExponential double* for vdRngExponential	Vector of <i>n</i> exponentially distributed random numbers

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Laplace

Generates random numbers with Laplace distribution.

Syntax

Fortran:

```
status = vsrnglaplace( method, stream, n, r, a, beta )
status = vdrnglaplace( method, stream, n, r, a, beta )
```

C:

```
status = vsRngLaplace( method, stream, n, r, a, beta );
```

```
status = vdRngLaplace( method, stream, n, r, a, beta );
```

Description

This function generates random numbers with Laplace distribution with mean value (or average) a and scalefactor β , where $a, \beta \in \mathbb{R}$; $\beta > 0$. The scalefactor value determines the standard deviation as

$$\sigma = \beta\sqrt{2}$$

The probability density function is given by:

$$f_{a,\beta}(x) = \frac{1}{\sqrt{2\beta}} \exp\left(-\frac{|x-a|}{\beta}\right), -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$f_{a,\beta}(x) = \begin{cases} \frac{1}{2} \exp\left(-\frac{|x-a|}{\beta}\right), & x \geq a \\ 1 - \frac{1}{2} \exp\left(-\frac{|x-a|}{\beta}\right), & x < a \end{cases}, -\infty < x < +\infty.$$

Input Parameters

Name	Type	Description
<i>method</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Generation method. The specific values are as follows: VSL_METHOD_SLAPLACE_ICDF VSL_METHOD_DLAPLACE_ICDF Inverse cumulative distribution function method
<i>stream</i>	FORTTRAN: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	FORTTRAN: Descriptor of the stream state structure. C: Pointer to the stream state structure
<i>n</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Number of random values to be generated
<i>a</i>	FORTTRAN: REAL, INTENT (IN) for vsrnglaplace DOUBLE PRECISION, INTENT (IN) for vdrnglaplace C: float for vsRngLaplace double for vdRngLaplace	Mean value <i>a</i>
<i>beta</i>	FORTTRAN: REAL, INTENT (IN) for vsrnglaplace DOUBLE PRECISION, INTENT (IN) for vdrnglaplace C: float for vsRngLaplace	Scalefactor β .

Name	Type	Description
	double	for vdRngLaplace

Output Parameters

Name	Type	Description
<i>r</i>	FORTTRAN: REAL, INTENT(OUT) for vsrnglaplace DOUBLE PRECISION, INTENT(OUT) for vdrnglaplace C: float* for vsRngLaplace double* for vdRngLaplace	Vector of <i>n</i> Laplace distributed random numbers

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, <i>< 0 or > nmax</i> .
VSL_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Weibull

Generates Weibull distributed random numbers.

Syntax

Fortran:

```
status = vsrngweibull( method, stream, n, r, alpha, a, beta )
status = vdrngweibull( method, stream, n, r, alpha, a, beta )
```

C:

```
status = vsRngWeibull( method, stream, n, r, alpha, a, beta );
status = vdRngWeibull( method, stream, n, r, alpha, a, beta );
```

Description

This function generates Weibull distributed random numbers with displacement a , scalefactor β , and shape α , where $\alpha, \beta, a \in \mathbb{R}$; $\alpha > 0$, $\beta > 0$.

The probability density function is given by:

$$f_{a,\alpha,\beta}(x) = \begin{cases} \frac{\alpha}{\beta^\alpha} (x - a)^{\alpha-1} \exp\left(-\left(\frac{x - a}{\beta}\right)^\alpha\right), & x \geq a \\ 0, & x < a \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,\alpha,\beta}(x) = \begin{cases} 1 - \exp\left(-\left(\frac{x - a}{\beta}\right)^\alpha\right), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

Input Parameters

Name	Type	Description
<i>method</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Generation method. The specific values are as follows: VSL_METHOD_SWEIBULL_ICDF VSL_METHOD_DWEIBULL_ICDF VSL_METHOD_SWEIBULL_ICDF_ACCURATE VSL_METHOD_DWEIBULL_ICDF_ACCURATE Inverse cumulative distribution function method
<i>stream</i>	FORTTRAN: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	FORTTRAN: Descriptor of the stream state structure. C: Pointer to the stream state structure
<i>n</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Number of random values to be generated
<i>alpha</i>	FORTTRAN: REAL, INTENT (IN) for vsrngweibull DOUBLE PRECISION, INTENT (IN) for vdrngweibull C: float for vsRngWeibull double for vdRngWeibull	Shape α .
<i>a</i>	FORTTRAN: REAL, INTENT (IN) for vsrngweibull	Displacement a

Name	Type	Description
	DOUBLE PRECISION, INTENT(IN) for vdrngweibull C: float for vsRngWeibull double for vdRngWeibull	
<i>beta</i>	FORTTRAN: REAL, INTENT(IN) for vsrngweibull DOUBLE PRECISION, INTENT(IN) for vdrngweibull C: float for vsRngWeibull double for vdRngWeibull	Scalefactor β .

Output Parameters

Name	Type	Description
<i>r</i>	FORTTRAN: REAL, INTENT(OUT) for vsrngweibull DOUBLE PRECISION, INTENT(OUT) for vdrngweibull C: float* for vsRngWeibull double* for vdRngWeibull	Vector of n Weibull distributed random numbers

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.

VSL_ERROR_BAD_STREAM

stream is not a valid random stream.

VSL_ERROR_BAD_UPDATE

Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> n_{\max}$.

VSL_ERROR_NO_NUMBERS

Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Cauchy

Generates Cauchy distributed random values.

Syntax

Fortran:

```
status = vsrngcauchy( method, stream, n, r, a, beta )
```

```
status = vdrngcauchy( method, stream, n, r, a, beta )
```

C:

```
status = vsRngCauchy( method, stream, n, r, a, beta );
```

```
status = vdRngCauchy( method, stream, n, r, a, beta );
```

Description

This function generates Cauchy distributed random numbers with displacement a and scalefactor β , where $a, \beta \in \mathbb{R}$; $\beta > 0$.

The probability density function is given by:

$$f_{a,\beta}(x) = \frac{1}{\pi\beta \left(1 + \left(\frac{x-a}{\beta} \right)^2 \right)}, \quad -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \frac{1}{2} + \frac{1}{\pi} \arctan\left(\frac{x-a}{\beta}\right), -\infty < x < +\infty.$$

Input Parameters

Name	Type	Description
<i>method</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Generation method. The specific values are as follows: VSL_METHOD_SCAUCHY_ICDF VSL_METHOD_DCAUCHY_ICDF Inverse cumulative distribution function method
<i>stream</i>	FORTRAN: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	FORTRAN: Descriptor of the stream state structure. C: Pointer to the stream state structure
<i>n</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Number of random values to be generated
<i>a</i>	FORTRAN: REAL, INTENT (IN) for vsrngcauchy DOUBLE PRECISION, INTENT (IN) for vdrngcauchy C: float for vsRngCauchy double for vdRngCauchy	Displacement <i>a</i> .
<i>beta</i>	FORTRAN: REAL, INTENT (IN) for vsrngcauchy	Scalefactor β .

Name	Type	Description
	DOUBLE PRECISION, INTENT (IN) for vdrngcauchy	
	C: float for vsRngCauchy double for vdRngCauchy	

Output Parameters

Name	Type	Description
<i>r</i>	FORTRAN: REAL, INTENT (OUT) for vsrngcauchy	Vector of <i>n</i> Cauchy distributed random numbers
	DOUBLE PRECISION, INTENT (OUT) for vdrngcauchy	
	C: float* for vsRngCauchy double* for vdRngCauchy	

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> n_{max}$.
VSL_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Rayleigh

Generates Rayleigh distributed random values.

Syntax

Fortran:

```
status = vsrngrayleigh( method, stream, n, r, a, beta )
```

```
status = vdrnggrayleigh( method, stream, n, r, a, beta )
```

C:

```
status = vsRngRayleigh( method, stream, n, r, a, beta );
```

```
status = vdRngRayleigh( method, stream, n, r, a, beta );
```

Description

This function generates Rayleigh distributed random numbers with displacement a and scalefactor β , where $a, \beta \in \mathbb{R}$; $\beta > 0$.

Rayleigh distribution is a special case of [Weibull](#) distribution, where the shape parameter $\alpha = 2$.

The probability density function is given by:

$$f_{a,\beta}(x) = \begin{cases} \frac{2(x-a)}{\beta^2} \exp\left(-\frac{(x-a)^2}{\beta^2}\right), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \begin{cases} 1 - \exp\left(-\frac{(x-a)^2}{\beta^2}\right), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

Input Parameters

Name	Type	Description
<i>method</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Generation method. The specific values are as follows: VSL_METHOD_SRAYLEIGH_ICDF VSL_METHOD_DRAYLEIGH_ICDF VSL_METHOD_SRAYLEIGH_ICDF_ACCURATE VSL_METHOD_DRAYLEIGH_ICDF_ACCURATE Inverse cumulative distribution function method
<i>stream</i>	FORTRAN: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	FORTRAN: Descriptor of the stream state structure. C: Pointer to the stream state structure
<i>n</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Number of random values to be generated
<i>a</i>	FORTRAN: REAL, INTENT (IN) for vsrnggrayleigh DOUBLE PRECISION, INTENT (IN) for vdrnggrayleigh	Displacement <i>a</i>

Name	Type	Description
	C: float for vsRngRayleigh double for vdRngRayleigh	
<i>beta</i>	FORTTRAN: REAL, INTENT(IN) for vsrnggrayleigh DOUBLE PRECISION, INTENT(IN) for vdrnggrayleigh C: float for vsRngRayleigh double for vdRngRayleigh	Scalefactor β .

Output Parameters

Name	Type	Description
<i>r</i>	FORTTRAN: REAL, INTENT(OUT) for vsrnggrayleigh DOUBLE PRECISION, INTENT(OUT) for vdrnggrayleigh C: float* for vsRngRayleigh double* for vdRngRayleigh	Vector of n Rayleigh distributed random numbers

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.

VSL_ERROR_BAD_UPDATE

Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> n_{\max}$.

VSL_ERROR_NO_NUMBERS

Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Lognormal

Generates lognormally distributed random numbers.

Syntax

Fortran:

```
status = vsrnglognormal( method, stream, n, r, a, sigma, b, beta )
```

```
status = vdrnglognormal( method, stream, n, r, a, sigma, b, beta )
```

C:

```
status = vsRngLognormal( method, stream, n, r, a, sigma, b, beta );
```

```
status = vdRngLognormal( method, stream, n, r, a, sigma, b, beta );
```

Description

This function generates lognormally distributed random numbers with average of distribution a and standard deviation σ of subject normal distribution, displacement b , and scalefactor β , where $a, \sigma, b, \beta \in \mathbb{R}$; $\sigma > 0$, $\beta > 0$.

The probability density function is given by:

$$f_{a,\sigma,b,\beta}(x) = \begin{cases} \frac{1}{\sigma(x-b)\sqrt{2\pi}} \exp\left(-\frac{[\ln((x-b)/\beta) - a]^2}{2\sigma^2}\right), & x > b \\ 0, & x \leq b \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,\sigma,b,\beta}(x) = \begin{cases} \Phi((\ln((x-b)/\beta) - a)/\sigma), & x > b \\ 0, & x \leq b \end{cases}$$

Input Parameters

Name	Type	Description
<i>method</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Generation method. The specific values are as follows: VSL_METHOD_SLOGNORMAL_BOXMULLER2 VSL_METHOD_DLOGNORMAL_BOXMULLER2 VSL_METHOD_SLOGNORMAL_BOXMULLER2_ACCURATE VSL_METHOD_DLOGNORMAL_BOXMULLER2_ACCURATE Inverse cumulative distribution function method
<i>stream</i>	FORTTRAN: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	FORTTRAN: Descriptor of the stream state structure. C: Pointer to the stream state structure
<i>n</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Number of random values to be generated
<i>a</i>	FORTTRAN: REAL, INTENT (IN) for vsrnglognormal DOUBLE PRECISION, INTENT (IN) for vdrnglognormal C: float for vsRngLognormal	Average <i>a</i> of the subject normal distribution

Name	Type	Description
	double for vdRngLognormal	
<i>sigma</i>	FORTTRAN: REAL, INTENT (IN) for vsrnglognormal DOUBLE PRECISION, INTENT (IN) for vdrnglognormal C: float for vsRngLognormal double for vdRngLognormal	Standard deviation σ of the subject normal distribution
<i>b</i>	FORTTRAN: REAL, INTENT (IN) for vsrnglognormal DOUBLE PRECISION, INTENT (IN) for vdrnglognormal C: float for vsRngLognormal double for vdRngLognormal	Displacement b
<i>beta</i>	FORTTRAN: REAL, INTENT (IN) for vsrnglognormal DOUBLE PRECISION, INTENT (IN) for vdrnglognormal C: float for vsRngLognormal double for vdRngLognormal	Scalefactor β .

Output Parameters

Name	Type	Description
<i>r</i>	FORTTRAN: REAL, INTENT (OUT) for vsrnglognormal DOUBLE PRECISION, INTENT (OUT) for vdrnglognormal C: float* for vsRngLognormal double* for vdRngLognormal	Vector of <i>n</i> lognormally distributed random numbers

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> n_{\max}$.
VSL_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Gumbel

Generates Gumbel distributed random values.

Syntax

Fortran:

```
status = vsrnggumbel( method, stream, n, r, a, beta )
status = vdrnggumbel( method, stream, n, r, a, beta )
```

C:

```
status = vsRngGumbel( method, stream, n, r, a, beta );
status = vdRngGumbel( method, stream, n, r, a, beta );
```

Description

This function generates Gumbel distributed random numbers with displacement a and scalefactor β , where $a, \beta \in R ; \beta > 0$.

The probability density function is given by:

$$f_{a,\beta}(x) = \left\{ \frac{1}{\beta} \exp \left(\frac{x - a}{\beta} \right) \exp \left(- \exp \left((x - a) / \beta \right) \right), -\infty < x < +\infty . \right.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = 1 - \exp \left(- \exp \left((x - a) / \beta \right) \right), -\infty < x < +\infty$$

Input Parameters

Name	Type	Description
<i>method</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Generation method. The specific values are as follows: VSL_METHOD_SGUMBEL_ICDF VSL_METHOD_DGUMBEL_ICDF Inverse cumulative distribution function method
<i>stream</i>	FORTRAN: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	FORTRAN: Descriptor of the stream state structure C: Pointer to the stream state structure

Name	Type	Description
<i>n</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Number of random values to be generated
<i>a</i>	FORTTRAN: REAL, INTENT (IN) for vsrnggumbel DOUBLE PRECISION, INTENT (IN) for vdrnggumbel C: float for vsRngGumbel double for vdRngGumbel	Displacement <i>a</i> .
<i>beta</i>	FORTTRAN: REAL, INTENT (IN) for vsrnggumbel DOUBLE PRECISION, INTENT (IN) for vdrnggumbel C: float for vsRngGumbel double for vdRngGumbel	Scalefactor β .

Output Parameters

Name	Type	Description
<i>r</i>	FORTTRAN: REAL, INTENT (OUT) for vsrnggumbel DOUBLE PRECISION, INTENT (OUT) for vdrnggumbel C: float* for vsRngGumbel	Vector of <i>n</i> random numbers with Gumbel distribution

Name	Type	Description
	double* for vdRngGumbel	
Return Values		
VSL_ERROR_OK, VSL_STATUS_OK		Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR		<i>stream</i> is a NULL pointer.
VSL_ERROR_BAD_STREAM		<i>stream</i> is not a valid random stream.
VSL_ERROR_BAD_UPDATE		Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> n_{\max}$.
VSL_ERROR_NO_NUMBERS		Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Gamma

Generates gamma distributed random values.

Syntax

Fortran:

```
status = vsrnggamma( method, stream, n, r, alpha, a, beta )
status = vdrnggamma( method, stream, n, r, alpha, a, beta )
```

C:

```
status = vsRngGamma( method, stream, n, r, alpha, a, beta );
status = vdRngGamma( method, stream, n, r, alpha, a, beta );
```

Description

This function generates random numbers with gamma distribution that has shape parameter α , displacement a , and scale parameter β , where α, β , and $a \in \mathbb{R}$; $\alpha > 0$, $\beta > 0$.

The probability density function is given by:

$$f_{\alpha, a, \beta}(x) = \begin{cases} \frac{1}{\Gamma(\alpha)\beta^\alpha} (x - a)^{\alpha-1} e^{-(x-a)/\beta}, & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty$$

where $\Gamma(\alpha)$ is the complete gamma function.

The cumulative distribution function is as follows:

$$F_{\alpha, a, \beta}(x) = \begin{cases} \int_a^x \frac{1}{\Gamma(\alpha)\beta^\alpha} (y - a)^{\alpha-1} e^{-(y-a)/\beta} dy, & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty$$

Input Parameters

Name	Type	Description
<i>method</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Generation method. The specific values are as follows: VSL_METHOD_SGAMMA_GNORM VSL_METHOD_DGAMMA_GNORM VSL_METHOD_SGAMMA_GNORM_ACCURATE VSL_METHOD_DGAMMA_GNORM_ACCURATE Acceptance/rejection method using random numbers with Gaussian distribution. See brief description of the method GNORM in Table 10-1
<i>stream</i>	FORTRAN: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	FORTRAN: Descriptor of the stream state structure C: Pointer to the stream state structure
<i>n</i>	FORTRAN: INTEGER, INTENT (IN)	Number of random values to be generated

Name	Type	Description
	C: int	
<i>alpha</i>	FORTTRAN: REAL, INTENT (IN) for vsrnggamma DOUBLE PRECISION, INTENT (IN) for vdrnggamma C: float for vsRngGamma double for vdRngGamma	Shape α .
<i>a</i>	FORTTRAN: REAL, INTENT (IN) for vsrnggamma DOUBLE PRECISION, INTENT (IN) for vdrnggamma C: float for vsRngGamma double for vdRngGamma	Displacement a .
<i>beta</i>	FORTTRAN: REAL, INTENT (IN) for vsrnggamma DOUBLE PRECISION, INTENT (IN) for vdrnggamma C: float for vsRngGamma double for vdRngGamma	Scalefactor β .

Output Parameters

Name	Type	Description
<i>r</i>	FORTTRAN: REAL, INTENT (OUT) for vsrnggamma DOUBLE PRECISION, INTENT (OUT) for vdrnggamma	Vector of n random numbers with gamma distribution

Name	Type	Description

	C: float* for vsRngGamma	

	double* for vdRngGamma	

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	

Indicates no error, execution is successful.

VSL_ERROR_NULL_PTR	

stream is a NULL pointer.

VSL_ERROR_BAD_STREAM	

stream is not a valid random stream.

VSL_ERROR_BAD_UPDATE	

Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> n_{\max}$.

VSL_ERROR_NO_NUMBERS	

Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Beta

Generates beta distributed random values.

Syntax

Fortran:

```
status = vsrngbeta( method, stream, n, r, p, q, a, beta )
```

```
status = vdrngbeta( method, stream, n, r, p, q, a, beta )
```

C:

```
status = vsRngBeta( method, stream, n, r, p, q, a, beta );
```

```
status = vdRngBeta( method, stream, n, r, p, q, a, beta );
```

Description

This function generates random numbers with beta distribution that has shape parameters p and q , displacement a , and scale parameter β , where p, q, a , and $\beta \in \mathbb{R}$; $p > 0$, $q > 0$, $\beta > 0$.

The probability density function is given by:

$$f_{p,q,a,\beta}(x) = \begin{cases} \frac{1}{B(p,q)\beta^{p+q-1}} (x-a)^{p-1} (\beta+a-x)^{q-1}, & a \leq x < a+\beta \\ 0, & x < a, x \geq a+\beta \end{cases}, -\infty < x < \infty,$$

where $B(p, q)$ is the complete beta function.
The cumulative distribution function is as follows:

$$F_{p,q,a,\beta}(x) = \begin{cases} 0, & x < a \\ \int_a^x \frac{1}{B(p,q)\beta^{p+q-1}} (y-a)^{p-1} (\beta+a-y)^{q-1} dy, & a \leq x < a+\beta \\ 1, & x \geq a+\beta \end{cases}, -\infty < x < \infty.$$

Input Parameters

Name	Type	Description
<i>method</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Generation method. The specific values are as follows: VSL_METHOD_SBETA_CJA VSL_METHOD_DBETA_CJA VSL_METHOD_SBETA_CJA_ACCURATE VSL_METHOD_DBETA_CJA_ACCURATE See brief description of the method CJA in Table 10-1
<i>stream</i>	FORTRAN: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	FORTRAN: Descriptor of the stream state structure C: Pointer to the stream state structure
<i>n</i>	FORTRAN: INTEGER, INTENT (IN)	Number of random values to be generated

Name	Type	Description
	C: int	
p	FORTTRAN: REAL, INTENT (IN) for vsrngbeta DOUBLE PRECISION, INTENT (IN) for vdrngbeta C: float for vsRngBeta double for vdRngBeta	Shape p
q	FORTTRAN: REAL, INTENT (IN) for vsrngbeta DOUBLE PRECISION, INTENT (IN) for vdrngbeta C: float for vsRngBeta double for vdRngBeta	Shape q
a	FORTTRAN: REAL, INTENT (IN) for vsrngbeta DOUBLE PRECISION, INTENT (IN) for vdrngbeta C: float for vsRngBeta double for vdRngBeta	Displacement a .
β	FORTTRAN: REAL, INTENT (IN) for vsrngbeta DOUBLE PRECISION, INTENT (IN) for vdrngbeta C: float for vsRngBeta double for vdRngBeta	Scalefactor β .

Output Parameters

Name	Type	Description
r	FORTTRAN: REAL, INTENT (OUT) for vsrngbeta DOUBLE PRECISION, INTENT (OUT) for vdrngbeta C: float* for vsRngBeta double* for vdRngBeta	Vector of n random numbers with beta distribution

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	$stream$ is a NULL pointer.
VSL_ERROR_BAD_STREAM	$stream$ is not a valid random stream.
VSL_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.
VSL_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Discrete Distributions

This section describes routines for generating random numbers with discrete distribution.

Uniform

Generates random numbers uniformly distributed over the interval $[a, b)$.

Syntax

Fortran:

`status = virnguniform( method, stream, n, r, a, b )`

C:

```
status = viRngUniform( method, stream, n, r, a, b );
```

Description

This function generates random numbers uniformly distributed over the interval $[a, b)$, where a, b are the left and right bounds of the interval respectively, and $a, b \in \mathbb{Z}; a < b$.

The probability distribution is given by:

$$P(X = k) = \frac{1}{b - a}, k \in \{a, a + 1, \dots, b - 1\}.$$

The cumulative distribution function is as follows:

$$F_{a,b}(x) = \begin{cases} 0, & x < a \\ \frac{\lfloor x - a + 1 \rfloor}{b - a}, & a \leq x < b, x \in \mathbb{R}. \\ 1, & x \geq b \end{cases}$$

Input Parameters

Name	Type	Description
<i>method</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Generation method; dummy and set to 0 in case of uniform distribution. The specific value is as follows: VSL_METHOD_IUNIFORM_STD Standard method. Currently there is only one method for this distribution generator.

Name	Type	Description
<i>stream</i>	FORTRAN: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	FORTRAN: Descriptor of the stream state structure. C: Pointer to the stream state structure
<i>n</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Number of random values to be generated
<i>a</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Left interval bound <i>a</i>
<i>b</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Right interval bound <i>b</i>

Output Parameters

Name	Type	Description
<i>r</i>	FORTRAN: INTEGER, INTENT (OUT) C: int*	Vector of <i>n</i> random numbers uniformly distributed over the interval $[a, b)$

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> n_{\max}$.

VSL_ERROR_NO_NUMBERS

Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

UniformBits

Generates integer random values with uniform bit distribution.

Syntax

Fortran:

```
status = virnguniformbits( method, stream, n, r )
```

C:

```
status = viRngUniformBits( method, stream, n, r );
```

Description

This function generates integer random values with uniform bit distribution. The generators of uniformly distributed numbers can be represented as recurrence relations over integer values in modular arithmetic. Apparently, each integer can be treated as a vector of several bits. In a truly random generator, these bits are random, while in pseudorandom generators this randomness can be violated. For example, a well known drawback of linear congruential generators is that lower bits are less random than higher bits (for example, see [Knuth81]). For this reason, care should be taken when using this function. Typically, in a 32-bit *LCG* only 24 higher bits of an integer value can be considered random. See [VSL Notes](#) for details.

Input Parameters

Name	Type	Description
<i>method</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Generation method; dummy and set to 0. The specific value is as follows: VSL_METHOD_IUNIFORMBITS_STD Standard method. Currently there is only one method for this distribution generator.
<i>stream</i>	FORTTRAN: TYPE (VSL_STREAM_STATE), INTENT (IN)	FORTTRAN: Descriptor of the stream state structure. C: Pointer to the stream state structure

Name	Type	Description
	C: VSLStreamStatePtr	
<i>n</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Number of random values to be generated

Output Parameters

Name	Type	Description
<i>r</i>	FORTRAN: INTEGER, INTENT (OUT) C: unsigned int*	FORTRAN: Vector of <i>n</i> random integer numbers. If the <i>stream</i> was generated by a 64 or a 128-bit generator, each integer value is represented by two or four elements of <i>r</i> respectively. The number of bytes occupied by each integer is contained in the field <i>wordsize</i> of the structure VSL_BRNG_PROPERTIES. The total number of bits that are actually used to store the value are contained in the field <i>nbits</i> of the same structure. See Advanced Service Routines for a more detailed discussion of VSLBRngProperties. C: Vector of <i>n</i> random integer numbers. If the <i>stream</i> was generated by a 64 or a 128-bit generator, each integer value is represented by two or four elements of <i>r</i> respectively. The number of bytes occupied by each integer is contained in the field <i>WordSize</i> of the structure VSLBRngProperties. The total number of bits that are actually used to store the value are contained in the field <i>NBits</i> of the same structure. See Advanced Service Routines for a more detailed discussion of VSLBRngProperties.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.

VSL_ERROR_NULL_PTR
VSL_ERROR_BAD_STREAM
VSL_ERROR_BAD_UPDATE

VSL_ERROR_NO_NUMBERS

stream is a NULL pointer.

stream is not a valid random stream.

Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> n_{\max}$.

Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Bernoulli

Generates Bernoulli distributed random values.

Syntax

Fortran:

```
status = virngbernoulli( method, stream, n, r, p )
```

C:

```
status = viRngBernoulli( method, stream, n, r, p );
```

Description

This function generates Bernoulli distributed random numbers with probability p of a single trial success, where

$p \in \mathbb{R}; 0 \leq p \leq 1$.

A variate is called Bernoulli distributed, if after a trial it is equal to 1 with probability of success p , and to 0 with probability $1 - p$.

The probability distribution is given by:

$$P(X = 1) = p$$

$$P(X = 0) = 1 - p$$

The cumulative distribution function is as follows:

$$F_p(x) = \begin{cases} 0, & x < 0 \\ 1 - p, & 0 \leq x < 1, x \in R. \\ 1, & x \geq 1 \end{cases}$$

Input Parameters

Name	Type	Description
<i>method</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Generation method. The specific value is as follows: VSL_METHOD_IBERNOULLI_ICDF Inverse cumulative distribution function method.
<i>stream</i>	FORTRAN: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	FORTRAN: Descriptor of the stream state structure. C: Pointer to the stream state structure
<i>n</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Number of random values to be generated
<i>p</i>	FORTRAN: DOUBLE PRECISION, INTENT (IN) C: double	Success probability <i>p</i> of a trial

Output Parameters

Name	Type	Description
<i>r</i>	FORTRAN: INTEGER, INTENT (OUT) C: int*	Vector of <i>n</i> Bernoulli distributed random values

Return Values

VSL_ERROR_OK, VSL_STATUS_OK

Indicates no error, execution is successful.

VSL_ERROR_NULL_PTR

stream is a NULL pointer.

VSL_ERROR_BAD_STREAM

stream is not a valid random stream.

VSL_ERROR_BAD_UPDATE

Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> n_{\max}$.

VSL_ERROR_NO_NUMBERS

Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Geometric

Generates geometrically distributed random values.

Syntax

Fortran:

```
status = virnggeometric( method, stream, n, r, p )
```

C:

```
status = viRngGeometric( method, stream, n, r, p );
```

Description

This function generates geometrically distributed random numbers with probability p of a single trial success, where $p \in \mathbb{R}$; $0 < p < 1$.

A geometrically distributed variate represents the number of independent Bernoulli trials preceding the first success. The probability of a single Bernoulli trial success is p .

The probability distribution is given by:

$$P(X = k) = p \cdot (1 - p)^k, \quad k \in \{0, 1, 2, \dots\}.$$

The cumulative distribution function is as follows:

$$F_p(x) = \begin{cases} 0, & x < 0 \\ 1 - (1 - p)^{\lfloor x+1 \rfloor}, & 0 \leq x \end{cases} \quad x \in \mathbb{R}.$$

Input Parameters

Name	Type	Description
<i>method</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Generation method. The specific value is as follows: VSL_METHOD_IGEOMETRIC_ICDF Inverse cumulative distribution function method.
<i>stream</i>	FORTRAN: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	FORTRAN: Descriptor of the stream state structure. C: Pointer to the stream state structure
<i>n</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Number of random values to be generated
<i>p</i>	FORTRAN: DOUBLE PRECISION, INTENT (IN) C: double	Success probability <i>p</i> of a trial

Output Parameters

Name	Type	Description
<i>r</i>	FORTRAN: INTEGER, INTENT (OUT) C: int*	Vector of <i>n</i> geometrically distributed random values

Return Values

VSL_ERROR_OK, VSL_STATUS_OK

Indicates no error, execution is successful.

VSL_ERROR_NULL_PTR

stream is a NULL pointer.

VSL_ERROR_BAD_STREAM

stream is not a valid random stream.

VSL_ERROR_BAD_UPDATE

Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> n_{\max}$.

VSL_ERROR_NO_NUMBERS

Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Binomial

Generates binomially distributed random numbers.

Syntax

Fortran:

```
status = virngbinomial( method, stream, n, r, ntrial, p )
```

C:

```
status = viRngBinomial( method, stream, n, r, ntrial, p );
```

Description

This function generates binomially distributed random numbers with number of independent Bernoulli trials m , and with probability p of a single trial success, where $p \in R$; $0 \leq p \leq 1$, $m \in N$.

A binomially distributed variate represents the number of successes in m independent Bernoulli trials with probability of a single trial success p .

The probability distribution is given by:

$$P(X = k) = C_m^k p^k (1 - p)^{m-k}, k \in \{0, 1, \dots, m\}.$$

The cumulative distribution function is as follows:

$$F_{m,p}(x) = \begin{cases} 0, & x < 0 \\ \sum_{k=0}^{\lfloor x \rfloor} C_m^k p^k (1-p)^{m-k}, & 0 \leq x < m, x \in R \\ 1, & x \geq m \end{cases}$$

Input Parameters

Name	Type	Description
<i>method</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Generation method. The specific value is as follows: VSL_METHOD_IBINOMIAL_BTPE See brief description of the BTPE method in Table 10-1 .
<i>stream</i>	FORTRAN: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	FORTRAN: Descriptor of the stream state structure. C: Pointer to the stream state structure
<i>n</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Number of random values to be generated
<i>ntrials</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Number of independent trials m
<i>p</i>	FORTRAN: DOUBLE PRECISION, INTENT (IN) C: double	Success probability p of a single trial

Output Parameters

Name	Type	Description
<i>r</i>	FORTRAN: INTEGER, INTENT (OUT) C: int*	Vector of <i>n</i> binomially distributed random values

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.
VSL_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Hypergeometric

Generates hypergeometrically distributed random values.

Syntax

Fortran:

```
status = virnghypergeometric( method, stream, n, r, l, s, m )
```

C:

```
status = viRngHypergeometric( method, stream, n, r, l, s, m );
```

Description

This function generates hypergeometrically distributed random values with lot size *l*, size of sampling *s*, and number of marked elements in the lot *m*, where $l, m, s \in \mathbb{N} \setminus \{0\}$; $l \geq \max(s, m)$.

Consider a lot of 1 elements comprising m “marked” and $1-m$ “unmarked” elements. A trial sampling without replacement of exactly s elements from this lot helps to define the hypergeometric distribution, which is the probability that the group of s elements contains exactly k marked elements.

The probability distribution is given by:)

$$P(X = k) = \frac{C_m^k C_{1-m}^{s-k}}{C_1^s}$$

, $k \in \{\max(0, s + m - 1), \dots, \min(s, m)\}$

The cumulative distribution function is as follows:

$$F_{1,s,m}(x) = \begin{cases} 0, & x < \max(0, s + m - 1) \\ \sum_{k=\max(0,s+m-1)}^{\lfloor x \rfloor} \frac{C_m^k C_{1-m}^{s-k}}{C_1^s}, & \max(0, s + m - 1) \leq x \leq \min(s, m) \\ 1, & x > \min(s, m) \end{cases}$$

Input Parameters

Name	Type	Description
<i>method</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Generation method. The specific value is as follows: VSL_METHOD_IHYPERGEOMETRIC_H2PE See brief description of the H2PE method in Table 10-1
<i>stream</i>	FORTRAN: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	FORTRAN: Descriptor of the stream state structure. C: Pointer to the stream state structure

Name	Type	Description
<i>n</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Number of random values to be generated
<i>l</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Lot size <i>l</i>
<i>s</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Size of sampling without replacement <i>s</i>
<i>m</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Number of marked elements <i>m</i>

Output Parameters

Name	Type	Description
<i>r</i>	FORTTRAN: INTEGER, INTENT (OUT) C: int*	Vector of <i>n</i> hypergeometrically distributed random values

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, <i>< 0</i> or <i>> nmax</i> .
VSL_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Poisson

Generates Poisson distributed random values.

Syntax

Fortran:

```
status = virngpoisson( method, stream, n, r, lambda )
```

C:

```
status = viRngPoisson( method, stream, n, r, lambda );
```

Description

This function generates Poisson distributed random numbers with distribution parameter λ , where $\lambda \in R$; $\lambda > 0$.

The probability distribution is given by:

$$P(X = k) = \frac{\lambda^k e^{-\lambda}}{k!},$$

$k \in \{0, 1, 2, \dots\}$.

The cumulative distribution function is as follows:

$$F_{\lambda}(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} \frac{\lambda^k e^{-\lambda}}{k!}, & x \geq 0 \\ 0, & x < 0 \end{cases}, x \in R$$

Input Parameters

Name	Type	Description
<i>method</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Generation method. The specific values are as follows: VSL_METHOD_IPOISSON_PTPE VSL_METHOD_IPOISSON_POISNORM See brief description of the PTPE and POISNORM methods in Table 10-1 .
<i>stream</i>	FORTTRAN: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	FORTTRAN: Descriptor of the stream state structure. C: Pointer to the stream state structure
<i>n</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Number of random values to be generated
<i>lambda</i>	FORTTRAN: DOUBLE PRECISION, INTENT (IN) C: double	Distribution parameter λ .

Output Parameters

Name	Type	Description
<i>r</i>	FORTTRAN: INTEGER, INTENT (OUT) C: int*	Vector of <i>n</i> Poisson distributed random values

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.

VSL_ERROR_BAD_STREAM

stream is not a valid random stream.

VSL_ERROR_BAD_UPDATE

Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> n_{\max}$.

VSL_ERROR_NO_NUMBERS

Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

PoissonV

Generates Poisson distributed random values with varying mean.

Syntax

Fortran:

```
status = virngpoissonv( method, stream, n, r, lambda )
```

C:

```
status = viRngPoissonV( method, stream, n, r, lambda );
```

Description

This function generates n Poisson distributed random numbers $x_i (i = 1, \dots, n)$ with distribution parameter λ_i , where $\lambda_i \in \mathbb{R}; \lambda_i > 0$.

The probability distribution is given by:

$$P(X_i = k) = \frac{\lambda_i^k \exp(-\lambda_i)}{k!}, k \in \{0, 1, 2, \dots\}.$$

The cumulative distribution function is as follows:

$$F_{\lambda_i}(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} \frac{\lambda_i^k e^{-\lambda_i}}{k!}, & x \geq 0 \\ 0, & x < 0 \end{cases}, x \in R$$

Input Parameters

Name	Type	Description
<i>method</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Generation method. The specific value is as follows: VSL_METHOD_IPOISSONV_POISNORM See brief description of the POISNORM method in Table 10-1
<i>stream</i>	FORTTRAN: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	FORTTRAN: Descriptor of the stream state structure. C: Pointer to the stream state structure
<i>n</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Number of random values to be generated
<i>lambda</i>	FORTTRAN: DOUBLE PRECISION, INTENT (IN) C: double*	Array of <i>n</i> distribution parameters λ_i .

Output Parameters

Name	Type	Description
<i>r</i>	FORTTRAN: INTEGER, INTENT (OUT) C: int*	Vector of <i>n</i> Poisson distributed random values

Return Values

VSL_ERROR_OK, VSL_STATUS_OK

Indicates no error, execution is successful.

VSL_ERROR_NULL_PTR

stream is a NULL pointer.

VSL_ERROR_BAD_STREAM

stream is not a valid random stream.

VSL_ERROR_BAD_UPDATE

Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> n_{\max}$.

VSL_ERROR_NO_NUMBERS

Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

NegBinomial

Generates random numbers with negative binomial distribution.

Syntax

Fortran:

```
status = virngnegbinomial( method, stream, n, r, a, p )
```

C:

```
status = viRngNegbinomial( method, stream, n, r, a, p );
```

Description

This function generates random numbers with negative binomial distribution and distribution parameters a and p , where $p, a \in \mathbb{R}$; $0 < p < 1$; $a > 0$.

If the first distribution parameter $a \in \mathbb{N}$, this distribution is the same as Pascal distribution. If $a \in \mathbb{N}$, the distribution can be interpreted as the expected time of a -th success in a sequence of Bernoulli trials, when the probability of success is p .

The probability distribution is given by:

$$P(X = k) = C_{a+k-1}^k p^a (1-p)^k, k \in \{0, 1, 2, \dots\}.$$

The cumulative distribution function is as follows:

$$F_{a,p}(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} C_{a+k-1}^k p^a (1-p)^k, & x \geq 0 \\ 0, & x < 0 \end{cases}, x \in R$$

Input Parameters

Name	Type	Description
<i>method</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Generation method. The specific value is: VSL_METHOD_INEGBINOMIAL_NBAR See brief description of the NBAR method in Table 10-1
<i>stream</i>	FORTTRAN: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	FORTTRAN: descriptor of the stream state structure. C: pointer to the stream state structure
<i>n</i>	FORTTRAN: INTEGER, INTENT (IN) C: int	Number of random values to be generated
<i>a</i>	FORTTRAN: DOUBLE PRECISION, INTENT (IN) C: double	The first distribution parameter <i>a</i>
<i>p</i>	FORTTRAN: DOUBLE PRECISION, INTENT (IN) C: double	The second distribution parameter <i>p</i>

Output Parameters

Name	Type	Description
r	FORTRAN: INTEGER, INTENT (OUT) C: int*	Vector of n random values with negative binomial distribution.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> n_{\max}$.
VSL_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Advanced Service Routines

This section describes service routines for registering a user-designed basic generator ([RegisterBrng](#)) and for obtaining properties of the previously registered basic generators ([GetBrngProperties](#)). See [VSL Notes](#) ("Basic Generators" section of VSL Structure chapter) for substantiation of the need for several basic generators including user-defined BRNGs.

Data types

The Advanced Service routines refer to a structure defining the properties of the basic generator. This structure is described in Fortran as follows:

```

TYPE VSL_BRNG_PROPERTIES
    INTEGER streamstatesize
    INTEGER nseeds
    INTEGER includeszero
    INTEGER wordsize
    INTEGER nbits
    INTEGER nitstream
    INTEGER sbrng
    INTEGER dbrng
    INTEGER ibrng
END TYPE VSL_BRNG_PROPERTIES
    
```

The C version is as follows:

```

typedef struct _VSLBRngProperties {
    int StreamStateSize;
    int NSeeds;
    int IncludesZero;
    int WordSize;
    int NBits;
    InitStreamPtr InitStream;
    sBRngPtr sBRng;
    dBRngPtr dBRng;
    iBRngPtr iBRng;
} VSLBRngProperties;
    
```

The following table provides brief descriptions of the fields engaged in the above structure:

Table 10-13 Field Descriptions

Field	Short Description
FORTTRAN: streamstatesize C: StreamStateSize	The size, in bytes, of the stream state structure for a given basic generator.
FORTTRAN: nseeds C: NSeeds	The number of 32-bit initial conditions (seeds) necessary to initialize the stream state structure for a given basic generator.
FORTTRAN: includeszero C: IncludesZero	Flag value indicating whether the generator can produce a random 0.
FORTTRAN: wordsize C: WordSize	Machine word size, in bytes, used in integer-value computations. Possible values: 4, 8, and 16 for 32, 64, and 128-bit generators, respectively.
FORTTRAN: nbits C: NBits	The number of bits required to represent a random value in integer arithmetic. Note that, for instance, 48-bit random values are stored to 64-bit (8 byte) memory locations. In this case, wordsize/WordSize is equal to 8 (number of bytes used to store the random value), while nbits/NBits contains the actual number of bits occupied by the value (in this example, 48).
FORTTRAN: initstream C: InitStream	Contains the pointer to the initialization routine of a given basic generator.
FORTTRAN: sbrng C: sBRng	Contains the pointer to the basic generator of single precision real numbers uniformly distributed over the interval (<i>a</i> , <i>b</i>) (real in FORTRAN and float in C).
FORTTRAN: dbrng C: dBRng	Contains the pointer to the basic generator of double precision real numbers uniformly distributed over the interval (<i>a</i> , <i>b</i>) (double PRECISION in FORTRAN and double in C).
FORTTRAN: ibrng C: iBRng	Contains the pointer to the basic generator of integer numbers with uniform bit distribution ¹ (INTEGER in FORTRAN and unsigned int in C).

¹A specific generator that permits operations over single bits and bit groups of random numbers.

RegisterBrng

Registers user-defined basic generator.

Syntax

Fortran:

```
brng = vslregisterbrng( properties )
```

C:

```
brng = vslRegisterBrng( &properties );
```

Description

An example of a registration procedure can be found in the respective directory of VSL examples.

Input Parameters

Name	Type	Description
<i>properties</i>	FORTTRAN: TYPE (VSL_BRNG_PROPERTIES), INTENT (IN) C: VSLBRngProperties*	Pointer to the structure containing properties of the basic generator to be registered

Output Parameters

Name	Type	Description
<i>brng</i>	FORTTRAN: INTEGER, INTENT (OUT) C: int	Number (index) of the registered basic generator; used for identification. Negative values indicate the registration error.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_BRNG_TABLE_FULL	Registration cannot be completed due to lack of free entries in the table of registered BRNGs.
VSL_ERROR_BAD_STREAM_STATE_SIZE	Bad value in StreamStateSize field.

VSL_ERROR_BAD_WORD_SIZE	Bad value in <code>WordSize</code> field.
VSL_ERROR_BAD_NSEEDS	Bad value in <code>NSeeds</code> field.
VSL_ERROR_BAD_NBITS	Bad value in <code>NBits</code> field.
VSL_ERROR_NULL_PTR	At least one of the fields <code>iBrng</code> , <code>dBrng</code> , <code>sBrng</code> or <code>InitStream</code> is a <code>NULL</code> pointer.

GetBrngProperties

Returns structure with properties of a given basic generator.

Syntax

Fortran:

```
status = vslgetbrngproperties( brng, properties )
```

C:

```
status = vslGetBrngProperties( brng, &properties );
```

Input Parameters

Name	Type	Description
<i>brng</i>	FORTRAN: INTEGER, INTENT (IN) C: int	Number (index) of the registered basic generator; used for identification. See specific values in Table 10-2 . Negative values indicate the registration error.

Output Parameters

Name	Type	Description
<i>properties</i>	FORTRAN: TYPE (VSL_BRNG_PROPERTIES), INTENT (OUT) C: VSLBRngProperties*	Pointer to the structure containing properties of the generator with number <i>brng</i>

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_INVALID_BRNG_INDEX	BRNG index is invalid.

Formats for User-Designed Generators

To register a user-designed basic generator using [RegisterBrng](#) function, you need to pass the pointer `iBrng` to the integer-value implementation of the generator; the pointers `sBrng` and `dBrng` to the generator implementations for single and double precision values, respectively; and pass the pointer `InitStream` to the stream initialization routine. See below recommendations on defining such functions with input and output arguments. An example of the registration procedure for a user-designed generator can be found in the respective directory of VSL examples.

The respective pointers are defined as follows:

```
typedef int(*InitStreamPtr)( int method, VSLStreamStatePtr stream, int n, const unsigned
int params[] );

typedef int(*sBrngPtr)( VSLStreamStatePtr stream, int n, float r[], float a, float b );

typedef int(*dBrngPtr)( VSLStreamStatePtr stream, int n, double r[], double a, double b );

typedef int(*iBrngPtr)( VSLStreamStatePtr stream, int n, unsigned int r[] );
```

InitStream

C:

```
int MyBrngInitStream( int method, VSLStreamStatePtr stream, int n, const unsigned int
params[] )
{
    /* Initialize the stream */

    ...

} /* MyBrngInitStream */
```

Description

The initialization routine of a user-designed generator must initialize *stream* according to the specified initialization *method*, initial conditions *params* and the argument *n*. The value of *method* determines the initialization method to be used.

- If *method* is equal to 1, the initialization is by the standard generation method, which must be supported by all basic generators. In this case the function assumes that the *stream* structure was not previously initialized. The value of *n* is used as the actual number of 32-bit values passed as initial conditions through *params*. Note, that the situation when the actual number of initial conditions passed to the function is not sufficient to initialize the generator is not an error. Whenever it occurs, the basic generator must initialize the missing conditions using default settings.
- If *method* is equal to 2, the generation is by the leapfrog method, where *n* specifies the number of computational nodes (independent streams). Here the function assumes that the *stream* was previously initialized by the standard generation method. In this case *params* contains only one element, which identifies the computational node. If the generator does not support the leapfrog method, the function must return the error code `VSL_ERROR_LEAPFROG_UNSUPPORTED`.
- If *method* is equal to 3, the generation is by the block-splitting method. Same as above, the *stream* is assumed to be previously initialized by the standard generation method; *params* is not used, *n* identifies the number of skipped elements. If the generator does not support the block-splitting method, the function must return the error code `VSL_ERROR_SKIPAHEAD_UNSUPPORTED`.

For a more detailed description of the leapfrog and the block-splitting methods, refer to the description of [LeapfrogStream](#) and [SkipAheadStream](#), respectively.

Stream state structure is individual for every generator. However, each structure has a number of fields that are the same for all the generators:

C:

```
typedef struct
{
    unsigned int Reserved1[2];
    unsigned int Reserved2[2];
    [fields specific for the given generator]
} MyStreamState;
```

The fields *Reserved1* and *Reserved2* are reserved for private needs only, and must not be modified by the user. When including specific fields into the structure, follow the rules below:

- The fields must fully describe the current state of the generator. For example, the state of a linear congruential generator can be identified by only one initial condition;

- If the generator can use both the leapfrog and the block-splitting methods, additional fields should be introduced to identify the independent streams. For example, in $LCG(a, c, m)$, apart from the initial conditions, two more fields should be specified: the value of the multiplier a^k and the value of the increment $(a^k - 1) c / (a - 1)$.

For a more detailed discussion, refer to [Knuth81], and [Gentle98]. An example of the registration procedure can be found in the respective directory of VSL examples.

iBrng

C:

```
int iMyBrng( VSLStreamStatePtr stream, int n, unsigned int r[] )
{
    int i; /* Loop variable */
    /* Generating integer random numbers */
    /* Pay attention to word size needed to
       store only random number */
    for( i = 0; i < n; i++)
    {
        r[i] = ...;
    }
    /* Update stream state */
    ...
    return errcode;
} /* iMyBrng */
```



NOTE. When using 64 and 128-bit generators, consider digit capacity to store the numbers to the random vector r correctly. For example, storing one 64-bit value requires two elements of r , the first to store the lower 32 bits and the second to store the higher 32 bits. Similarly, use 4 elements of r to store a 128-bit value.

sBRng

C:

```
int sMyBrng( VSLStreamStatePtr stream, int n, float r[], float a, float b )
{
    int i;    /* Loop variable */
    /* Generating float (a,b) random numbers */
    for ( i = 0; i < n; i++ )
    {
        r[i] = ...;
    }
    /* Update stream state */
    ...
    return errcode;
} /* sMyBrng */
```

dBRng

C:

```
int dMyBrng( VSLStreamStatePtr stream, int n, double r[], double a, double b )
{
    int i;    /* Loop variable */
    /* Generating double (a,b) random numbers */
    for ( i = 0; i < n; i++ )
    {
        r[i] = ...;
    }
    /* Update stream state */
    ...
    return errcode;
} /* dMyBrng */
```

Convolution and Correlation

Overview

VSL provides a set of routines intended to perform linear convolution and correlation transformations for single and double precision data.

For correct definition of implemented operations, see [Mathematical Notation and Definitions](#) section.

The current implementation provides:

- Fourier algorithms for one-dimensional single and double precision data
- Fourier algorithms for multi-dimensional single and double precision data
- Direct algorithms for one-dimensional single and double precision data.
- Direct algorithms for multi-dimensional single and double precision data.

One-dimensional algorithms cover the following functions from the IBM* ESSL library:

SCONF, SCORF

SCOND, SCORD

SDCON, SDCOR

DDCON, DDCOR

SDDCON, SDDCOR.

Special wrappers are designed to simulate these ESSL functions. The wrappers are provided as sample sources for FORTRAN and C. To reuse them, use the following directories:

```
${MKL}/examples/vslc/essl/vsl_wrappers
```

```
${MKL}/examples/vslf/essl/vsl_wrappers
```

Additionally, you can browse the examples demonstrating the calculation of the ESSL functions through the wrappers. You can find the examples in the following directories:

```
${MKL}/examples/vslc/essl
```

```
${MKL}/examples/vslf/essl
```

Convolution and correlation API provides interfaces for Fortran 90 and C/89 languages. You may use the C/89 interface also with later versions of C or C++, or Fortran 90 interface with programs written in Fortran 95. Note that there is no Fortran 77 support.

For users of the C/C++ and FORTRAN languages, the `mkl_vsl.h` and `mkl_vsl.fi` headers are provided. Both header files are found under the directory:

```
${MKL}/include
```

See more details about the FORTRAN header in [Random Number Generators](#) section of this chapter.

Convolution and correlation API is implemented through task objects, or tasks. Task object is a data structure, or descriptor, which holds parameters that determine the specific convolution or correlation operation. Such parameters may be precision, type, and number of dimensions of user data, an identifier of the computation algorithm to be used, shapes of data arrays, and so on.

All Intel MKL convolution and correlation routines process task objects in one way or another: either create a new task descriptor, change the parameter settings, compute mathematical results of the convolution or correlation using the stored parameters, or perform other operations. Accordingly, all routines are split into the following groups:

Task Constructors - routines that create a new task object descriptor and set up most common parameters.

Task Editors - routines that can set or modify some parameter settings in the existing task descriptor.

Task Execution Routines - compute results of the convolution or correlation operation over the actual input data, using the operation parameters held in the task descriptor.

Task Copy - routines used to make several copies of the task descriptor.

Task Destructors - routines that delete task objects and free the memory.

When the task is executed or copied for the first time, a special process runs which is called task commitment. During this process, consistency of task parameters is checked and the required work data are prepared. If the parameters are consistent, the task is tagged as committed successfully. The task remains committed until you edit its parameters. Hence, the task can be executed multiple times after a single commitment process. Since the task commitment process may include costly intermediate calculations such as preparation of Fourier transform of input data, launching the process only once can help speed up overall performance.

Naming Conventions

The names of FORTRAN routines in the convolution and correlation API are written in lowercase, while the names of FORTRAN types and constants are written in uppercase. The names are not case-sensitive.

In C, the names of routines, types, and constants are case-sensitive and can be lowercase and uppercase.

The names of routines have the following structure:

`vsl[datatype]{Conv|Corr}<base name>` for C-interface

`vsl[datatype]{conv|corr}<base name>` for FORTRAN-interface

where `vsl` is a prefix indicating that the routine belongs to Vector Statistical Library of Intel® MKL.

The field `[datatype]` is optional. If present, the symbol specifies the type of the input and output data and can be either `s` (for single precision real type) or `d` (for double precision real type).

The prefix `Conv` or `Corr` specifies whether the routine refers to convolution or correlation task, respectively.

The `<base name>` field specifies a particular functionality that the routine is designed for, for example, `NewTask`, `DeleteTask`.



NOTE. In this document, routines are often referred to by their base name when this does not lead to ambiguity. In the routine reference, the full name is always used in prototypes and code examples.

Data Types

All convolution or correlation routines use the following types for specifying data objects:

Type	Data Object
FORTTRAN: <code>TYPE (VSL_CONV_TASK)</code> C: <code>VSLConvTaskPtr</code>	Pointer to a task descriptor for convolution
FORTTRAN: <code>TYPE (VSL_CORR_TASK)</code> C: <code>VSLCorrTaskPtr</code>	Pointer to a task descriptor for correlation
FORTTRAN: <code>REAL (KIND=4)</code> C: <code>float</code>	Input/output user data in single precision
FORTTRAN: <code>REAL (KIND=8)</code>	Input/output user data in double precision

Type	Data Object
C: double	
FORTRAN: INTEGER	All other data
C: int	

Generic integer type (without specifying the byte size) is used for all integer data.



NOTE. The actual size of the generic integer type is platform-dependent. The appropriate byte size for integers must be chosen at the stage of compiling your software.

Parameters

Basic parameters held by the task descriptor are assigned values when the task object is created, copied, or modified by task editors. Parameters of the correlation or convolution task are initially set up by task constructors when the task object is created. Parameter changes or additional settings are made by task editors. More parameters which define location of the data being convolved need to be specified when the task execution routine is invoked.

According to how the parameters are passed or assigned values, all of them can be categorized as either explicit (directly passed as routine parameters when a task object is created or executed) or optional (assigned some default or implicit values during task construction).

The following table lists all applicable parameters used in the Intel MKL convolution and correlation API.

Table 10-14 Convolution and Correlation Task Parameters

Name	Category	Type	Default Value Label	Description
<i>job</i>	explicit	integer	Implied by the constructor name	Specifies whether the task relates to convolution or correlation
<i>type</i>	explicit	integer	Implied by the constructor name	Specifies the type (real or complex) of the input/output data. Set to real in the current version.
<i>precision</i>	explicit	integer	Implied by the constructor name	Specifies precision (single or double) of the input/output data to be provided in arrays <i>x</i> , <i>y</i> , <i>z</i> .

Name	Category	Type	Default Value Label	Description
<i>kind</i>	optional	integer	"linear"	Specifies whether the task relates to computing linear or circular convolution/correlation
<i>mode</i>	explicit	integer	None	Specifies whether the convolution/correlation computation should be done via Fourier transforms, or by a direct method, or by automatically choosing between the two. See SetMode for the list of named constants for this parameter.
<i>method</i>	optional	integer	"auto"	Hints at a particular computation method if several methods are available for the given <i>mode</i> . Setting this parameter to "auto" means that software will choose the best available method.
<i>internal_precision</i>	optional	integer	Set equal to the value of <i>precision</i>	Specifies precision of internal calculations. Can enforce double precision calculations even when input/output data are single precision. See SetInternalPrecision for the list of named constants for this parameter.
<i>dims</i>	explicit	integer	None	Specifies the rank (number of dimensions) of the user data provided in arrays <i>x</i> , <i>y</i> , <i>z</i> . Can be in the range from 1 to 7.
<i>x,y</i>	explicit	real arrays	None	Specify input data arrays. See Data Allocation for more information.
<i>z</i>	explicit	real array	None	Specifies output data array. See Data Allocation for more information.
<i>xshape, yshape, zshape</i>	explicit	integer arrays	None	Define shapes of the arrays <i>x</i> , <i>y</i> , <i>z</i> . See Data Allocation for more information.

Name	Category	Type	Default Value Label	Description
<i>xstride, ystride, zstride</i>	explicit	integer arrays	None	Define strides within arrays <i>x</i> , <i>y</i> , <i>z</i> , that is specify the physical location of the input and output data in these arrays. See Data Allocation for more information.
<i>start</i>	optional	integer array	Undefined	Defines the first element of the mathematical result that will be stored to output array <i>z</i> . See SetStart and Data Allocation for more information.
<i>decimation</i>	optional	integer array	Undefined	Defines how to thin out the mathematical result that will be stored to output array <i>z</i> . See SetDecimation and Data Allocation for more information.

Users of the C or C++ language may pass the NULL pointer instead of either or all of the parameters *xstride*, *ystride*, or *zstride* for multi-dimensional calculations. In this case, the software assumes the dense data allocation for the arrays *x*, *y*, or *z* due to the FORTRAN-style “by columns” representation of multi-dimensional arrays.

Task Status and Error Reporting

Task status is an integer value which is zero if no error has been detected while processing the task, or a specific non-zero error code otherwise. Negative status values are used for errors, and positive values are reserved for warnings.

An error can be caused by bad parameter values, system fault like memory allocation failure, or can be an internal error self-detected by the software.

Each task descriptor contains the current status of the task. When creating a task object, constructor assigns the `VSL_STATUS_OK` status to the task. When processing the task afterwards, other routines such as editors of executors can change the task status if an error occurs and write a corresponding error code into the task status field.

Note that at the stage of creating a task or editing its parameters, the set of parameters may be inconsistent. The parameter consistency check is only performed during the task commitment operation which is implicitly invoked before task execution or task copying. If an error is detected

at this stage, task execution or task copying is terminated and the task descriptor saves the corresponding error code. Once an error occurs, any further attempts to process that task descriptor will be terminated and the task will keep the same error code.

Normally, every convolution or correlation function (except `DeleteTask`) returns the status assigned to the task while performing the function operation.

The status codes are given symbolic names defined in the respective header files. For C/C++, these names are defined as macros via `#define` statements, and for FORTRAN as integer constants via `PARAMETER` operators.

If there is no error, the `VSL_STATUS_OK` is returned, which is defined as zero:

```
C/C++:          #define, VSL_STATUS_OK 0
F90/F95:        INTEGER, PARAMETER:: VSL_STATUS_OK = 0
```

In case of an error, a non-zero error code is returned, which signals about the origin of the failure. The following status codes for the convolution/correlation error codes are pre-defined in the header files for both C/C++ and FORTRAN languages.

Table 10-15 Convolution/Correlation Status Codes and Messages

Status Code	Message
<code>VSL_CC_ERROR_NOT_IMPLEMENTED,</code>	Requested functionality is not implemented.
<code>VSL_CC_ERROR_ALLOCATION_FAILURE</code>	Memory allocation failure.
<code>VSL_CC_ERROR_BAD_DESCRIPTOR</code>	Task descriptor is corrupted.
<code>VSL_CC_ERROR_SERVICE_FAILURE</code>	A service function has failed.
<code>VSL_CC_ERROR_EDIT_FAILURE</code>	Failure while editing the task.
<code>VSL_CC_ERROR_EDIT_PROHIBITED</code>	You cannot edit this parameter.
<code>VSL_CC_ERROR_COMMIT_FAILURE</code>	Task commitment has failed.
<code>VSL_CC_ERROR_COPY_FAILURE</code>	Failure while copying the task.
<code>VSL_CC_ERROR_DELETE_FAILURE</code>	Failure while deleting the task.
<code>VSL_CC_ERROR_BAD_ARGUMENT</code>	Bad argument or task parameter.
<code>VSL_CC_ERROR_JOB</code>	Bad parameter: <i>job</i> .
<code>SL_CC_ERROR_KIND</code>	Bad parameter: <i>kind</i> .

Status Code	Message
VSL_CC_ERROR_MODE	Bad parameter: <i>mode</i> .
VSL_CC_ERROR_METHOD	Bad parameter: <i>method</i> .
VSL_CC_ERROR_TYPE	Bad parameter: <i>type</i> .
VSL_CC_ERROR_EXTERNAL_PRECISION	Bad parameter: <i>external_precision</i> .
VSL_CC_ERROR_INTERNAL_PRECISION	Bad parameter: <i>internal_precision</i> .
VSL_CC_ERROR_PRECISION	Incompatible external/internal precisions.
VSL_CC_ERROR_DIMS	Bad parameter: <i>dims</i> .
VSL_CC_ERROR_XSHAPE	Bad parameter: <i>xshape</i> .
VSL_CC_ERROR_YSHAPE	Bad parameter: <i>yshape</i> .
	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> n_{\max}$.
VSL_CC_ERROR_ZSHAPE	Bad parameter: <i>zshape</i> .
VSL_CC_ERROR_XSTRIDE	Bad parameter: <i>xstride</i> .
VSL_CC_ERROR_YSTRIDE	Bad parameter: <i>ystride</i> .
VSL_CC_ERROR_ZSTRIDE	Bad parameter: <i>zstride</i> .
VSL_CC_ERROR_X	Bad parameter: <i>x</i> .
VSL_CC_ERROR_Y	Bad parameter: <i>y</i> .
VSL_CC_ERROR_Z	Bad parameter: <i>z</i> .
VSL_CC_ERROR_START	Bad parameter: <i>start</i> .
VSL_CC_ERROR_DECIMATION	Bad parameter: <i>decimation</i> .
VSL_CC_ERROR_OTHER	Some other error.

Task Constructors

Task constructors are routines intended for creating a new task descriptor and setting up basic parameters. This means that no additional parameter adjustment is typically required and other routines can use the task object.

Intel® MKL implementation of the convolution and correlation API provides two different forms of constructors: a general form and an X-form. X-form constructors work in the same way as the general form but also assign particular data to the first operand vector used in convolution or correlation operation (stored in array *x*).

Using X-form constructors is recommended when you need to compute multiple convolutions or correlations with the same data vector held in array *x* against different vectors held in array *y*. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

For each constructor routine there is also an associated one-dimensional version which exploits the algorithmic and computational benefits provided by the simplicity of the data structures for one-dimensional case.



NOTE. If constructor fails to create a task descriptor, it returns `NULL` task pointer.

The [Table 10-16](#) lists available task constructors:

Table 10-16 Task Constructors

Routine	Description
NewTask	Creates a new convolution or correlation task descriptor for a multidimensional case.
NewTask1D	Creates a new convolution or correlation task descriptor for a one-dimensional case.
NewTaskX	Creates a new convolution or correlation task descriptor as an X-form for a multidimensional case.
NewTaskX1D	Creates a new convolution or correlation task descriptor as an X-form for a one-dimensional case.

NewTask

Creates a new convolution or correlation task descriptor for multidimensional case.

Syntax

Fortran:

```
status = vslsconvnewtask(task, mode, dims, xshape, yshape, zshape)
status = vsldconvnewtask(task, mode, dims, xshape, yshape, zshape)
status = vslscorrnewtask(task, mode, dims, xshape, yshape, zshape)
status = vsldcorrnewtask(task, mode, dims, xshape, yshape, zshape)
```

C:

```
status = vslsConvNewTask(task, mode, dims, xshape, yshape, zshape);
status = vsldConvNewTask(task, mode, dims, xshape, yshape, zshape);
status = vslsCorrNewTask(task, mode, dims, xshape, yshape, zshape);
status = vsldCorrNewTask(task, mode, dims, xshape, yshape, zshape);
```

Description

Each `NewTask` constructor creates a new convolution or correlation task descriptor with the user specified values for explicit parameters. The optional parameters are set to their default values (see [Table 10-14](#)).

The parameters `xshape`, `yshape`, and `zshape` define the shapes of the input and output data provided by the arrays `x`, `y`, and `z`, respectively. Each shape parameter is an array of integers with its length equal to the value of `dims`. You explicitly assign the shape parameters when calling the constructor. If the value of the parameter `dims` is 1, then `xshape`, `yshape`, `zshape` are equal to the number of elements read from the arrays `x` and `y` or stored to the array `z`. Note that values of shape parameters may differ from physical shapes of arrays `x`, `y`, and `z` if non-trivial strides are assigned.

If constructor fails to create a task descriptor, it returns `NULL` task pointer.

Input Parameters

Name	Type	Description
<i>mode</i>	FORTTRAN: INTEGER C: int	Specifies whether convolution/correlation calculation must be performed by using a direct algorithm or through Fourier transform of the input data. See Table 10-18 for a list of possible values.
<i>dims</i>	FORTTRAN: INTEGER C: int	Rank of user data. Specifies number of dimensions for the input and output arrays <i>x</i> , <i>y</i> , and <i>z</i> used during the execution stage. Must be in the range from 1 to 7. The value is explicitly assigned by the constructor.
<i>xshape</i>	FORTTRAN: INTEGER, DIMENSION(*) C: int[]	Defines the shape of the input data for the source array <i>x</i> . See Data Allocation for more information.
<i>yshape</i>	FORTTRAN: INTEGER, DIMENSION(*) C: int[]	Defines the shape of the input data for the source array <i>y</i> . See Data Allocation for more information.
<i>zshape</i>	FORTTRAN: INTEGER, DIMENSION(*) C: int[]	Defines the shape of the output data to be stored in array <i>z</i> . See Data Allocation for more information.

Output Parameters

Name	Type	Description
<i>task</i>	FORTTRAN: TYPE(VSL_CONV_TASK) for vslsconvnewtask, vsldconvnewtask TYPE(VSL_CORR_TASK) for vslscorrnewtask, vsldcorrnewtask	Pointer to the task descriptor if created successfully or NULL pointer otherwise.

Name	Type	Description
	C: VSLConvTaskPtr* for vslsConvNewTask, vsldConvNewTask VSLCorrTaskPtr* for vslsCorrNewTask, vsldCorrNewTask	
<i>status</i>	FORTRAN: INTEGER C: int	Set to VSL_STATUS_OK if the task is created successfully or set to non-zero error code otherwise.

NewTask1D

*Creates a new convolution or correlation task
descriptor for one-dimensional case.*

Syntax

Fortran:

```
status = vslsconvnewtask1d(task, mode, xshape, yshape, zshape)
status = vsldconvnewtask1d(task, mode, xshape, yshape, zshape)
status = vslscorrnewtask1d(task, mode, xshape, yshape, zshape)
status = vsldcorrnewtask1d(task, mode, xshape, yshape, zshape)
```

C:

```
status = vslsConvNewTask1D(task, mode, xshape, yshape, zshape);
status = vsldConvNewTask1D(task, mode, xshape, yshape, zshape);
status = vslsCorrNewTask1D(task, mode, xshape, yshape, zshape);
status = vsldCorrNewTask1D(task, mode, xshape, yshape, zshape);
```

Description

Each `NewTask1D` constructor creates a new convolution or correlation task descriptor with the user specified values for explicit parameters. The optional parameters are set to their default values (see [Table 10-14](#)). Unlike `NewTask`, these routines represent a special one-dimensional

version of the constructor which assumes that the value of the parameter *dims* is 1. The parameters *xshape*, *yshape*, and *zshape* are equal to the number of elements read from the arrays *x* and *y* or stored to the array *z*. You explicitly assign the shape parameters when calling the constructor.

Input Parameters

Name	Type	Description
<i>mode</i>	FORTTRAN: INTEGER C: int	Specifies whether convolution/correlation calculation must be performed by using a direct algorithm or through Fourier transform of the input data. See Table 10-18 for a list of possible values.
<i>xshape</i>	FORTTRAN: INTEGER C: int	Defines the length of the input data sequence for the source array <i>x</i> . See Data Allocation for more information.
<i>yshape</i>	FORTTRAN: INTEGER C: int	Defines the length of the input data sequence for the source array <i>y</i> . See Data Allocation for more information.
<i>zshape</i>	FORTTRAN: INTEGER C: int	Defines the length of the output data sequence to be stored in array <i>z</i> . See Data Allocation for more information.

Output Parameters

Name	Type	Description
<i>task</i>	FORTTRAN: TYPE(VSL_CONV_TASK) for vslsconvnewtaskld, vsldconvnewtaskld TYPE(VSL_CORR_TASK) for vslscorrnewtaskld, vsldcorrnewtaskld	Pointer to the task descriptor if created successfully or NULL pointer otherwise.

Name	Type	Description
	C: VSLConvTaskPtr* for vslsConvNewTask1D, vsldConvNewTask1D VSLCorrTaskPtr* for vslsCorrNewTask1D, vsldCorrNewTask1D	
<i>status</i>	FORTRAN: INTEGER C: int	Set to VSL_STATUS_OK if the task is created successfully or set to non-zero error code otherwise.

NewTaskX

*Creates a new convolution or correlation task
descriptor for multidimensional case and assigns
source data to the first operand vector.*

Syntax

Fortran:

```
status = vslsconvnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
status = vsldconvnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
status = vslscorrnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
status = vsldcorrnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
```

C:

```
status = vslsConvNewTaskX(task, mode, dims, xshape, yshape, zshape, x,
xstride);

status = vsldConvNewTaskX(task, mode, dims, xshape, yshape, zshape, x,
xstride);

status = vslsCorrNewTaskX(task, mode, dims, xshape, yshape, zshape, x,
xstride);

status = vsldCorrNewTaskX(task, mode, dims, xshape, yshape, zshape, x,
xstride);
```

Description

Each `NewTaskX` constructor creates a new convolution or correlation task descriptor with the user specified values for explicit parameters. The optional parameters are set to their default values (see [Table 10-14](#)).

Unlike `NewTask`, these routines represent the so called X-form version of the constructor, which means that in addition to creating the task descriptor they assign particular data to the first operand vector in array `x` used in convolution or correlation operation. The task descriptor created by the `NewTaskX` constructor keeps the pointer to the array `x` all the time, that is, until the task object is deleted by one of the destructor routines (see `DeleteTask`).

Using this form of constructors is recommended when you need to compute multiple convolutions or correlations with the same data vector in array `x` against different vectors in array `y`. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

The parameters `xshape`, `yshape`, and `zshape` define the shapes of the input and output data provided by the arrays `x`, `y`, and `z`, respectively. Each shape parameter is an array of integers with its length equal to the value of `dims`. You explicitly assign the shape parameters when calling the constructor. If the value of the parameter `dims` is 1, then `xshape`, `yshape`, and `zshape` are equal to the number of elements read from the arrays `x` and `y` or stored to the array `z`. Note that values of shape parameters may differ from physical shapes of arrays `x`, `y`, and `z` if non-trivial strides are assigned.

The stride parameter `xstride` specifies the physical location of the input data in the array `x`. In a one-dimensional case, stride is an interval between locations of consecutive elements of the array. For example, if the value of the parameter `xstride` is `s`, then only every s^{th} element of the array `x` will be used to form the input sequence. The stride value must be positive or negative but not zero.

Input Parameters

Name	Type	Description
<code>mode</code>	FORTAN: INTEGER C: <code>int</code>	Specifies whether convolution/correlation calculation must be performed by using a direct algorithm or through Fourier transform of the input data. See Table 10-18 for a list of possible values.

Name	Type	Description
<i>dims</i>	FORTRAN: INTEGER C: int	Rank of user data. Specifies number of dimensions for the input and output arrays <i>x</i> , <i>y</i> , and <i>z</i> used during the execution stage. Must be in the range from 1 to 7. The value is explicitly assigned by the constructor.
<i>xshape</i>	FORTRAN: INTEGER, DIMENSION (*) C: int[]	Defines the shape of the input data for the source array <i>x</i> . See Data Allocation for more information.
<i>yshape</i>	FORTRAN: INTEGER, DIMENSION (*) C: int[]	Defines the shape of the input data for the source array <i>y</i> . See Data Allocation for more information.
<i>zshape</i>	FORTRAN: INTEGER, DIMENSION (*) C: int[]	Defines the shape of the output data to be stored in array <i>z</i> .See Data Allocation for more information.
<i>x</i>	FORTRAN: REAL (KIND=4) , DIMENSION (*) for single precision flavors, REAL (KIND=8) , DIMENSION (*) for double precision flavors C: float[] for single precision flavors double[] for double precision flavors	Pointer to the array containing input data for the first operand vector.See Data Allocation for more information.
<i>xstride</i>	FORTRAN: INTEGER, DIMENSION (*) C: int[]	Strides for input data in the array <i>x</i> .

Output Parameters

Name	Type	Description
<i>task</i>	FORTTRAN: TYPE(VSL_CONV_TASK) for vslsconvnewtaskx, vsldconvnewtaskx TYPE(VSL_CORR_TASK) for vslscorrnewtaskx, vsldcorrnewtaskx C: VSLConvTaskPtr* for vslsConvNewTaskX, vsldConvNewTaskX VSLCorrTaskPtr* for vslsCorrNewTaskX, vsldCorrNewTaskX	Pointer to the task descriptor if created successfully or NULL pointer otherwise.
<i>status</i>	FORTTRAN: INTEGER C: int	Set to VSL_STATUS_OK if the task is created successfully or set to non-zero error code otherwise.

NewTaskX1D

Creates a new convolution or correlation task descriptor for one-dimensional case and assigns source data to the first operand vector.

Syntax

Fortran:

```
status = vslsconvnewtaskx1d(task, mode, xshape, yshape, zshape, x, xstride)
```

```
status = vsldconvnewtaskx1d(task, mode, xshape, yshape, zshape, x, xstride)
```

```
status = vslscorrnewtaskx1d(task, mode, xshape, yshape, zshape, x, xstride)
```

```
status = vsldcorrnewtaskx1d(task, mode, xshape, yshape, zshape, x, xstride)
```

C:

```
status = vslsConvNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
status = vsldConvNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
status = vslsCorrNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
status = vsldCorrNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
```

Description

Each `NewTaskX1D` constructor creates a new convolution or correlation task descriptor with the user specified values for explicit parameters. The optional parameters are set to their default values (see [Table 10-14](#)).

These routines represent a special one-dimensional version of the so called X-form of the constructor. This assumes that the value of the parameter `dims` is 1 and that in addition to creating the task descriptor, constructor routines assign particular data to the first operand vector in array `x` used in convolution or correlation operation. The task descriptor created by the `NewTaskX1D` constructor keeps the pointer to the array `x` all the time, that is, until the task object is deleted by one of the destructor routines (see [DeleteTask](#)).

Using this form of constructors is recommended when you need to compute multiple convolutions or correlations with the same data vector in array `x` against different vectors in array `y`. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

The parameters `xshape`, `yshape`, and `zshape` are equal to the number of elements read from the arrays `x` and `y` or stored to the array `z`. You explicitly assign the shape parameters when calling the constructor.

The [stride parameters](#) `xstride` specifies the physical location of the input data in the array `x` and is an interval between locations of consecutive elements of the array. For example, if the value of the parameter `xstride` is `s`, then only every s^{th} element of the array `x` will be used to form the input sequence. The stride value must be positive or negative but not zero.

Input Parameters

Name	Type	Description
<code>mode</code>	FORTRAN: INTEGER C: int	Specifies whether convolution/correlation calculation must be performed by using a direct algorithm or through Fourier transform of the input data. See Table 10-18 for a list of possible values.

Name	Type	Description
<i>xshape</i>	FORTTRAN: INTEGER C: int	Defines the length of the input data sequence for the source array <i>x</i> . See Data Allocation for more information.
<i>yshape</i>	FORTTRAN: INTEGER C: int	Defines the length of the input data sequence for the source array <i>y</i> . See Data Allocation for more information.
<i>zshape</i>	FORTTRAN: INTEGER C: int	Defines the length of the output data sequence to be stored in array <i>z</i> . See Data Allocation for more information.
<i>x</i>	FORTTRAN: REAL (KIND=4) , DIMENSION (*) for single precision flavors, REAL (KIND=8) , DIMENSION (*) for double precision flavors C: float[] for single precision flavors double[] for double precision flavors	Pointer to the array containing input data for the first operand vector. See Data Allocation for more information.
<i>xstride</i>	FORTTRAN: INTEGER C: int	Stride for input data sequence in the array <i>x</i> .

Output Parameters

Name	Type	Description
<i>task</i>	FORTTRAN: TYPE (VSL_CONV_TASK) for vslsconvnewtaskxld, vsldconvnewtaskxld	Pointer to the task descriptor if created successfully or NULL pointer otherwise.

Name	Type	Description
	TYPE (VSL_CORR_TASK) for vslscorrnewtaskxld, vsldcorrnewtaskxld C: VSLConvTaskPtr* for vslsConvNewTaskXlD, vsldConvNewTaskXlD VSLCorrTaskPtr* for vslsCorrNewTaskXlD, vsldCorrNewTaskXlD	
<i>status</i>	FORTRAN: INTEGER C: int	Set to VSL_STATUS_OK if the task is created successfully or set to non-zero error code otherwise.

Task Editors

Task editors in convolution and correlation API of the Intel MKL are routines intended for setting up or changing the following task parameters (see [Table 10-14](#)):

- *mode*
- *internal_precision*
- *start*
- *decimation.*

For setting up or changing each of the above parameters, a separate routine exists.



NOTE. Fields of the task descriptor structure are accessible only through the set of task editor routines provided with the software.

After applying any of the editor routines to change the task descriptor settings, the task loses its commitment status and will go through the full commitment process again during the next execution or copy operation. This is motivated by the fact that the currently stored work data computed during the last commitment process may become invalid with respect to new parameter settings. For more information about task commitment, see [Overview](#).

[Table 10-17](#) lists available task editors.

Table 10-17 Task Editors

Routine	Description
<code>SetMode</code>	Changes the value of the parameter <i>mode</i> for the operation of convolution or correlation.
<code>SetInternalPrecision</code>	Changes the value of the parameter <i>internal_precision</i> for the operation of convolution or correlation.
<code>SetStart</code>	Sets the value of the parameter <i>start</i> for the operation of convolution or correlation.
<code>SetDecimation</code>	Sets the value of the parameter <i>decimation</i> for the operation of convolution or correlation.



NOTE. You can use the `NULL` task pointer in calls to editor routines. In this case, the routine will be terminated and no system crash will occur.

SetMode

Changes the value of the parameter `mode` in the convolution or correlation task descriptor.

Syntax

Fortran:

```
status = vslconvsetmode(task, newmode)
status = vslcorrsetmode(task, newmode)
```

C:

```
status = vslConvSetMode(task, newmode);
status = vslCorrSetMode(task, newmode);
```

Description

The routine changes the value of the parameter `mode` for the operation of convolution or correlation. This parameter defines whether the computation should be done via Fourier transforms of the input/output data or using a direct algorithm. Initial value for `mode` is assigned by a task constructor.

Predefined values for the `mode` parameter are as follows:

Table 10-18 Values of `mode` parameter

Value	Purpose
VSL_CONV_MODE_FFT	Compute convolution by using fast Fourier transform.
VSL_CORR_MODE_FFT	Compute correlation by using fast Fourier transform.
VSL_CONV_MODE_DIRECT	Compute convolution directly.
VSL_CORR_MODE_DIRECT	Compute correlation directly.
VSL_CONV_MODE_AUTO	Automatically choose direct or Fourier mode for convolution.
VSL_CORR_MODE_AUTO	Automatically choose direct or Fourier mode for correlation.

Input Parameters

Name	Type	Description
<i>task</i>	FORTRAN: TYPE (VSL_CONV_TASK) for vslconvsetmode TYPE (VSL_CORR_TASK) for vslcorrsetmode C: VSLConvTaskPtr for vslConvSetMode VSLCorrTaskPtr for vslCorrSetMode	Pointer to the task descriptor.
<i>newmode</i>	FORTRAN: INTEGER C: int	New value of the parameter <code>mode</code> .

Output Parameters

Name	Type	Description
<i>status</i>	FORTTRAN: INTEGER C: int	Current status of the task.

SetInternalPrecision

Changes the value of the parameter `internal_precision` in the convolution or correlation task descriptor.

Syntax

Fortran:

```
status = vslconvsetinternalprecision(task, precision)  
status = vslcorrsetinternalprecision(task, precision)
```

C:

```
status = vslConvSetInternalPrecision(task, precision);  
status = vslCorrSetInternalPrecision(task, precision);
```

Description

The routine changes the value of the parameter *internal_precision* for the operation of convolution or correlation. This parameter defines whether the internal computations of the convolution or correlation result should be done in single or double precision. Initial value for *internal_precision* is assigned by a task constructor and set to either “single” or “double” according to the particular flavor of the constructor used.

Changing the *internal_precision* can be useful if the default setting of this parameter was “single” but you want to calculate the result with double precision even if input and output data are represented in single precision.

Predefined values for the *internal_precision* input parameter are as follows:

Table 10-19 Values of *internal_precision* Parameter

Value	Purpose
VSL_CONV_PRECISION_SINGLE	Compute convolution with single precision.
VSL_CORR_PRECISION_SINGLE	Compute correlation with single precision.
VSL_CONV_PRECISION_DOUBLE	Compute convolution with double precision.
VSL_CORR_PRECISION_DOUBLE	Compute correlation with double precision.

Input Parameters

Name	Type	Description
<i>task</i>	FORTRAN: TYPE(VSL_CONV_TASK) for vslconvsetinternalprecision TYPE(VSL_CORR_TASK) for vslcorrsetinternalprecision C: VSLConvTaskPtr for vslConvSetInternalPrecision VSLCorrTaskPtr for vslCorrSetInternalPrecision	Pointer to the task descriptor.
<i>precision</i>	FORTRAN: INTEGER C: int	New value of the parameter <i>internal_precision</i> .

Output Parameters

Name	Type	Description
<i>status</i>	FORTRAN: INTEGER C: int	Current status of the task.

SetStart

Changes the value of the parameter `start` in the convolution or correlation task descriptor.

Syntax

Fortran:

```
status = vslconvsetstart(task, start)
```

```
status = vslcorrsetstart(task, start)
```

C:

```
status = vslConvSetStart(task, start);
```

```
status = vslCorrSetStart(task, start);
```

Description

The routine sets the value of the parameter `start` for the operation of convolution or correlation. In a one-dimensional case, this parameter points to the first element in the mathematical result that should be stored in the output array. In a multidimensional case, `start` is an array of indices and its length is equal to the number of dimensions specified by the parameter `dims`. For more information about the definition and effect of this parameter, see [Data Allocation](#).

During the initial task descriptor construction, the default value for `start` is undefined and this parameter is not used. Hence, the only way to set and use the `start` parameter is via assigning it some value by one of the `SetStart` routines.

Input Parameters

Name	Type	Description
<code>task</code>	FORTTRAN: TYPE(VSL_CONV_TASK) for vslconvsetstart TYPE(VSL_CORR_TASK) for vslcorrsetstart C: VSLConvTaskPtr for vslConvSetStart	Pointer to the task descriptor.

Name	Type	Description
	VSLCorrTaskPtr for vslCorrSetStart	
<i>start</i>	FORTRAN: INTEGER, DIMENSION (*) C: int[]	New value of the parameter <i>start</i> .

Output Parameters

Name	Type	Description
<i>status</i>	FORTRAN: INTEGER C: int	Current status of the task.

SetDecimation

Changes the value of the parameter `decimation` in the convolution or correlation task descriptor.

Syntax

Fortran:

```
status = vslconvsetdecimation(task, decimation)
status = vslcorrsetdecimation(task, decimation)
```

C:

```
status = vslConvSetDecimation(task, decimation);
status = vslCorrSetDecimation(task, decimation);
```

Description

The routine sets the value of the parameter *decimation* for the operation of convolution or correlation.

This parameter determines how to thin out the mathematical result of convolution or correlation before writing it into the output data array. For example, in a one-dimensional case, if *decimation* = *d* > 1, only every *d*-th element of the mathematical result is written to the output

array *z*. In a multidimensional case, *decimation* is an array of indices and its length is equal to the number of dimensions specified by the parameter *dims*. For more information about the definition and effect of this parameter, see [Data Allocation](#).

During the initial task descriptor construction, the default value for *decimation* is undefined and this parameter is not used. Hence, the only way to set and use the *decimation* parameter is via assigning it some value by one of the `SetDecimation` routines.

Input Parameters

Name	Type	Description
<i>task</i>	FORTTRAN: TYPE(VSL_CONV_TASK) for vslconvsetdecimation TYPE(VSL_CORR_TASK) for vslcorrsetdecimation C: VSLConvTaskPtr for vslConvSetDecimation VSLCorrTaskPtr for vslCorrSetDecimation	Pointer to the task descriptor.
<i>decimation</i>	FORTTRAN: INTEGER, DIMENSION (*) C: int[]	New value of the parameter <i>decimation</i> .

Output Parameters

Name	Type	Description
<i>status</i>	FORTTRAN: INTEGER C: int	Current status of the task.

Task Execution Routines

Task execution routines compute convolution or correlation results based on parameters held by the task descriptor and on the supplied user data for input vectors.

Once created and adjusted, the task can be executed multiple times by applying to different input/output data of the same type, precision, and shape.

Intel MKL implementation of the convolution and correlation API provides two different forms of execution routines: a general form and an X-form. General form executors use the task descriptor created by the general form constructor and expect to get two source data arrays *x* and *y* on input. Alternatively, X-form executors use the task descriptor created by the X-form constructor and expect to get only one source data array *y* on input because the first array *x* has been already specified on the construction stage.

When the task is executed for the first time, the execution routine includes task commitment operation, which involves two basic steps: parameters consistency check and preparation of auxiliary data (for example, this might be the calculation of Fourier transform for input data).

For each execution routine there is also an associated one-dimensional version which exploits the algorithmic and computational benefits provided by the simplicity of the data structures for one-dimensional case.



NOTE. You can use the `NULL` task pointer in calls to execution routines. In this case, the routine will be terminated and no system crash will occur.

If the task is executed successfully, the execution routine returns zero status code. If an error is detected, the execution routine returns an error code which signals that a specific error has occurred. In particular, an error status code is returned in the following cases:

- if the task pointer is `NULL`,
- if the task descriptor is corrupted,
- if calculation has failed for some other reason.



NOTE. Intel® MKL does not control floating-point errors, like overflow or gradual underflow, or operations with NaNs, etc.

If an error occurs, the task descriptor stores the error code.

The table below lists all task execution routines.

Table 10-20 Task Execution Routines

Routine	Description
<code>Exec</code>	Computes convolution or correlation for a multidimensional case.
<code>Exec1D</code>	Computes convolution or correlation for a one-dimensional case.

Routine	Description
ExecX	Computes convolution or correlation as X-form for a multidimensional case.
ExecX1D	Computes convolution or correlation as X-form for a one-dimensional case.

Exec

Computes convolution or correlation for multidimensional case.

Syntax

Fortran:

```
status = vslsconvexec(task, x, xstride, y, ystride, z, zstride)
status = vsldconvexec(task, x, xstride, y, ystride, z, zstride)
status = vslscorrexec(task, x, xstride, y, ystride, z, zstride)
status = vsldcorrexec(task, x, xstride, y, ystride, z, zstride)
```

C:

```
status = vslsConvExec(task, x, xstride, y, ystride, z, zstride);
status = vsldConvExec(task, x, xstride, y, ystride, z, zstride);
status = vslsCorrExec(task, x, xstride, y, ystride, z, zstride);
status = vsldCorrExec(task, x, xstride, y, ystride, z, zstride);
```

Description

Each of the `Exec` routines computes convolution or correlation of the data provided by the arrays `x` and `y` and then stores the results in the array `z`. Parameters of the operation are read from the task descriptor created previously by a corresponding `NewTask` constructor and pointed to by `task`. If `task` is `NULL`, no operation is done.

The stride parameters `xstride`, `ystride`, and `zstride` specify the physical location of the input and output data in the arrays `x`, `y`, and `z`, respectively. In a one-dimensional case, stride is an interval between locations of consecutive elements of the array. For example, if the value of the parameter `zstride` is `s`, then only every s^{th} element of the array `z` will be used to store the output data. The stride value must be positive or negative but not zero.

Input Parameters

Name	Type	Description
<i>task</i>	FORTTRAN: TYPE (VSL_CONV_TASK) for vslsconvexec and vsldconvexec TYPE (VSL_CORR_TASK) for vslscorrexec and vsldcorrexec C: VSLConvTaskPtr for vslsConvExec and vsldConvExec VSLCorrTaskPtr for vslsCorrExec and vsldCorrExec	Pointer to the task descriptor.
<i>x, y</i>	FORTTRAN: REAL (KIND=4) , DIMENSION (*) for vslsconvexec and vslscorrexec REAL (KIND=8) , DIMENSION (*) for vsldconvexec and vsldcorrexec C: float[] for vslsConvExec and vslsCorrExec double[] for vsldConvExec and vsldCorrExec	Pointers to arrays containing input data. See Data Allocation for more information.
<i>xstride, ystride, zstride</i>	FORTTRAN: INTEGER, DIMENSION (*) C: int[]	Strides for input and output data. For more information, see stride parameters .

Output Parameters

Name	Type	Description
<i>z</i>	FORTTRAN: REAL(KIND=4) , DIMENSION(*) for vslsconvexec and vslscorexec REAL(KIND=8) , DIMENSION(*) for vsldconvexec and vsldcorexec C: float[] for vslsConvExec and vslsCorrExec double[] for vsldConvExec and vsldCorrExec	Pointer to the array that stores output data. See Data Allocation for more information.
<i>status</i>	FORTTRAN: INTEGER C: int	Set to VSL_STATUS_OK if the task is executed successfully or set to non-zero error code otherwise.

Exec1D

Computes convolution or correlation for one-dimensional case.

Syntax

Fortran:

```

status = vslsconvexec1d(task, x, xstride, y, ystride, z, zstride)
status = vsldconvexec1d(task, x, xstride, y, ystride, z, zstride)
status = vslscorexec1d(task, x, xstride, y, ystride, z, zstride)
status = vsldcorexec1d(task, x, xstride, y, ystride, z, zstride)

```


C:

```
status = vslsConvExec1D(task, x, xstride, y, ystride, z, zstride);
status = vsldConvExec1D(task, x, xstride, y, ystride, z, zstride);
status = vslsCorrExec1D(task, x, xstride, y, ystride, z, zstride);
status = vsldCorrExec1D(task, x, xstride, y, ystride, z, zstride);
```

Description

Each of the `Exec1D` routines computes convolution or correlation of the data provided by the arrays `x` and `y` and then stores the results in the array `z`. These routines represent a special one-dimensional version of the operation, assuming that the value of the parameter `dims` is 1. Using this version of execution routines can help speed up performance in case of one-dimensional data.

Parameters of the operation are read from the task descriptor created previously by a corresponding `NewTask1D` constructor and pointed to by `task`. If `task` is `NULL`, no operation is done.

Input Parameters

Name	Type	Description
<code>task</code>	FORTRAN: TYPE (VSL_CONV_TASK) for vslsconvexec1d and vsldconvexec1d TYPE (VSL_CORR_TASK) for vslscorexec1d and vsldcorexec1d C: VSLConvTaskPtr for vslsConvExec1D and vsldConvExec1D VSLCorrTaskPtr for vslsCorrExec1D and vsldCorrExec1D	Pointer to the task descriptor.

Name	Type	Description
<i>x, y</i>	FORTTRAN: REAL(KIND=4) , DIMENSION(*) for vslsconvexecld and vslscorexecld REAL(KIND=8) , DIMENSION(*) for vsldconvexecld and vsldcorexecld C: float[] for vslsConvExec1D and vslsCorrExec1D double[] for vsldConvExe1D and vsldCorrExec1D	Pointers to arrays containing input data. See Data Allocation for more information.
<i>xstride, ystride, zstride</i>	FORTTRAN: INTEGER C: int	Strides for input and output data. For more information, see stride parameters .

Output Parameters

Name	Type	Description
<i>z</i>	FORTTRAN: REAL(KIND=4) , DIMENSION(*) for vslsconvexecld and vslscorexecld REAL(KIND=8) , DIMENSION(*) for vsldconvexecld and vsldcorexecld C: float[] for vslsConvExec1D and vslsCorrExec1D	Pointer to the array that stores output data. See Data Allocation for more information.

Name	Type	Description
	double[] for vsldConvExec1D and vsldCorrExec1D	
<i>status</i>	FORTRAN: INTEGER C: int	Set to VSL_STATUS_OK if the task is executed successfully or set to non-zero error code otherwise.

ExecX

Computes convolution or correlation for multidimensional case with the fixed first operand vector.

Syntax

Fortran:

```
status = vslsconvexec(task, y, ystride, z, zstride)
status = vsldconvexec(task, y, ystride, z, zstride)
status = vslscorrexec(task, y, ystride, z, zstride)
status = vsldcorrexec(task, y, ystride, z, zstride)
```

C:

```
status = vslsConvExecX(task, y, ystride, z, zstride);
status = vsldConvExecX(task, y, ystride, z, zstride);
status = vslsCorrExecX(task, y, ystride, z, zstride);
status = vsldCorrExecX(task, y, ystride, z, zstride);
```

Description

Each of the ExecX routines computes convolution or correlation of the data provided by the arrays *x* and *y* and then stores the results in the array *z*. These routines represent a special version of the operation, which assumes that the first operand vector was set on the task construction stage and the task object keeps the pointer to the array *x*.

Parameters of the operation are read from the task descriptor created previously by a corresponding [NewTaskX](#) constructor and pointed to by *task*. If *task* is NULL, no operation is done.

Using this form of execution routines is recommended when you need to compute multiple convolutions or correlations with the same data vector in array *x* against different vectors in array *y*. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

Input Parameters

Name	Type	Description
<i>task</i>	FORTTRAN: TYPE(VSL_CONV_TASK) for vslsconvexecx and vsldconvexecx TYPE(VSL_CORR_TASK) for vslscorreexecx and vsldcorreexecx C: VSLConvTaskPtr for vslsConvExecX and vsldConvExecX VSLCorrTaskPtr for vslsCorrExecX and vsldCorrExecX	Pointer to the task descriptor.
<i>x ,y</i>	FORTTRAN: REAL(KIND=4) , DIMENSION(*) for vslsconvexecx and vslscorreexecx REAL(KIND=8) , DIMENSION(*) for vsldconvexecx and vsldcorreexecx C: float[] for vslsConvExecX and vslsCorrExecX	Pointer to array containing input data (for the second operand vector). See Data Allocation for more information.

Name	Type	Description
	double[] for vsldConvExeX and vsldCorrExecX	
<i>ystride</i> <i>,zstride</i>	FORTTRAN: INTEGER, DIMENSION(*) C: int[]	Strides for input and output data. For more information, see stride parameters .

Output Parameters

Name	Type	Description
<i>z</i>	FORTTRAN: REAL(KIND=4) , DIMENSION(*) for vslsconvexecx and vsldcorrexecx REAL(KIND=8) , DIMENSION(*) for vsldconvexecx and vsldcorrexecx C: float[] for vsldConvExecX and vsldCorrExecX double[] for vsldConvExecX and vsldCorrExecX	Pointer to the array that stores output data. See Data Allocation for more information.
<i>status</i>	FORTTRAN: INTEGER C: int	Set to VSL_STATUS_OK if the task is executed successfully or set to non-zero error code otherwise.

ExecX1D

Computes convolution or correlation for one-dimensional case with the fixed first operand vector.

Syntax

Fortran:

```
status = vslsconvexecx1d(task, y, ystride, z, zstride)
status = vsldconvexecx1d(task, y, ystride, z, zstride)
status = vslscorrexx1d(task, y, ystride, z, zstride)
status = vsldcorrexx1d(task, y, ystride, z, zstride)
```

C:

```
status = vslsConvExecX1D(task, y, ystride, z, zstride);
status = vsldConvExecX1D(task, y, ystride, z, zstride);
status = vslsCorrExecX1D(task, y, ystride, z, zstride);
status = vsldCorrExecX1D(task, y, ystride, z, zstride);
```

Description

Each of the `ExecX1D` routines computes convolution or correlation of one-dimensional (assuming that `dims = 1`) data provided by the arrays `x` and `y` and then stores the results in the array `z`. These routines represent a special version of the operation, which expects that the first operand vector was set on the task construction stage.

Parameters of the operation are read from the task descriptor created previously by a corresponding `NewTaskX1D` constructor and pointed to by `task`. If `task` is `NULL`, no operation is done.

Using this form of execution routines is recommended when you need to compute multiple one-dimensional convolutions or correlations with the same data vector in array `x` against different vectors in array `y`. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

Input Parameters

Name	Type	Description
<i>task</i>	FORTTRAN: TYPE (VSL_CONV_TASK) for vslsconvexecx1d and vsldconvexecx1d TYPE (VSL_CORR_TASK) for vslscorrexecx1d and vsldcorrexecx1d C: VSLConvTaskPtr for vslsConvExecX1D and vsldConvExecX1D VSLCorrTaskPtr for vslsCorrExecX1D and vsldCorrExecX1D	Pointer to the task descriptor.
<i>x, y</i>	FORTTRAN: REAL (KIND=4) , DIMENSION (*) for vslsconvexecx1d and vslscorrexecx1d REAL (KIND=8) , DIMENSION (*) for vsldconvexecx1d and vsldcorrexecx1d C: float[] for vslsConvExecX1D and vslsCorrExecX1D double[] for vsldConvExeX1D and vsldCorrExecX1D	Pointer to array containing input data (for the second operand vector). See Data Allocation for more information.
<i>ystride, zstride</i>	FORTTRAN: INTEGER C: int	Strides for input and output data. For more information, see stride parameters .

Output Parameters

Name	Type	Description
<i>z</i>	FORTTRAN: REAL (KIND=4) , DIMENSION(*) for vslsconvexecx1d and vslscorreexecx1d REAL (KIND=8) , DIMENSION(*) for vsldconvexecx1d and vsldcorreexec1d C: float[] for vslsConvExecX1D and vslsCorrExecX1D double[] for vsldConvExecX1D and vsldCorrExecX1D	Pointer to the array that stores output data. See Data Allocation for more information.
<i>status</i>	FORTTRAN: INTEGER C: int	Set to VSL_STATUS_OK if the task is executed successfully or set to non-zero error code otherwise.

Task Destructors

Task destructors are routines designed for deleting task objects and deallocating memory.

DeleteTask

Destroys the task object and frees the memory.

Syntax

Fortran:

```
errcode = vslconvdeletetask(task)
errcode = vslcorrdeletetask(task)
```


C:

```
errcode = vslConvDeleteTask(task);  
errcode = vslCorrDeleteTask(task);
```

Description

Given a pointer to a task descriptor, this routine deletes the task descriptor object and frees the memory allocated for the data structure. If the task holds a work memory, the latter is also freed. The task pointer is set to `NULL`.

Note that if by some reason the task was not deleted successfully, the routine returns an error code. This error code has no relation to the task status code and does not change it.



NOTE. You can use the `NULL` task pointer in calls to destructor routines. In this case, the routine will be terminated and no system crash will occur.

Input Parameters

Name	Type	Description
<i>task</i>	FORTTRAN: TYPE (VSL_CONV_TASK) for vslconvdeletetask TYPE (VSL_CORR_TASK) for vslcorrdeletetask C: VSLConvTaskPtr* for vslConvDeleteTask VSLCorrTaskPtr* for vslCorrDeleteTask	Pointer to the task descriptor.

Output Parameters

Name	Type	Description
<i>errcode</i>	FORTTRAN: INTEGER C: int	Contains 0 if the task object is deleted successfully. Contains error code if an error occurred.

Task Copy

The routines are designed for copying convolution and correlation task descriptors.

CopyTask

Copies a descriptor for convolution or correlation task.

Syntax

Fortran:

```
status = vslconvcopytask(newtask, srctask)
```

```
status = vslcorrcopytask(newtask, srctask)
```

C:

```
status = vslConvCopyTask(newtask, srctask);
```

```
status = vslCorrCopyTask(newtask, srctask);
```

Description

If a task object *srctask* already exists, you can use an appropriate *CopyTask* routine to make its copy in *newtask*. After the copy operation, both source and new task objects will become committed (see [Overview](#) for information about task commitment). If the source task was not previously committed, the commitment operation for this task is implicitly invoked before copying starts. If an error occurs during source task commitment, the task stores the error code in the status field. If an error occurs during copy operation, the routine returns a `NULL` pointer instead of a reference to a new task object.

Input Parameters

Name	Type	Description
<i>srctask</i>	FORTTRAN: <code>TYPE(VSL_CONV_TASK)</code> for <code>vslconvcopytask</code> <code>TYPE(VSL_CORR_TASK)</code> for <code>vslcorrcopytask</code>	Pointer to the source task descriptor.

Name	Type	Description
	C: VSLConvTaskPtr for vslConvCopyTask	
	VSLCorrTaskPtr for vslCorrCopyTask	

Output Parameters

Name	Type	Description
<i>newtask</i>	FORTTRAN: TYPE (VSL_CONV_TASK) for vslconvcopytask TYPE (VSL_CORR_TASK) for vslcorrcopytask C: VSLConvTaskPtr* for vslConvCopyTask VSLCorrTaskPtr* for vslCorrCopyTask	Pointer to the new task descriptor.
<i>status</i>	FORTTRAN: INTEGER C: int	Current status of the source task.

Usage Examples

This section demonstrates how you can use the Intel MKL routines to perform some common convolution and correlation operations both for single threaded and multiple threaded calculations. The following two sample functions `scond1` and `sconf1` simulate the convolution and correlation functions `SCOND` and `SCONF` found in IBM ESSL* library. The functions assume single threaded calculations and can be used with C or C++ compilers.

Example 10-5 Function `scond1` for Single Threaded Calculations

```
#include "mkl_vsl.h"

int scont1(
    float h[], int inch,
    float x[], int incx,
    float y[], int incy,
    int nh, int nx, int iy0, int ny)
{
    int status;
    VSLConvTaskPtr task;
    vslsConvNewTask1D(&task, VSL_CONV_MODE_DIRECT, nh, nx, ny);
    vslConvSetStart(task, &iy0);
    status = vslsConvExec1D(task, h, inch, x, incx, y, incy);
    vslConvDeleteTask(&task);
    return status;
}
```

Example 10-6 Function `sconf1` for Single Threaded Calculations

```
#include "mkl_vsl.h"

int sconf1(
    int init,
    float h[], int inclh,
    float x[], int inclx, int inc2x,
    float y[], int incly, int inc2y,
    int nh, int nx, int m, int iy0, int ny,
    void* aux1, int naux1, void* aux2, int naux2)
{
    int status;

    /* assume that aux1!=0 and naux1 is big enough */
```

```

VSLConvTaskPtr* task = (VSLConvTaskPtr*)aux1;
if (init != 0)
    /* initialization: */
    status = vslsConvNewTaskX1D(task, VSL_CONV_MODE_FFT,
        nh, nx, ny, h, inclh);
if (init == 0) {
    /* calculations: */
    int i;
    vslConvSetStart(*task, &iy0);
    for (i=0; i<m; i++) {
        float* xi = &x[inc2x * i];
        float* yi = &y[inc2y * i];
        /* task is implicitly committed at i==0 */
        status = vslsConvExecX1D(*task, xi, inclx, yi, incly);
    };
};
vslConvDeleteTask(task);
return status;
}

```

Using Multiple Threads

For functions such as `sconf1` described in the previous example, parallel calculations may be more preferable instead of cycling. If `m>1`, you can use multiple threads for invoking the task execution against different data sequences. For such cases, use task copy routines to create `m` copies of the task object before the calculations stage and then run these copies with different threads. Ensure that you make all necessary parameter adjustments for the task (using [Task Editors](#)) before copying it.

The sample code for that can look like following:

```
if (init == 0) {
    int i, status, ss[M];
    VSLConvTaskPtr tasks[M];
    /* assume that M is big enough */
    . . .
    vslConvSetStart(*task, &iy0);
    . . .
    for (i=0; i<m; i++)
        /* implicit commitment at i==0 */
        vslConvCopyTask(&tasks[i],*task);
    . . .
```

Then, *m* threads may be started to execute different copies of the task:

```
. . .
    float* xi = &x[inc2x * i];
    float* yi = &y[inc2y * i];
    ss[i]=vs1sConvExecX1D(tasks[i], xi,inc1x, yi,inc1y);
    . . .
```

And finally, after all threads have finished the calculations, overall status ought to be collected from all task objects. The following code assumes signaling the first error found, if any:

```
. . .
for (i=0; i<m; i++) {
    status = ss[i];
    if (status != 0) /* 0 means "OK" */
        break;
};
return status;
}; /* end if init==0 */
```

Execution routines modify the task internal state (fields of the task structure). Such modifications may conflict with each other if different threads work with the same task object simultaneously. This is the reason why different threads must use different copies of the task.

Mathematical Notation and Definitions

The following notation is necessary to explain the underlying mathematical definitions used in the text:

$\mathbf{R} = (-\infty, +\infty)$	The set of real numbers.
$\mathbf{Z} = \{0, \pm 1, \pm 2, \dots\}$	The set of integer numbers.
$\mathbf{Z}^N = \mathbf{Z} \times \dots \times \mathbf{Z}$	The set of N-dimensional series of integer numbers.
$p = (p_1, \dots, p_N) \in \mathbf{Z}^N$	N-dimensional series of integers.
$u: \mathbf{Z}^N \rightarrow \mathbf{R}$	Function u with arguments from $\mathbf{Z}^N$ and values from $\mathbf{R}$.
$u(p) = u(p_1, \dots, p_N)$	The value of the function u for the argument $(p_1, \dots, p_N)$.
$w = u * v$	Function w is the convolution of the functions u, v .
$w = u \bullet v$	Function w is the correlation of the functions u, v .

Given series $p, q \in \mathbf{Z}^N$:

- series $r = p + q$ is defined as $r^n = p^n + q^n$ for every $n=1, \dots, N$
- series $r = p - q$ is defined as $r^n = p^n - q^n$ for every $n=1, \dots, N$
- series $r = \sup\{p, q\}$ is defines as $r^n = \max\{p^n, q^n\}$ for every $n=1, \dots, N$
- series $r = \inf\{p, q\}$ is defined as $r^n = \min\{p^n, q^n\}$ for every $n=1, \dots, N$
- inequality $p \leq q$ means that $p^n \leq q^n$ for every $n=1, \dots, N$.

A function $u(p)$ is called a finite function if there exist series $p^{\min}, p^{\max} \in \mathbf{Z}^N$ such that:

$$u(p) \neq 0$$

implies

$$p^{\min} \leq p \leq p^{\max}.$$

Operations of convolution and correlation are only defined for finite functions.

Consider functions u, v and series $p^{\min}, p^{\max}, q^{\min}, q^{\max} \in \mathbf{Z}^N$ such that:

$$u(p) \neq 0 \text{ implies } p^{\min} \leq p \leq p^{\max}.$$

$v(q) \neq 0$ implies $Q^{\min} \leq q \leq Q^{\max}$.

Definitions of linear correlation and linear convolution for functions u and v are given below.

Linear Convolution

If function $w = u * v$ is the convolution of u and v , then:

$w(r) \neq 0$ implies $R^{\min} \leq r \leq R^{\max}$,

where $R^{\min} = P^{\min} + Q^{\min}$ and $R^{\max} = P^{\max} + Q^{\max}$.

If $R^{\min} \leq r \leq R^{\max}$, then:

$w(r) = \sum u(t) \cdot v(r-t)$ is the sum for all $t \in \mathbf{Z}^N$ such that $T^{\min} \leq t \leq T^{\max}$,

where $T^{\min} = \sup\{P^{\min}, r - Q^{\max}\}$ and $T^{\max} = \inf\{P^{\max}, r - Q^{\min}\}$.

Linear Correlation

If function $w = u \bullet v$ is the correlation of u and v , then:

$w(r) \neq 0$ implies $R^{\min} \leq r \leq R^{\max}$,

where $R^{\min} = Q^{\min} - P^{\max}$ and $R^{\max} = Q^{\max} - P^{\min}$.

If $R^{\min} \leq r \leq R^{\max}$, then:

$w(r) = \sum u(t) \cdot v(r+t)$ is the sum for all $t \in \mathbf{Z}^N$ such that $T^{\min} \leq t \leq T^{\max}$,

where $T^{\min} = \sup\{P^{\min}, Q^{\min} - r\}$ and $T^{\max} = \inf\{P^{\max}, Q^{\max} - r\}$.

Representation of the functions u , v , w as the input/output data for the Intel MKL convolution and correlation functions is described in the [Data Allocation](#) section below.

Data Allocation

This section explains the relation between:

- mathematical finite functions u , v , w introduced in the section [Mathematical Notation and Definitions](#);

- multi-dimensional input and output data vectors representing the functions u , v , w ;
- arrays u , v , w used to store the input and output data vectors in computer memory

The convolution and correlation routine parameters that determine the allocation of input and output data are the following:

- Data arrays x , y , z
- Shape arrays $xshape$, $yshape$, $zshape$
- Strides within arrays $xstride$, $ystride$, $zstride$
- Parameters $start$, $decimation$

Finite Functions and Data Vectors

The finite functions $u(p)$, $v(q)$, and $w(r)$ introduced above are represented as multi-dimensional vectors of input and output data:

`inputu(i1, ..., idims) for u(p1, ..., pN)`

`inputv(j1, ..., jdims) for v(q1, ..., qN)`

`output(k1, ..., kdims) for w(r1, ..., rN)`.

Parameter *dims* represents the number of dimensions and is equal to N .

The parameters *xshape*, *yshape*, and *zshape* define the shapes of input/output vectors:

`inputu(i1, ..., idims)` is defined if $1 \leq i_n \leq xshape(n)$ for every $n=1, \dots, dims$

`inputv(j1, ..., jdims)` is defined if $1 \leq j_n \leq yshape(n)$ for every $n=1, \dots, dims$

`output(k1, ..., kdims)` is defined if $1 \leq k_n \leq zshape(n)$ for every $n=1, \dots, dims$.

Relation between the input vectors and the functions u and v is defined by the following formulas:

`inputu(i1, ..., idims) = u(p1, ..., pN)`, where $p_n = p_n^{\min} + (i_n - 1)$ for every n

`inputv(j1, ..., jdims) = v(q1, ..., qN)`, where $q_n = q_n^{\min} + (j_n - 1)$ for every n .

Relation between the output vector and the function $w(r)$ is similar (but only in the case when parameters *start* and *decimation* are not defined):

`output(k1, ..., kdims) = w(r1, ..., rN)`, where $r_n = r_n^{\min} + (k_n - 1)$ for every n .

If the parameter *start* is defined, it must belong to the interval $R_n^{\min} \leq start(n) \leq R_n^{\max}$.

If defined, the *start* parameter replaces $R^{\min}$ in the formula:

$\text{output}(k_1, \dots, k_{\text{dims}}) = w(r_1, \dots, r_N)$, where $r_n = \text{start}(n) + (k_n - 1)$

If the parameter *decimation* is defined, it changes the relation according to the following formula:

$\text{output}(k_1, \dots, k_{\text{dims}}) = w(r_1, \dots, r_N)$, where $r_n = R_n^{\min} + (k_n - 1) \cdot \text{decimation}(n)$

If both parameters *start* and *decimation* are defined, the formula is as follows:

$\text{output}(k_1, \dots, k_{\text{dims}}) = w(r_1, \dots, r_N)$, where $r_n = \text{start}(n) + (k_n - 1) \cdot \text{decimation}(n)$

The convolution and correlation software checks the values of *zshape*, *start*, and *decimation* during task commitment. If r_n exceeds $R_n^{\max}$ for some $k_n, n=1, \dots, \text{dims}$, an error is raised.

Allocation of Data Vectors

Both parameter arrays *x* and *y* contain input data vectors in memory, while array *z* is intended for storing output data vector. To access the memory, the convolution and correlation software uses only pointers to these arrays and ignores the array shapes.

For parameters *x*, *y*, and *z*, you can provide one-dimensional arrays with the requirement that actual length of these arrays be sufficient to store the data vectors.

The allocation of the input and output data inside the arrays *x*, *y*, and *z* is described below assuming that the arrays are one-dimensional. Given multi-dimensional indices $i, j, k \in \mathbf{Z}^N$, one-dimensional indices $e, f, g \in \mathbf{Z}$ are defined such that:

$\text{inputu}(i_1, \dots, i_{\text{dims}})$ is allocated at $x(e)$

$\text{inputv}(j_1, \dots, j_{\text{dims}})$ is allocated at $y(f)$

$\text{output}(k_1, \dots, k_{\text{dims}})$ is allocated at $z(g)$.

The indices *e*, *f*, and *g* are defined as follows:

$e = 1 + \sum x_{\text{stride}}(n) \cdot dx(n)$ (the sum is for all $n=1, \dots, \text{dims}$)

$f = 1 + \sum y_{\text{stride}}(n) \cdot dy(n)$ (the sum is for all $n=1, \dots, \text{dims}$)

$g = 1 + \sum z_{\text{stride}}(n) \cdot dz(n)$ (the sum is for all $n=1, \dots, \text{dims}$)

The distances $dx(n)$, $dy(n)$, and $dz(n)$ depend on the signum of the stride:

$dx(n) = i_n - 1$ if $x_{\text{stride}}(n) > 0$, or $dx(n) = i_n - x_{\text{shape}}(n)$ if $x_{\text{stride}}(n) < 0$

$dy(n) = j_n - 1$ if $y_{\text{stride}}(n) > 0$, or $dy(n) = j_n - y_{\text{shape}}(n)$ if $y_{\text{stride}}(n) < 0$

$dz(n) = k_n - 1$ if $zstride(n) > 0$, or $dz(n) = k_n - zshape(n)$ if $zstride(n) < 0$

The definitions of indices e , f , and g assume that indexes for arrays x , y , and z are started from unity:

$x(e)$ is defined for $e=1, \dots, \text{length}(x)$

$y(f)$ is defined for $f=1, \dots, \text{length}(y)$

$z(g)$ is defined for $g=1, \dots, \text{length}(z)$

Below is a detailed explanation about how elements of the multi-dimensional output vector are stored in the array z for one-dimensional and two-dimensional cases.

One-dimensional case. If $dims=1$, then $zshape$ is the number of the output values to be stored in the array z . The actual length of array z may be greater than $zshape$ elements.

If $zstride > 1$, output values are stored with the stride: $\text{output}(1)$ is stored to $z(1)$, $\text{output}(2)$ is stored to $z(1+zstride)$, and so on. Hence, the actual length of z must be at least $1+zstride*(zshape-1)$ elements or more.

If $zstride < 0$, it still defines the stride between elements of array z . However, the order of the used elements is the opposite. For the k -th output value, $\text{output}(k)$ is stored in $z(1+|zstride|*(zshape-k))$, where $|zstride|$ is the absolute value of $zstride$. The actual length of the array z must be at least $1+|zstride|*(zshape - 1)$ elements.

Two-dimensional case. If $dims=2$, the output data is a two-dimensional matrix. The value $zstride(1)$ defines the stride inside matrix columns, that is, the stride between the $\text{output}(k_1, k_2)$ and $\text{output}(k_1+1, k_2)$ for every pair of indices k_1, k_2 . On the other hand, $zstride(2)$ defines the stride between columns, that is, the stride between $\text{output}(k_1, k_2)$ and $\text{output}(k_1, k_2+1)$.

If $zstride(2)$ is greater than $zshape(1)$, this causes sparse allocation of columns. If the value of $zstride(2)$ is smaller than $zshape(1)$, this may result in the transposition of the output matrix. For example, if $zshape = (2, 3)$, you can define $zstride = (3, 1)$ to allocate output values like transposed matrix of the shape 3×2 .

Whether $zstride$ assumes this kind of transformations or not, you need to ensure that different elements $\text{output}(k_1, \dots, k_{dims})$ will be stored in different locations $z(g)$.

Fourier Transform Functions

This chapter describes the following implementations of the fast Fourier transform functions available in Intel® MKL:

- Fast Fourier transform (FFT) functions for single-processor or shared-memory systems (see [FFT Functions](#) below)
- [Cluster FFT functions](#) for distributed-memory architectures (available with Intel® MKL for the Linux* and Windows* OSs only).

The general form of the discrete Fourier transform is

$$Z_{k_1, k_2, \dots, k_d} = \sigma \times \sum_{j_d=0}^{n_d-1} \dots \sum_{j_2=0}^{n_2-1} \sum_{j_1=0}^{n_1-1} w_{j_1, j_2, \dots, j_d} \exp \left(\delta i 2\pi \sum_{l=1}^d j_l k_l / n_l \right)$$

for $k_l = 0, \pm 1, \pm 2, \dots$, where σ is an arbitrary real-valued scale factor and $\delta = \pm 1$. The forward transform is defined by $\sigma = 1$ and $\delta = -1$. In most common situations, the domain of the forward transform, that is, the set where the input (periodic) sequence $\{w_{j_1, j_2, \dots, j_d}\}$ belongs, can be either the set of complex-valued sequences, real-valued sequences, and complex-valued conjugate even sequences.

Both groups of FFT functions above present a uniform and easy-to-use applications programmer interface providing fast computation of discrete Fourier transform via the fast Fourier transform algorithm.

The FFT functions support arbitrary lengths. See the *Intel MKL User's Guide* for the specific radices supported.



NOTE. The FFT functions assume the Cartesian representation of complex data (that is, the real and imaginary parts define a complex number). The Intel MKL Vector Mathematical Functions provide an efficient tool for conversion to and from the polar representation (see [Example C-30a in Appendix C](#) and [Example C-30b in Appendix C](#)).

FFT Functions

The fast Fourier transform function library of Intel MKL provides one-dimensional, two-dimensional, and multi-dimensional (up to the order of 7) functions and both Fortran and C interfaces for all transform functions.

Table 11-1 lists FFT functions implemented in Intel MKL:

Table 11-1 FFT Functions in Intel MKL

Function Name	Operation
Descriptor Manipulation Functions	
<code>DftiCreateDescriptor</code>	Allocates memory for the descriptor data structure and preliminarily initializes it.
<code>DftiCommitDescriptor</code>	Performs all initialization for the actual FFT computation.
<code>DftiCopyDescriptor</code>	Copies an existing descriptor.
<code>DftiFreeDescriptor</code>	Frees memory allocated for a descriptor.
FFT Computation Functions	
<code>DftiComputeForward</code>	Computes the forward FFT.
<code>DftiComputeBackward</code>	Computes the backward FFT.
Descriptor Configuration Functions	
<code>DftiSetValue</code>	Sets one particular configuration parameter with the specified configuration value.
<code>DftiGetValue</code>	Gets the value of one particular configuration parameter.
Status Checking Functions	
<code>DftiErrorClass</code>	Checks if the status reflects an error of a predefined class.
<code>DftiErrorMessage</code>	Generates an error message.

Description of the FFT functions is followed by discussion of configuration settings (see [Configuration Settings](#)) and various configuration parameters used.

Computing FFT

The FFT functions described later in this chapter are provided with the Fortran and C interfaces. Fortran stands for Fortran 95. The FFT interface relies critically on many modern features offered in Fortran 95 that have no counterpart in FORTRAN 77



NOTE. Following the explicit function interface in Fortran, data array must be defined as one-dimensional for any transformation type.

The materials presented in this chapter assume the availability of native complex types in C as they are specified in C9X.

You can find code examples that use FFT interface functions to compute transform results in [Fourier Transform Functions Code Examples](#) section in the Appendix C.

For most common situations, an FFT computation can be effected by four function calls. The approach adopted in Intel MKL for FFT computation uses one single data structure, the descriptor, to record flexible configuration whose parameters can be changed independently. This results in enhanced functionality and ease of use.

The record of type `DFTI_DESCRIPTOR`, when created, contains information about the length and domain of the FFT to be computed, as well as the setting of rather a large number of configuration parameters. The default settings for all of these parameters include, for example, the following:

- the FFT to be computed does not have a scale factor;
- there is only one set of data to be transformed;
- the data is stored contiguously in memory;
- the computed result overwrites (in place) the input data; etc.

Should any one of these many default settings be inappropriate, they can be changed one-at-a-time through the function `DftiSetValue` as illustrated in the [Example C-20](#) and [Example C-21](#).

FFT Interface

To use the FFT functions, you need to access the module `MKL_DFTI` through the "use" statement in Fortran; or access the header file `mkl_dfti.h` through "include" in C.

The Fortran interface provides a derived type `DFTI_DESCRIPTOR`; a number of named constants representing various names of configuration parameters and their possible values; and a number of overloaded functions through the generic functionality of Fortran 95.

The C interface provides a structure type `DFTI_DESCRIPTOR`, a macro definition

```
#define DFTI_DESCRIPTOR_HANDLE DFTI_DESCRIPTOR *
```

a number of named constants of two enumeration types `DFTI_CONFIG_PARAM` and `DFTI_CONFIG_VALUE`; and a number of functions, some of which accept different number of input arguments.



NOTE. Some of the FFT functions and/or functionality described in the subsequent sections of this chapter may not be supported by the currently available implementation of the library. You can find the complete list of the implementation-specific exceptions in the release notes to your version of the library.

There are four main categories of FFT functions in Intel MKL:

- 1. Descriptor Manipulation.** There are four functions in this category. The first one, [DftiCreateDescriptor](#), creates an FFT descriptor whose storage is allocated dynamically. This function configures the descriptor with default settings corresponding to a few input values supplied by the user.

The second, [DftiCommitDescriptor](#), "commits" the descriptor to all its setting. In practice, this usually means that all the necessary precomputation will be performed. This may include factorization of the input length and computation of all the required twiddle factors. The third function, [DftiCopyDescriptor](#), makes an extra copy of a descriptor, and the fourth function, [DftiFreeDescriptor](#), frees up all the memory allocated for the descriptor information.
- 2. FFT Computation.** There are two functions in this category. The first, [DftiComputeForward](#), performs the forward FFT computation, and the second function, [DftiComputeBackward](#), performs a backward FFT computation.
- 3. Descriptor configuration.** There are two functions in this category. One function, [DftiSetValue](#), sets one specific value to one of the many configuration parameters that are changeable (a few are not); the other, [DftiGetValue](#), gets the current value of any one of these configuration parameters (all are readable). These parameters, though many, are handled one-at-a-time.
- 4. Status Checking.** The functions described in the three categories above return an integer value denoting the status of the operation. In particular, a non-zero return value always indicates a problem of some sort. Envisioned to be further enhanced in later releases of Intel MKL, FFT interface at present provides for one logical status class function, [DftiErrorClass](#), and a simple status message generation function, [DftiErrorMessage](#).

Status Checking Functions

All of the descriptor manipulation, FFT computation, and descriptor configuration functions return an integer value denoting the status of the operation. Two functions serve to check the status. The first function is a logical function that checks if the status reflects an error of a predefined class, and the second is an error message function that returns a character string.

ErrorClass

Checks if the status reflects an error of a predefined class.

Syntax

Fortran:

```
Predicate = DftiErrorClass( Status, Error_Class )
```

C:

```
predicate = DftiErrorClass( status, error_class );
```

Description

The FFT interface in Intel MKL provides a set of predefined error class listed in [Table 11-2](#). These are named constants and have the type `INTEGER` in Fortran and `long` in C.

Table 11-2 Predefined Error Class

Named Constants	Comments
<code>DFTI_NO_ERROR</code>	No error
<code>DFTI_MEMORY_ERROR</code>	Usually associated with memory allocation
<code>DFTI_INVALID_CONFIGURATION</code>	Invalid settings of one or more configuration parameters
<code>DFTI_INCONSISTENT_CONFIGURATION</code>	Inconsistent configuration or input parameters
<code>DFTI_NUMBER_OF_THREADS_ERROR</code>	Number of OMP threads in the computation function is not equal to the number of OMP threads in the initialization stage (commit function)

Named Constants	Comments
DFTI_MULTITHREADED_ERROR	Usually associated with a value that OMP routines return in case of errors
DFTI_BAD_DESCRIPTOR	Descriptor is unusable for computation
DFTI_UNIMPLEMENTED	Unimplemented legitimate settings; implementation dependent
DFTI_MKL_INTERNAL_ERROR	Internal library error
DFTI_1D_LENGTH_EXCEEDS_INT32	Length of one of dimensions exceeds $2^{32} - 1$ (4 bytes).

Note that the correct usage is to check if the status returns `.TRUE.` or `.FALSE.` through the use of `DftiErrorClass` with a specific error class. Direct comparison of a status with the predefined class is an incorrect usage. see [Example C-22](#) on a correct use of the status checking functions.

Interface and Prototype

```
//Fortran interface
INTERFACE DftiErrorClass
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
  FUNCTION some_actual_function_8( Status, Error_Class )
    LOGICAL some_actual_function_8
    INTEGER, INTENT(IN) :: Status, Error_Class
  END FUNCTION some_actual_function_8
END INTERFACE DftiErrorClass
```

```
/* C prototype */  
long DftiErrorClass( long , long );
```

ErrorMessage

Generates an error message.

Syntax

Fortran:

```
ERROR_MESSAGE = DftiErrorMessage( Status )
```

C:

```
error_message = DftiErrorMessage( status );
```

Description

The error message function generates an error message character string. The maximum length of the string in Fortran is given by the named constant `DFTI_MAX_MESSAGE_LENGTH`. The actual error message is implementation dependent. In Fortran, the user needs to use a character string of length `DFTI_MAX_MESSAGE_LENGTH` as the target. In C, the function returns a pointer to a character string, that is, a character array with the delimiter '0'.

[Example C-22](#) shows how this function can be implemented.

Interface and Prototype

```
//Fortran interface
INTERFACE DftiErrorMessage
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
FUNCTION some_actual_function_9( Status, Error_Class )
CHARACTER(LEN=DFTI_MAX_MESSAGE_LENGTH) some_actual_function_9( Status )
INTEGER, INTENT(IN) :: Status
END FUNCTION some_actual_function_9
END INTERFACE DftiErrorMessage

/* C prototype */
char *DftiErrorMessage( long );
```

Descriptor Manipulation Functions

There are four functions in this category: create a descriptor, commit a descriptor, copy a descriptor, and free a descriptor.

CreateDescriptor

Allocates memory for the descriptor data structure and preliminarily initializes it.

Syntax

Fortran:

```
Status = DftiCreateDescriptor( Desc_Handle, &
Precision, &
Forward_Domain, &
Dimension, &
Length )
```

C:

```
status = DftiCreateDescriptor( &desc_handle, precision, forward_domain,
dimension, length );
```

Description

This function allocates memory for the descriptor data structure and instantiates it with all the default configuration settings with respect to the precision, domain, dimension, and length of the desired transform. The domain is understood to be the domain of the forward transform. Since memory is allocated dynamically, the result is actually a pointer to the created descriptor. This function is slightly different from the "initialization" function in more traditional software packages or libraries used for computing FFT. This function does not perform any significant computation work such as twiddle factors computation. The function [DftiCommitDescriptor](#) does this work after the function [DftiSetValue](#) has set values of all needed parameters.

The precision and (forward) domain are specified through named constants provided in the FFT interface for the configuration values. The choices for precision are `DFTI_SINGLE` and `DFTI_DOUBLE`; and the choices for (forward) domain are `DFTI_COMPLEX` and `DFTI_REAL`. See [Table 11-5](#) for the complete table of named constants for configuration values.

Dimension is a simple positive integer indicating the dimension of the transform. Length is either a simple positive integer for one-dimensional transform, or an integer array (pointer in C) containing the positive integers corresponding to the lengths dimensions for multi-dimensional transform.

The function returns `DFTI_NO_ERROR` when completes successfully. See [Status Checking Functions](#) for more information on returned status.

Interface and Prototype

```
!Fortran interface.
INTERFACE DftiCreateDescriptor
!Note that the body provided here is to illustrate the different
!argument list and types of dummy arguments. The interface
!does not guarantee what the actual function names are.
!Users can only rely on the function name following the keyword INTERFACE
  FUNCTION some_actual_function_1D( Desc_Handle, Prec, Dom, Dim, Length )
    INTEGER :: some_actual_function_1D

    TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
    INTEGER, INTENT(IN) :: Prec, Dom
    INTEGER, INTENT(IN) :: Dim, Length
  END FUNCTION some_actual_function_1D

  FUNCTION some_actual_function_HIGHD( Desc_Handle, Prec, Dom, Dim, Length )
    INTEGER :: some_actual_function_HIGHD
    TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
    INTEGER, INTENT(IN) :: Prec, Dom
    INTEGER, INTENT(IN) :: Dim, Length(*)
  END FUNCTION some_actual_function_HIGHD
END INTERFACE DftiCreateDescriptor
```

Note that the function is overloaded as the actual argument for `Length` can be a scalar or a rank-one array.

```
/* C prototype */  
long DftiCreateDescriptor( DFTI_DESCRIPTOR_HANDLE *,  
    DFTI_CONFIG_PARAM ,  
    DFTI_CONFIG_PARAM ,  
    long , ... );
```

The variable arguments facility is used to cope with the argument for lengths that can be a scalar (`long`), or an array (`long *`).

CommitDescriptor

Performs all initialization for the actual FFT computation.

Syntax

Fortran:

```
Status = DftiCommitDescriptor( Desc_Handle )
```

C:

```
status = DftiCommitDescriptor( desc_handle );
```

Description

The interface requires a function that commits a previously created descriptor be invoked before the descriptor can be used for FFT computations. Typically, this committal performs all initialization that facilitates the actual FFT computation. For a modern implementation, it may involve exploring many different factorizations of the input length to search for highly efficient computation method.

Any changes of configuration parameters of a committed descriptor via the set value function (see [Descriptor Configuration](#)) requires a re-committal of the descriptor before a computation function can be invoked. Typically, this committal function call is immediately followed by a computation function call (see [FFT Computation](#)).

The function returns `DFTI_NO_ERROR` when completes successfully. See [Status Checking Functions](#) for more information on returned status.

Interface and Prototype

```
! Fortran interface
INTERFACE DftiCommitDescriptor
!Note that the body provided here is to illustrate the different
!argument list and types of dummy arguments. The interface
!does not guarantee what the actual function names are.
!Users can only rely on the function name following the
!keyword INTERFACE
    FUNCTION some_actual function_1 ( Desc_Handle )
    INTEGER :: some_actual function_1
    TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
    END FUNCTION some_actual function_1
END INTERFACE DftiCommitDescriptor

/* C prototype */
long DftiCommitDescriptor( DFTI_DESCRIPTOR_HANDLE );
```

CopyDescriptor

Copies an existing descriptor.

Syntax

Fortran:

```
Status = DftiCopyDescriptor( Desc_Handle_Original, Desc_Handle_Copy )
```


C:

```
status = DftiCopyDescriptor( desc_handle_original, &desc_handle_copy );
```

Description

This function makes a copy of an existing descriptor and provides a pointer to it. The purpose is that all information of the original descriptor will be maintained even if the original is destroyed via the free descriptor function `DftiFreeDescriptor`.

The function returns `DFTI_NO_ERROR` when completes successfully. See [Status Checking Functions](#) for more information on returned status.

Interface and Prototype

```
! Fortran interface
INTERFACE DftiCopyDescriptor
! Note that the body provided here is to illustrate the different
! argument list and types of dummy arguments. The interface
! does not guarantee what the actual function names are.
! Users can only rely on the function name following the
! keyword INTERFACE
    FUNCTION some_actual_function_2( Desc_Handle_Original,
    Desc_Handle_Copy )
    INTEGER :: some_actual_function_2
    TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle_Original, Desc_Handle_Copy
    END FUNCTION some_actual_function_2
END INTERFACE DftiCopyDescriptor

/* C prototype */
long DftiCopyDescriptor( DFTI_DESCRIPTOR_HANDLE, DFTI_DESCRIPTOR_HANDLE * );
```

FreeDescriptor

Frees memory allocated for a descriptor.

Syntax

Fortran:

```
Status = DftiFreeDescriptor( Desc_Handle )
```

C:

```
status = DftiFreeDescriptor( &desc_handle );
```

Description

This function frees up all memory space allocated for a descriptor.



NOTE. Memory allocation/deallocation inside Intel MKL is managed by Intel MKL Memory Manager. So, even after successful completion of `FreeDescriptor`, the memory space may continue being allocated for the application because the Memory Manager sometimes does not return the memory space to the OS but considers the space free and can reuse it for future memory allocation. See [Example “MKL_FreeBuffers Usage with FFT Functions”](#) in the description of the service function `FreeBuffers` on how to use Intel MKL Memory Manager and actually release memory.

The function returns `DFTI_NO_ERROR` when completes successfully. See [Status Checking Functions](#) for more information on returned status.

Interface and Prototype

```
! Fortran interface
INTERFACE DftiFreeDescriptor
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
    FUNCTION some_actual_function_3( Desc_Handle )
    INTEGER :: some_actual_function_3
    TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
    END FUNCTION some_actual_function_3
END INTERFACE DftiFreeDescriptor

/* C prototype */
long DftiFreeDescriptor( DFTI_DESCRIPTOR_HANDLE * );
```

FFT Computation Functions

There are two functions in this category: compute the forward transform, and compute the backward transform.

ComputeForward

Computes the forward FFT.

Syntax

Fortran:

```
Status = DftiComputeForward( Desc_Handle, X_inout )
Status = DftiComputeForward( Desc_Handle, X_in, X_out )
```

C:

```
status = DftiComputeForward( desc_handle, x_inout );
status = DftiComputeForward( desc_handle, x_in, x_out );
```

Description

As soon as a descriptor is configured and committed successfully, actual computation of the FFT can be performed. The `DftiComputeForward` function computes the forward FFT.

This is the transform using the factor $e^{-i2\pi/n}$. Because of the flexibility in configuration, input data can be represented in various ways as well as output result can be placed differently. Consequently, the number of input parameters as well as their type vary. This variation is accommodated by the generic function facility of Fortran 95. Data and result parameters are all declared as assumed-size rank-1 array `DIMENSION(0:*)`.

The function returns `DFTI_NO_ERROR` when completes successfully. See [Status Checking Functions](#) for more information on returned status.

Interface and Prototype

```
//Fortran interface.
INTERFACE DftiComputeFoward
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
// One argument single precision complex
FUNCTION some_actual_function_4_C( Desc_Handle, X )
INTEGER :: some_actual_function_4_C
TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
COMPLEX, INTENT(INOUT) :: X(*)
END FUNCTION some_actual_function_4_C
// One argument double precision complex
FUNCTION some_actual_function_4_Z( Desc_Handle, X )
INTEGER :: some_actual_function_4_Z
TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
COMPLEX (Kind((0D0,0D0))), INTENT(INOUT) :: X(*)
END FUNCTION some_actual_function_4_Z
// One argument single precision real
FUNCTION some_actual_function_4_R( Desc_Handle, X )
INTEGER :: some_actual_function_4_R
TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
REAL, INTENT(INOUT) :: X(*)
END FUNCTION some_actual_function_4_R
// One argument double precision real
```

```

...
// Two argument single precision complex
...
...

FUNCTION some_actual_function_4_CC( Desc_Handle, X_In, Y_Out )
INTEGER :: some_actual_function_4_CC
TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
COMPLEX, INTENT(IN) :: X_In(*)
COMPLEX, INTENT(OUT) :: Y_Out(*)
END FUNCTION some_actual_function_4_CC
END INTERFACE DftiComputeFoward

/* C prototype */
long DftiComputeForward( DFTI_DESCRIPTOR_HANDLE,
                        void *,
                        ... );

```

The implementations of the FFT interface expect the data be treated as data stored linearly in memory with a regular "stride" pattern (discussed more fully in [Strides](#), see also [3]). The function expects the starting address of the first element. Hence we use the assume-size declaration in Fortran.

The descriptor by itself contains sufficient information to determine exactly how many arguments and of what type should be present. The implementation could use this information to check against possible input inconsistency.

ComputeBackward

Computes the backward FFT.

Syntax

Fortran:

```
Status = DftiComputeBackward( Desc_Handle, X_inout )  
Status = DftiComputeBackward( Desc_Handle, X_in, X_out )
```

C:

```
status = DftiComputeBackward( desc_handle, x_inout );  
status = DftiComputeBackward( desc_handle, x_in, x_out );
```

Description

As soon as a descriptor is configured and committed successfully, actual computation of the FFT can be performed. The `DftiComputeBackward` function computes the backward FFT.

This is the transform using the factor $e^{i2\pi/n}$. Because of the flexibility in configuration, input data can be represented in various ways as well as output result can be placed differently. Consequently, the number of input parameters as well as their type vary. This variation is accommodated by the generic function facility of Fortran 95. Data and result parameters are all declared as assumed-size rank-1 array `DIMENSION(0:*)`. The function returns `DFTI_NO_ERROR` when completes successfully. See [Status Checking Functions](#) for more information on returned status.

Interface and Prototype

```
! Fortran Interface
INTERFACE DftiComputeBackward
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
// One argument single precision complex
FUNCTION some_actual_function_5_C( Desc_Handle, X )
INTEGER :: some_actual_function_5_C
TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
COMPLEX, INTENT(INOUT) :: X(*)
END FUNCTION some_actual_function_5_C
// One argument double precision complex
FUNCTION some_actual_function_5_Z( Desc_Handle, X )
INTEGER :: some_actual_function_5_Z
TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
COMPLEX (Kind((0D0,0D0))), INTENT(INOUT) :: X(*)
END FUNCTION some_actual_function_5_Z
// One argument single precision real
FUNCTION some_actual_function_5_R( Desc_Handle, X )
INTEGER :: some_actual_function_5_R
TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
REAL, INTENT(INOUT) :: X(*)
END FUNCTION some_actual_function_5_R
// One argument double precision real
```



```

...
// Two argument single precision complex
...
...

FUNCTION some_actual_function_5_CC( Desc_Handle, X_In, Y_Out )
INTEGER :: some_actual_function_5_CC
TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
COMPLEX, INTENT(IN) :: X_In(*)
COMPLEX, INTENT(OUT) :: Y_Out(*)

END FUNCTION some_actual_function_5_CC
INTERFACE DftiComputeBackward END INTERFACE
DftiComputeBackward

/* C prototype */
long DftiComputeBackward( DFTI_DESCRIPTOR_HANDLE,
    void *,
    ... );

```

The implementations of the FFT interface expect the data be treated as data stored linearly in memory with a regular "stride" pattern (discussed more fully in [Strides](#), see also [3]). The function expects the starting address of the first element. Hence we use the assume-size declaration in Fortran.

The descriptor by itself contains sufficient information to determine exactly how many arguments and of what type should be present. The implementation could use this information to check against possible input inconsistency.

Descriptor Configuration Functions

There are two functions in this category: the value setting function [DftiSetValue](#) sets one particular configuration parameter to an appropriate value, and the value getting function [DftiGetValue](#) reads the value of one particular configuration parameter. While all configuration parameters are readable, a few of them cannot be set by user. Some of these contain fixed

information of a particular implementation such as version number, or dynamic information, but nevertheless are derived by the implementation during execution of one of the functions. See [Configuration Settings](#) for details.

SetValue

Sets one particular configuration parameter with the specified configuration value.

Syntax

Fortran:

```
Status = DftiSetValue( Desc_Handle, &
Config_Param, &
Config_Val )
```

C:

```
status = DftiSetValue( desc_handle, config_param, config_val );
```

Description

This function sets one particular configuration parameter with the specified configuration value. The configuration parameter is one of the named constants listed in [Table 11-3](#), and the configuration value must have the corresponding type, which can be a named constant or a native type. See [Configuration Settings](#) for details of the meaning of the setting. Note that you can set configuration parameters `DFTI_FORWARD_DOMAIN`, `DFTI_PRECISION`, `DFTI_DIMENSION`, and `DFTI_LENGTHS` only through the [DftiCreateDescriptor](#) function. The `DftiSetValue` function does not change these parameters.

The function returns `DFTI_NO_ERROR` when completes successfully. See [Status Checking Functions](#) for more information on returned status.

Interface and Prototype

```

! Fortran interface
INTERFACE DftiSetValue
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
FUNCTION some_actual_function_6_INTVAL( Desc_Handle, Config_Param, INTVAL )
INTEGER :: some_actual_function_6_INTVAL
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
INTEGER, INTENT(IN) :: INTVAL
END FUNCTION some_actual_function_6_INTVAL

FUNCTION some_actual_function_6_SGLVAL( Desc_Handle, Config_Param, SGLVAL )
INTEGER :: some_actual_function_6_SGLVAL
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
REAL, INTENT(IN) :: SGLVAL
END FUNCTION some_actual_function_6_SGLVAL

FUNCTION some_actual_function_6_DBLVAL( Desc_Handle, Config_Param, DBLVAL )
INTEGER :: some_actual_function_6_DBLVAL
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
REAL (KIND(OD0)), INTENT(IN) :: DBLVAL
END FUNCTION some_actual_function_6_DBLVAL

```

```

FUNCTION some_actual_function_6_INTVEC( Desc_Handle, Config_Param, INTVEC )
INTEGER :: some_actual_function_6_INTVEC
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
INTEGER, INTENT(IN) :: INTVEC(*)
END FUNCTION some_actual_function_6_INTVEC

```

```

FUNCTION some_actual_function_6_CHARS( Desc_Handle, Config_Param, CHARS )
INTEGER :: some_actual_function_6_CHARS
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
CHARACTER(*), INTENT(IN) :: CHARS
END FUNCTION some_actual_function_6_CHARS
END INTERFACE DftiSetValue

```

```

/* C prototype */
long DftiSetValue( DFTI_DESCRIPTOR_HANDLE, DFTI_CONFIG_PARAM , ... );

```

GetValue

Gets the configuration value of one particular configuration parameter.

Syntax

Fortran:

```

Status = DftiGetValue( Desc_Handle, &
Config_Param, &
Config_Val )

```

C:

```
status = DftiGetValue( desc_handle, config_param, &config_val );
```

Description

This function gets the configuration value of one particular configuration parameter. The configuration parameter is one of the named constants listed in [Table 11-3](#) and [Table 11-4](#), and the configuration value is the corresponding appropriate type, which can be a named constant or a native type.

The function returns `DFTI_NO_ERROR` when completes successfully. See [Status Checking Functions](#) for more information on returned status.

Interface and Prototype

```
! Fortran interface
INTERFACE DftiGetValue
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
FUNCTION some_actual_function_7_INTVAL( Desc_Handle, Config_Param, INTVAL )
INTEGER :: some_actual_function_7_INTVAL
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
INTEGER, INTENT(OUT) :: INTVAL
END FUNCTION DFTI_GET_VALUE_INTVAL
```

```
FUNCTION some_actual_function_7_SGLVAL( Desc_Handle, Config_Param, SGLVAL )
INTEGER :: some_actual_function_7_SGLVAL
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
REAL, INTENT(OUT) :: SGLVAL
END FUNCTION some_actual_function_7_SGLVAL
```

```
FUNCTION some_actual_function_7_DBLVAL( Desc_Handle, Config_Param, DBLVAL )
INTEGER :: some_actual_function_7_DBLVAL
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
REAL (KIND(0D0)), INTENT(OUT) :: DBLVAL
END FUNCTION some_actual_function_7_DBLVAL
```

```
FUNCTION some_actual_function_7_INTVEC( Desc_Handle, Config_Param, INTVEC )
INTEGER :: some_actual_function_7_INTVEC
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
INTEGER, INTENT(OUT) :: INTVEC(*)
END FUNCTION some_actual_function_7_INTVEC
```

```
FUNCTION some_actual_function_7_INTPNT( Desc_Handle, Config_Param, INTPNT )
INTEGER :: some_actual_function_7_INTPNT
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
INTEGER, DIMENSION(*), POINTER :: INTPNT
END FUNCTION some_actual_function_7_INTPNT
```

```
FUNCTION some_actual_function_7_CHARS( Desc_Handle, Config_Param, CHARS )
  INTEGER :: some_actual_function_7_CHARS
  Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
  INTEGER, INTENT(IN) :: Config_Param
  CHARACTER(*), INTENT(OUT):: CHARS
END FUNCTION some_actual_function_7_CHARS
END INTERFACE DftiGetValue
```

```
/* C prototype */
long DftiGetValue( DFTI_DESCRIPTOR_HANDLE,
                  DFTI_CONFIG_PARAM ,
                  ... );
```

Configuration Settings

Each of the configuration parameters is identified by a named constant in the MKL_DFTI module. In C, these named constants have the enumeration type DFTI_CONFIG_PARAM. The list of configuration parameters whose values can be set by user is given in Table 11-3; the list of configuration parameters that are read-only is given in Table 11-4. All parameters are readable. Most of these parameters are self-explanatory, while some others are discussed more fully in the description of the relevant functions.

Table 11-3 Settable Configuration Parameters

Named Constants	Value Type	Comments
Most common configurations, no default, must be set explicitly		
DFTI_FORWARD_DOMAIN	Named constant	Domain for the forward transform, settable by DftiCreateDescriptor only
DFTI_PRECISION	Named constant	Precision of computation, settable by DftiCreateDescriptor only

Named Constants	Value Type	Comments
DFTI_DIMENSION	Integer scalar	Dimension of the transform, settable by DftiCreateDescriptor only
DFTI_LENGTHS	Integer scalar/array	Lengths of each dimension, settable by DftiCreateDescriptor only
<i>Common configurations including multiple transform and data representation</i>		
DFTI_NUMBER_OF_TRANSFORMS	Integer scalar	For multiple number of transforms
DFTI_FORWARD_SCALE	Floating-point scalar	Scale factor for forward transform
DFTI_BACKWARD_SCALE	Floating-point scalar	Scale factor for backward transform
DFTI_PLACEMENT	Named constant	Placement of the computation result
DFTI_COMPLEX_STORAGE	Named constant	Storage method, complex domain data
DFTI_REAL_STORAGE	Named constant	Storage method, real domain data
DFTI_CONJUGATE_EVEN_STORAGE	Named constant	Storage method, conjugate even domain data
DFTI_DESCRIPTOR_NAME	Character string	No longer than DFTI_MAX_NAME_LENGTH
DFTI_PACKED_FORMAT	Named constant	Packed format, real domain data
DFTI_NUMBER_OF_USER_THREADS	Integer scalar	Number of user threads employing the same descriptor for FFT computation

Configurations regarding stride of data

Named Constants	Value Type	Comments
DFTI_INPUT_DISTANCE	Integer scalar	Multiple transforms, distance of first elements
DFTI_OUTPUT_DISTANCE	Integer scalar	Multiple transforms, distance of first elements
DFTI_INPUT_STRIDES	Integer array	Stride information of input data
DFTI_OUTPUT_STRIDES	Integer array	Stride information of output data

Advanced configuration

DFTI_ORDERING	Named constant	Scrambling of data order
DFTI_TRANSPOSE	Named constant	Scrambling of dimension

Table 11-4 Read-Only Configuration Parameters

Named Constants	Value Type	Comments
DFTI_COMMIT_STATUS	Name constant	Whether descriptor has been committed
DFTI_VERSION	String	Intel MKL library version number

The configuration parameters are set by various values. Some of these values are specified by native data types such as an integer value (for example, number of simultaneous transforms requested), or a single-precision number (for example, the scale factor one would like to apply on a forward transform).

Other configuration values are discrete in nature (for example, the domain of the forward transform) and are thus provided in the DFTI module as named constants. In C, these named constants have the enumeration type `DFTI_CONFIG_VALUE`. The complete list of named constants used for this kind of configuration values is given in [Table 11-5](#).

Table 11-5 Named Constant Configuration Values

Named Constant	Comments
DFTI_SINGLE	Single precision
DFTI_DOUBLE	Double precision
DFTI_COMPLEX	Complex domain

Named Constant	Comments
DFTI_REAL	Real domain
DFTI_INPLACE	Output overwrites input
DFTI_NOT_INPLACE	Output does not overwrite input
DFTI_COMPLEX_COMPLEX	Storage method (see Storage schemes)
DFTI_REAL_REAL	Storage method (see Storage schemes)
DFTI_COMPLEX_REAL	Storage method (see Storage schemes)
DFTI_REAL_COMPLEX	Storage method (see Storage schemes)
DFTI_COMMITTED	Committal status of a descriptor
DFTI_UNCOMMITTED	Committal status of a descriptor
DFTI_ORDERED	Data ordered in both forward and backward domains
DFTI_BACKWARD_SCRAMBLED	Data scrambled in backward domain (by forward transform)
DFTI_NONE	Used to specify no transposition
DFTI_CCS_FORMAT	Packed format, real data (see Packed formats)
DFTI_PACK_FORMAT	Packed format, real data (see Packed formats)
DFTI_PERM_FORMAT	Packed format, real data (see Packed formats)
DFTI_CCE_FORMAT	Packed format, real data (see Packed formats)
DFTI_VERSION_LENGTH	Number of characters for library version length
DFTI_MAX_NAME_LENGTH	Maximum descriptor name length
DFTI_MAX_MESSAGE_LENGTH	Maximum status message length

Table 11-6 lists the possible values for those configuration parameters that are discrete in nature.

Table 11-6 Settings for Discrete Configuration Parameters

Named Constant	Possible Values
DFTI_PRECISION	DFTI_SINGLE, or DFTI_DOUBLE (no default)
DFTI_FORWARD_DOMAIN	DFTI_COMPLEX, or DFTI_REAL
DFTI_PLACEMENT	DFTI_INPLACE (default), or DFTI_NOT_INPLACE
DFTI_COMPLEX_STORAGE	DFTI_COMPLEX_COMPLEX (default)
DFTI_REAL_STORAGE	DFTI_REAL_REAL (default), or DFTI_REAL_COMPLEX
DFTI_CONJUGATE_EVEN_STORAGE	DFTI_COMPLEX_COMPLEX, or DFTI_COMPLEX_REAL (default)
DFTI_PACKED_FORMAT	DFTI_CCS_FORMAT (default), or DFTI_PACK_FORMAT, or DFTI_PERM_FORMAT, or DFTI_CCE_FORMAT
DFTI_ORDERING	DFTI_ORDERED (default), or DFTI_BACKWARD_SCRAMBLED

Table 11-7 lists the default values of the settable configuration parameters.

Table 11-7 Default Configuration Values of Settable Parameters

Named Constants	Default Value
DFTI_NUMBER_OF_TRANSFORMS	1
DFTI_NUMBER_OF_USER_THREADS	1
DFTI_FORWARD_SCALE	1.0

Named Constants	Default Value
DFTI_BACKWARD_SCALE	1.0
DFTI_PLACEMENT	DFTI_INPLACE
DFTI_COMPLEX_STORAGE	DFTI_COMPLEX_COMPLEX
DFTI_REAL_STORAGE	DFTI_REAL_REAL
DFTI_CONJUGATE_EVEN_STORAGE	DFTI_COMPLEX_REAL
DFTI_PACKED_FORMAT	DFTI_CCS_FORMAT
DFTI_DESCRIPTOR_NAME	no name, string of zero length
DFTI_INPUT_DISTANCE	0
DFTI_OUTPUT_DISTANCE	0
DFTI_INPUT_STRIDES	Tightly packed according to dimension and FFT lengths
DFTI_OUTPUT_STRIDES	Same as above. see Strides for details
DFTI_ORDERING	DFTI_ORDERED
DFTI_TRANSPOSE	DFTI_NONE

Precision of transform

The configuration parameter `DFTI_PRECISION` denotes the floating-point precision in which the transform is to be carried out. A setting of `DFTI_SINGLE` stands for single precision, and a setting of `DFTI_DOUBLE` stands for double precision. The data is meant to be presented in this precision; the computation will be carried out in this precision; and the result will be delivered in this precision. This is one of the four settable configuration parameters that do not have default values. The user must set them explicitly, most conveniently at the call to descriptor creation function [DftiCreateDescriptor](#).

Forward domain of transform

As already mentioned in the introduction to the Fourier Transform Functions chapter, the general form of the discrete Fourier transform is

$$Z_{k_1, k_2, \dots, k_d} = \sigma \times \sum_{j_d=0}^{n_d-1} \dots \sum_{j_2=0}^{n_2-1} \sum_{j_1=0}^{n_1-1} w_{j_1, j_2, \dots, j_d} \exp \left(\delta i 2\pi \sum_{l=1}^d j_l k_l / n_l \right)$$

for $k_l = 0, \pm 1, \pm 2, \dots$, where σ is an arbitrary real-valued scale factor and $\delta = \pm 1$. The forward transform is defined by $\sigma = 1$ and $\delta = -1$. In most common situations, the domain of the forward transform, that is, the set where the input (periodic) sequence $\{w_{j_1, j_2, \dots, j_d}\}$ belongs, can be either the set of complex-valued sequences, real-valued sequences, and complex-valued conjugate even sequences. The configuration parameter `DFTI_FORWARD_DOMAIN` indicates the domain for the forward transform. Note that this implicitly specifies the domain for the backward transform because of mathematical property of the FFT. See [Table 11-8](#) for details.

Table 11-8 Correspondence of Forward and Backward Domain

Forward Domain		Implied Backward Domain
Complex	(<code>DFTI_COMPLEX</code>)	Complex
Real	(<code>DFTI_REAL</code>)	Conjugate Even

On transforms in the real domain, some software packages only offer one "real-to-complex" transform. This in essence omits the conjugate even domain for the forward transform. The forward domain configuration parameter `DFTI_FORWARD_DOMAIN` is the second of four configuration parameters without default value.

Transform dimension and lengths

The dimension of the transform is a positive integer value represented in an integer scalar of `Integer` data type in Fortran and `MKL_LONG` data type in C. For one-dimensional transform, the transform length is specified by a positive integer value represented in an integer scalar of `Integer` data type in Fortran and `MKL_LONG` data type in C. For multi-dimensional (≥ 2) transform, the lengths of each of the dimension is supplied in an integer array (`Integer` data type in Fortran and `MKL_LONG` data type in C). `DFTI_DIMENSION` and `DFTI_LENGTHS` are the remaining two of four configuration parameters without default.

As mentioned, these four configuration parameters do not have default value. They are most conveniently set at the descriptor creation function. They can only be set in the descriptor creation function, and not by the function `DftiSetValue`.

Number of transforms

In some situations, the user may need to perform a number of FFTs of the same dimension and lengths. The most common situation would be to transform a number of one-dimensional data of the same length. This parameter has the default value of 1, and can be set to positive integer value by an `Integer` data type in Fortran and `MKL_LONG` data type in C. Data sets have no common elements. The distance parameter is obligatory if multiple number is more than one.

Scale

The forward transform and backward transform are each associated with a scale factor σ of its own with default value of 1. The user can set one or both of them via the two configuration parameters `DFTI_FORWARD_SCALE` and `DFTI_BACKWARD_SCALE`. For example, for a one-dimensional transform of length n , one can use the default scale of 1 for the forward transform while setting the scale factor for backward transform to be $1/n$, making the backward transform the inverse of the forward transform.

The scale factor configuration parameter should be set by a real floating-point data type of the same precision as the value for `DFTI_PRECISION`.

Placement of result

By default, the computational functions overwrite the input data with the output result. That is, the default setting of the configuration parameter `DFTI_PLACEMENT` is `DFTI_INPLACE`. The user can change that by setting it to `DFTI_NOT_INPLACE`. Data sets have no common elements.

Packed formats

The result of the forward transform (i.e. in the frequency-domain) of real data is represented in several possible packed formats: **Pack**, **Perm**, **CCS**, or **CCE**. The data can be packed due to the symmetry property of the FFT of a real data.

The **CCE** format stores the values of the first half of the output complex conjugate-even signal resulted from the forward FFT. Note that the one-dimensional signal stored in **CCE** format is one complex element longer. For multi-dimensional real transform, $n_1 * n_2 * n_3 * \dots * n_k$ the size of complex matrix in **CCE** format is $(n_1/2+1) * n_2 * n_3 * \dots * n_k$ for Fortran and $n_1 * n_2 * \dots * (n_k/2+1)$ for C.

The **CCS** format looks like the **CCE** format. It is the same format as **CCE** for one-dimensional transform. The **CCS** format is slightly different for multi-dimensional real transform. In **CCS** format, the output samples of the FFT are arranged as shown in [Table 11-9](#) for one-dimensional FFT and in [Table 11-10](#) for two-dimensional FFT.

The **Pack** format is a compact representation of a complex conjugate-symmetric sequence. The disadvantage of this format is that it is not the natural format used by the real FFT algorithms ("natural" in the sense that array is natural for complex FFTs). In **Pack** format, the output samples of the FFT are arranged as shown in [Table 11-9](#) for one-dimensional FFT and in [Table 11-11](#) for two-dimensional FFT.

The **Perm** format is an arbitrary permutation of the **Pack** format for even lengths and one is the same as the **Pack** format for odd lengths. In **Perm** format, the output samples of the FFT are arranged as shown in [Table 11-9](#) for one-dimensional FFT and in [Table 11-12](#) for two-dimensional FFT.

Table 11-9 Packed Format Output Samples

For $n = s*2$

FFT Real	0	1	2	3	...	n-2	n-1	n	n+1
CCS	R_0	0	R_1	I_1	...	$R_{n/2-1}$	$I_{n/2-1}$	$R_{n/2}$	0
Pack	R_0	R_1	I_1	R_2	...	$I_{n/2-1}$	$R_{n/2}$		
Perm	R_0	$R_{n/2}$	R_1	I_1	...	$R_{n/2-1}$	$I_{n/2-1}$		

For $n = s*2 + 1$

FFT Real	0	1	2	3	...	n-4	n-3	n-2	n-1	n	n+1
CCS	R_0	0	R_1	I_1	...	I_{s-2}	R_{s-1}	I_{s-1}	R_s	I_s	
Pack	R_0	R_1	I_1	R_2	...	R_{s-1}	I_{s-1}	R_s	I_s		
Perm	R_0	R_1	I_1	R_2	...	R_{s-1}	I_{s-1}	R_s	I_s		

Note that [Table 11-9](#) uses the following notation for complex data entries:

$$R_j = \text{Re } z_j$$

$$I_j = \text{Im } z_j$$

See also [Table 11-13](#) and [Table 11-14](#).

Table 11-10 CCS Format Output Samples (Two-Dimensional Matrix $(m+2)$ -by- $(n+2)$)

For $m = s*2$, $n = k*2$

$z(1,1)$	0	$\text{RE}z(1,2)$	$\text{IM}z(1,2)$	...	$\text{RE}z(1,k)$	$\text{IM}z(1,k)$	$z(1,k+1)$	0
0	0	0	0	...	0	0	0	0

For $m = s*2, n = k*2$

REz(2,1)	REz(2,2)	REz(2,3)	REz(2,4)	...	REz(2,n-1)	REz(2,n)	n/u*	n/u
IMz(2,1)	IMz(2,2)	IMz(2,3)	IMz(2,4)	...	IMz(2,n-1)	IMz(2,n)	n/u	n/u
...	...	...	...	...	...	...	n/u	n/u
REz(m/2,1)	REz(m/2,2)	REz(m/2,3)	REz(m/2,4)	...	REz(m/2,n-1)	REz(m/2,n)	n/u	n/u
IMz(m/2,1)	IMz(m/2,2)	IMz(m/2,3)	IMz(m/2,4)	...	IMz(m/2,n-1)	IMz(m/2,n)	n/u	n/u
z(m/2+1,1)	0	REz(m/2+1,2)	IMz(m/2+1,2)	...	REz(m/2+1,k)	IMz(m/2+1,k)	z(m/2+1,k+1)	0
0	0	0	0	...	0	0	n/u	n/u

For $m = s*2+1, n = k*2$

z(1,1)	0	REz(1,2)	IMz(1,2)	...	REz(1,k)	IMz(1,k)	z(1,k+1)	0
0	0	0	0	...	0	0	0	0
REz(2,1)	REz(2,2)	REz(2,3)	REz(2,4)	...	REz(2,n-1)	REz(2,n)	n/u	n/u
IMz(2,1)	IMz(2,2)	IMz(2,3)	IMz(2,4)	...	IMz(2,n-1)	IMz(2,n)	n/u	n/u
...	...	...	...	...	...	...	n/u	n/u
REz(s,1)	REz(s,2)	REz(s,3)	REz(s,4)	...	REz(s,n-1)	REz(s,n)	n/u	n/u
IMz(s,1)	IMz(s,2)	IMz(s,3)	IMz(s,4)	...	IMz(s,n-1)	IMz(s,n)	n/u	n/u

For $m = s*2, n = k*2+1$

z(1,1)	0	REz(1,2)	IMz(1,2)	...	IMz(1,k-1)	REz(1,k)	IM z(1,k)
0	0	0	0	...	0	0	0
REz(2,1)	REz(2,2)	REz(2,3)	REz(2,4)	...	REz(2,n-1)	REz(2,n)	n/u*
IMz(2,1)	IMz(2,2)	IMz(2,3)	IMz(2,4)	...	IMz(2,n-1)	IMz(2,n)	n/u
...	...	...	...	...	...	...	n/u

For $m = s*2, n = k*2+1$

REz(m/2,1)	REz(m/2,2)	REz(m/2,3)	REz(m/2,4)	...	REz(m/2,n-1)	REz(m/2,n)	n/u
IMz(m/2,1)	IMz(m/2,2)	IMz(m/2,3)	IMz(m/2,4)	...	IMz(m/2,n-1)	IMz(m/2,n)	n/u
z(m/2+1,1)	0	REz(m/2+1,2)	IMz(m/2+1,2)	...	IMz(m/2+1,k-1)	REz(m/2+1,k)	IMz(m/2+1,k)
0	0	0	0	...	0	0	n/u

For $m = s*2+1, n = k*2+1$

z(1,1)	0	REz(1,2)	IMz(1,2)	...	IMz(1,k-1)	REz(1,k)	IMz(1,k)
0	0	0	0	...	0	0	0
REz(2,1)	REz(2,2)	REz(2,3)	REz(2,4)	...	REz(2,n-1)	REz(2,n)	n/u
IMz(2,1)	IMz(2,2)	IMz(2,3)	IMz(2,4)	...	IMz(2,n-1)	IMz(2,n)	n/u
...	...	...	...	...	...	...	n/u
REz(s,1)	REz(s,2)	REz(s,3)	REz(s,4)	...	REz(s,n-1)	REz(s,n)	n/u
IMz(s,1)	IMz(s,2)	IMz(s,3)	IMz(s,4)	...	IMz(s,n-1)	IMz(s,n)	n/u

* n/u - not used.

Note that in the [Table 11-10](#), $(n+2)$ columns are used for even $n = k*2$, while n columns are used for odd $n = k*2+1$. In the latter case, the first row is

z(1,1) 0 REz(1,2) IMz(1,2) ... REz(1,k) IMz(1,k)

If m is even, the $(m+1)$ -th row is

z(m/2+1,1) 0 REz(m/2+1,2) IMz(m/2+1,2) ... REz(m/2+1,k) IMz(m/2+1,k)

Table 11-11 Pack Format Output Samples (Two-Dimensional Matrix m -by- n)

For $m = s*2$

z(1,1)	REz(1,2)	IMz(1,2)	REz(1,3)	...	IMz(1,k)	z(1,k+1)
REz(2,1)	REz(2,2)	REz(2,3)	REz(2,4)	...	REz(2,n-1)	REz(2,n)
IMz(2,1)	IMz(2,2)	IMz(2,3)	IMz(2,4)	...	IMz(2,n-1)	IMz(2,n)

For m = s*2

...	...	...	...	...	...	...
REz(m/2,1)	REz(m/2,2)	REz(m/2,3)	REz(m/2,4)	...	REz(m/2,n-1)	REz(m/2,n)
IMz(m/2,1)	IMz(m/2,2)	IMz(m/2,3)	IMz(m/2,4)	...	IMz(m/2,n-1)	IMz(m/2,n)
z(m/2+1,1)	REz(m/2+1,2)	IMz(m/2+1,2)	REz(m/2+1,3)	...	IMz(m/2+1,k)	z(m/2+1,k+1)

For m = s*2+1

z(1,1)	REz(1,2)	IMz(1,2)	REz(1,3)	...	IMz(1,k)	z(1,n/2+1)
REz(2,1)	REz(2,2)	REz(2,3)	REz(2,4)	...	REz(2,n-1)	REz(2,n)
IMz(2,1)	IMz(2,2)	IMz(2,3)	IMz(2,4)	...	IMz(2,n-1)	IMz(2,n)
...	...	...	...	...	...	...
REz(s,1)	REz(s,2)	REz(s,3)	REz(s,4)	...	REz(s,n-1)	REz(s,n)
IMz(s,1)	IMz(s,2)	IMz(s,3)	IMz(s,4)	...	IMz(s,n-1)	IMz(s,n)

Table 11-12 Perm Format Output Samples (Two-Dimensional Matrix m -by- n)

For m = s*2

z(1,1)	z(1,k+1)	REz(1,2)	IMz(1,2)	...	REz(1,k)	IMz(1,k)
z(m/2+1,1)	z(m/2+1,k+1)	REz(m/2+1,2)	IMz(m/2+1,2)	...	REz(m/2+1,k)	IMz(m/2+1,k)
REz(2,1)	REz(2,2)	REz(2,3)	REz(2,4)	...	REz(2,n-1)	REz(2,n)
IMz(2,1)	IMz(2,2)	IMz(2,3)	IMz(2,4)	...	IMz(2,n-1)	IMz(2,n)
...	...	...	...	...	...	...
REz(m/2,1)	REz(m/2,2)	REz(m/2,3)	REz(m/2,4)	...	REz(m/2,n-1)	REz(m/2,n)
IMz(m/2,1)	IMz(m/2,2)	IMz(m/2,3)	IMz(m/2,4)	...	IMz(m/2,n-1)	IMz(m/2,n)

For m = s*2+1

z(1,1)	z(1,k+1)	REz(1,2)	IMz(1,2)	...	REz(1,k)	IMz(1,k)

For $m = s*2+1$

REz(2,1)	REz(2,2)	REz(2,3)	REz(2,4)	...	REz(2,n-1)	REz(2,n)
IMz(2,1)	IMz(2,2)	IMz(2,3)	IMz(2,4)	...	IMz(2,n-1)	IMz(2,n)
...	...	...	...	...	...	...
REz(s,1)	REz(s,2)	REz(s,3)	REz(s,4)	...	REz(s,n-1)	REz(s,n)
IMz(s,1)	IMz(s,2)	IMz(s,3)	IMz(s,4)	...	IMz(s,n-1)	IMz(s,n)

Note that in the [Table 11-11](#) and [Table 11-12](#), for even number of columns $n = k*2$, while for odd number of columns $n = k*2+1$ and the first row is

$z(1,1)$ REz(1,2) IMz(1,2) ... REz(1,k) IMz(1,k)

If m is even, the last row in Pack format and the second row in Perm format is

$z(m/2+1,1)$ REz(m/2+1,2) IMz(m/2+1,2) ... REz(m/2+1,k) IMz(m/2+1,k)

The tables for two-dimensional FFT use Fortran-interface conventions. For C-interface specifics in storing packed data, see [Storage schemes](#) section below. See also [Table 11-15](#) and [Table 11-16](#) for examples of Fortran-interface and C-interface formats.

Storage schemes

Each value of the domain, that is, DFTI_COMPLEX, DFTI_REAL, and DFTI_CONJUGATE_EVEN (for the forward as well as the backward transform), determines a subset of the four values for storage schemes: DFTI_COMPLEX_COMPLEX, DFTI_COMPLEX_REAL, DFTI_REAL_COMPLEX, and DFTI_REAL_REAL. In [Table 11-6](#), these subsets correspond to DFTI_COMPLEX_STORAGE, DFTI_REAL_STORAGE, and DFTI_CONJUGATE_EVEN_STORAGE in the first-column, respectively. Specific examples are presented below to illustrate the storage schemes that are admissible for each domain. See document [3] for the rationale behind this definition of the storage schemes.



NOTE. The data is stored in the Fortran style only, that is, the real and imaginary parts are stored side by side.

Storage scheme for complex domain. This setting is recorded in the configuration parameter DFTI_COMPLEX_STORAGE. The only value that can be set is DFTI_COMPLEX_COMPLEX. Consider a one-dimensional n -length transform of the form

$$Z_k = \sum_{j=0}^{n-1} w_j e^{-i2\pi jk/n}, \quad w_j, Z_k \in \mathbb{C}.$$

Assume the stride has default value (unit stride) and `DFTI_PLACEMENT` has the default in-place setting.

DFTI_COMPLEX_COMPLEX storage scheme (by default). A typical usage will be as follows.

```
COMPLEX :: X(0:n-1)
...some other code...
Status = DftiComputeForward( Desc_Handle, X )
```

On input,

$$X(j) = w_j, \quad j = 0, 1, \dots, n-1.$$

On output,

$$X(k) = z_k, \quad k = 0, 1, \dots, n-1.$$

Storage scheme for the real and conjugate even domains. This setting for the storage schemes for these domains is recorded in the configuration parameters `DFTI_REAL_STORAGE` and `DFTI_CONJUGATE_EVEN_STORAGE`. Since a forward real domain corresponds to a conjugate even backward domain, they are considered together. The example uses [one-](#), [two-](#) and [three-dimensional](#) real to conjugate even transforms. In-place computation is assumed whenever possible (that is, when the input data type matches the output data type).

One-Dimensional Transform

Consider a one-dimensional n -length transform of the form

$$Z_k = \sum_{j=0}^{n-1} w_j e^{-i2\pi jk/n}, \quad w_j \in \mathbb{R}, Z_k \in \mathbb{C}.$$

There is a symmetry:

For even n : $z(n/2+i) = \text{conjg}(z(n/2-i))$, $1 \leq i \leq n/2-1$, and moreover $z(0)$ and $z(n/2)$ are real values.

For odd n : $z(m+i) = \text{conj}(z(m-i+1))$, $1 \leq i \leq m$, and moreover $z(0)$ is real value.
 $m = \text{floor}(n/2)$.

Table 11-13 Comparison of the Storage Effects of Complex-to-Complex and Real-to-Complex FFTs for Forward Transform

N=8							
Input Vectors			Output Vectors				
Complex FFT		Real FFT	complex FFT		real FFT		
Complex Data		Real Data	Complex Data		Real Data		
Real	Imaginary		Real	Imaginary	CCS	Pack	Perm
w0	0.000000	w0	z0	0.000000	z0	z0	z0
w1	0.000000	w1	Re(z1)	Im(z1)	0.000000	Re(z1)	z4
w2	0.000000	w2	Re(z2)	Im(z2)	Re(z1)	Im(z1)	Re(z1)
w3	0.000000	w3	Re(z3)	Im(z3)	Im(z1)	Re(z2)	Im(z1)
w4	0.000000	w4	z4	0.000000	Re(z2)	Im(z2)	Re(z2)
w5	0.000000	w5	Re(z3)	-Im(z3)	Im(z2)	Re(z3)	Im(z2)
w6	0.000000	w6	Re(z2)	-Im(z2)	Re(z3)	Im(z3)	Re(z3)
w7	0.000000	w7	Re(z1)	-Im(z1)	Im(z3)	z4	Im(z3)
					z4		
					0.000000		
N=7							
Input Vectors			Output Vectors				

N=7

Complex FFT		Real FFT	complex FFT		real FFT		
Complex Data		Real Data	Complex Data		Real Data		
Real	Imaginary		Real	Imaginary	CCS	Pack	Perm
w0	0.000000	w0	z0	0.000000	z0	z0	z0
w1	0.000000	w1	Re(z1)	Im(z1)	0.000000	Re(z1)	Re(z1)
w2	0.000000	w2	Re(z2)	Im(z2)	Re(z1)	Im(z1)	Im(z1)
w3	0.000000	w3	Re(z3)	Im(z3)	Im(z1)	Re(z2)	Re(z2)
w4	0.000000	w4	Re(z3)	-Im(z3)	Re(z2)	Im(z2)	Im(z2)
w5	0.000000	w5	Re(z2)	-Im(z2)	Im(z2)	Re(z3)	Re(z3)
w6	0.000000	w6	Re(z1)	-Im(z1)	Re(z3)	Im(z3)	Im(z3)
					Im(z3)		

Table 11-14 Comparison of the Storage Effects of Complex-to-Complex and Complex-to-Real FFTs for Backward Transform

N=8

Input Vectors		Output Vectors					
Complex FFT		Real FFT	complex FFT		real FFT		
Complex Data		Real Data	Complex Data		Real Data		
Real	Imaginary		Real	Imaginary	CCS	Pack	Perm

N=8							
w0	0.000000	w0	z0	0.000000	z0	z0	z0
w1	0.000000	w1	Re(z1)	Im(z1)	0.000000	Re(z1)	z4
w2	0.000000	w2	Re(z2)	Im(z2)	Re(z1)	Im(z1)	Re(z1)
w3	0.000000	w3	Re(z3)	Im(z3)	Im(z1)	Re(z2)	Im(z1)
w4	0.000000	w4	z4		Re(z2)	Im(z2)	Re(z2)
w5	0.000000	w5	Re(z3)	-Im(z3)	Im(z2)	Re(z3)	Im(z2)
w6	0.000000	w6	Re(z2)	-Im(z2)	Re(z3)	Im(z3)	Re(z3)
w7	0.000000	w7	Re(z1)	-Im(z1)	Im(z3)	z4	Im(z3)
					z4		
					0.000000		

N=7							
Input Vectors				Output Vectors			
Complex FFT		Real FFT		complex FFT		real FFT	
Complex Data		Real Data		Complex Data		Real Data	
Real	Imaginary			Real	Imaginary	CCS	Pack Perm
w0	0.000000	w0	z0	0.000000	z0	z0	z0
w1	0.000000	w1	Re(z1)	Im(z1)	0.000000	Re(z1)	Re(z1)
w2	0.000000	w2	Re(z2)	Im(z2)	Re(z1)	Im(z1)	Im(z1)
w3	0.000000	w3	Re(z3)	Im(z3)	Im(z1)	Re(z2)	Re(z2)

N=7

w4	0.000000	w4	Re(z3)	-Im(z3)	Re(z2)	Im(z2)	Im(z2)
w5	0.000000	w5	Re(z2)	-Im(z2)	Im(z2)	Re(z3)	Re(z3)
w6	0.000000	w6	Re(z1)	-Im(z1)	Re(z3)	Im(z3)	Im(z3)
					Im(z3)		

Assume that the stride has the default value (unit stride).

This complex conjugate-symmetric vector can be stored in the complex array of size $m+1$ or in the real array of size $2m+2$ or $2m$ depending on packed format.

Two-Dimensional Transform

Each of the real-to-complex functions computes the forward FFT of a two-dimensional real matrix according to the mathematical equation

$$z_{i,j} = \sum_{k=0}^{m-1} \sum_{l=0}^{n-1} t_{k,l} * W_m^{-i*k} * W_n^{-j*l},$$

$$0 \leq i \leq m-1, 0 \leq j \leq n-1$$

$t_{k,l} = \text{cmplx}(r_{k,l}, 0)$, where $r_{k,l}$ is a real input matrix, $0 \leq k \leq m-1, 0 \leq l \leq n-1$. The mathematical result $z_{i,j}, 0 \leq i \leq m-1, 0 \leq j \leq n-1$, is the complex matrix of size (m, n) . Each column is the complex conjugate-symmetric vector as follows:

For even m :

for $0 \leq j \leq n-1$,

$$z(m/2+i, j) = \text{conjg}(z(m/2-i, j)), 1 \leq i \leq m/2-1.$$

Moreover, $z(0, j)$ and $z(m/2, j)$ are real values for $j=0$ and $j=n/2$.

For odd m :

for $0 \leq j \leq n-1$,

$z(s+i,j) = \text{conjg}(z(s-i,j)), 1 \leq i \leq s-1,$

where $s = \text{floor}(m/2)$.

Moreover, $z(0,j)$ are real values for $j=0$ and $j=n/2$.

This mathematical result can be stored in the real two-dimensional array of size:

- $(m+2,n+2)$ (CCS format), or
- (m,n) (Pack or Perm formats), or
- $(2*(m/1+1), n)$ (CCE format, Fortran-interface),
- $((m, 2*(n/2+1)))$ (CCE format, C-interface)

or in the complex two-dimensional array of size:

- $(m/2+1, n)$ (CCE format, Fortran-interface),
- $(m, n/2+1)$ (CCE format, C-interface)

Since the multidimensional array data are arranged differently in Fortran and C (see [Strides](#)), the output array that holds the computational result contains complex conjugate-symmetric columns (for Fortran) or complex conjugate-symmetric rows (for C).

The following tables give examples of output data layout in `Pack` format for a forward two-dimensional real-to-complex FFT of a 6-by-4 real matrix. Note that the same layout is used for the input data of the corresponding backward complex-to-real FFT.

Table 11-15 Fortran-interface Data Layout for a 6-by-4 Matrix

$z(1,1)$	$\text{Re } z(1,2)$	$\text{Im } z(1,2)$	$z(1,3)$
$\text{Re } z(2,1)$	$\text{Re } z(2,2)$	$\text{Re } z(2,3)$	$\text{Re } z(2,4)$
$\text{Im } z(2,1)$	$\text{Im } z(2,2)$	$\text{Im } z(2,3)$	$\text{Im } z(2,4)$
$\text{Re } z(3,1)$	$\text{Re } z(3,2)$	$\text{Re } z(3,3)$	$\text{Re } z(3,4)$
$\text{Im } z(3,1)$	$\text{Im } z(3,2)$	$\text{Im } z(3,3)$	$\text{Im } z(3,4)$
$z(4,1)$	$\text{Re } z(4,2)$	$\text{Im } z(4,2)$	$z(4,3)$

For the above example, the stride array is taken to be (0, 1, 6).

Table 11-16 C-interface Data Layout for a 6-by-4 Matrix

$z(1,1)$	$\text{Re } z(1,2)$	$\text{Im } z(1,2)$	$z(1,3)$
$\text{Re } z(2,1)$	$\text{Re } z(2,2)$	$\text{Im } z(2,2)$	$\text{Re } z(2,3)$
$\text{Im } z(2,1)$	$\text{Re } z(3,2)$	$\text{Im } z(3,2)$	$\text{Im } z(2,3)$
$\text{Re } z(3,1)$	$\text{Re } z(4,2)$	$\text{Im } z(4,2)$	$\text{Re } z(3,3)$
$\text{Im } z(3,1)$	$\text{Re } z(5,2)$	$\text{Im } z(5,2)$	$\text{Im } z(3,3)$
$z(4,1)$	$\text{Re } z(6,2)$	$\text{Im } z(6,2)$	$z(4,3)$

For the second example, the stride array is taken to be (0, 4, 1).

See also [Packed formats](#).

Three-Dimensional Transform

Each of the real-to-complex functions computes the forward FFT of a three-dimensional real matrix according to the mathematical equation

$$Z_{i,j,q} = \sum_{k=0}^{m-1} \sum_{l=0}^{n-1} \sum_{s=0}^{k-1} t_{p,l,s} * W_m^{-i*p} * W_n^{-j*l} * W_k^{-q*s},$$

$$0 \leq i \leq m-1, 0 \leq j \leq n-1, 0 \leq s \leq k-1$$

$t_{p,l,s} = \text{cmplx}(r_{p,l,s}, 0)$, where $r_{p,l,s}$ is a real input matrix, $0 \leq k \leq m-1$, $0 \leq l \leq n-1$, $0 \leq s \leq k-1$. The mathematical result $z_{i,j,q}$, $0 \leq i \leq m-1$, $0 \leq j \leq n-1$, $0 \leq q \leq k-1$ is the complex matrix of size (m, n, k) , which is a complex conjugate-symmetric, or conjugate even, matrix as follows:

$z_{m1,n1,k1} = \text{conjg}(z_{m-m1,n-n1,k-k1})$, where each dimension is periodic.

This mathematical result can be stored in the real three-dimensional array of size:

$(m/2+1, n, k)$ (CCE format, Fortran-interface),

$(m, n, k/2+1)$ (CCE format, C-interface).

Since the multidimensional array data are arranged differently in Fortran and C (see [Strides](#)), the output array that holds the computational result contains complex conjugate-symmetric columns (for Fortran) or complex conjugate-symmetric rows (for C).



NOTE. There is one packed format for 3D REAL FFT - CCE format. In both in-place and out-of-place REAL FFT, for real data, the stride and distance parameters are in `REAL` units and for complex data, they are in `COMPLEX` units. So, elements of input and output data can be placed in different elements of input-output array of the in-place FFT.

1. `DFTI_REAL_REAL` for real domain, **`DFTI_COMPLEX_REAL`** for conjugate even domain (by default). It is used for 1D and 2D REAL FFT.

- A typical usage of in-place transform is as follows:

```
// m = floor( n/2 )
REAL :: X(0:2*m+1)
...some other code...
...assuming inplace transform...
Status = DftiComputeForward( Desc_Handle, X )
```

On input,

$X(j) = w_j, j = 0, 1, \dots, n-1.$

On output,

Output data stored in one of formats: `Pack`, `Perm` or `CCS` (see [Packed formats](#)).

CCS format: $X(2*k) = \text{Re}(z_k), X(2*k+1) = \text{Im}(z_k), k = 0, 1, \dots, m.$

Pack format:

even n: $X(0) = \text{Re}(z_0), X(2*k-1) = \text{Re}(z_k), X(2*k) = \text{Im}(z_k), k = 1, \dots, m-1,$
and $X(n-1) = \text{Re}(z_m)$

odd n: $X(0) = \text{Re}(z_0), X(2*k-1) = \text{Re}(z_k), X(2*k) = \text{Im}(z_k), k = 1, \dots, m$

Perm format:

even n: $X(0) = \text{Re}(z_0), X(1) = \text{Re}(z_m), X(2*k) = \text{Re}(z_k), X(2*k+1) = \text{Im}(z_k),$
 $k = 1, \dots, m-1,$

odd n: $X(0) = \text{Re}(z_0), X(2*k-1) = \text{Re}(z_k), X(2*k) = \text{Im}(z_k), k = 1, \dots, m.$

See [Example C-16](#), [Example C-17](#), [Example C-18](#), and [Example C-19](#).

Input and output data exchange the roles in the backward transform.

- A typical usage of out-of-place transform is as follows:

```
// m = floor( n/2 )
REAL :: X(0:n-1)
REAL :: Y(0:2*m+1)
...some other code...
...assuming out-of-place transform...
Status = DftiComputeForward( Desc_Handle, X, Y )
```

On input, $X(j) = w_j$, $j = 0, 1, \dots, n-1$.

On output,

Output data stored in one of formats: `Pack`, `Perm` or `CCS` (see [Packed formats](#)).

CCS format: $Y(2*k) = \text{Re}(z_k)$, $Y(2*k+1) = \text{Im}(z_k)$, $k = 0, 1, \dots, m$.

Pack format:

even n: $Y(0) = \text{Re}(z_0)$, $Y(2*k-1) = \text{Re}(z_k)$, $Y(2*k) = \text{Im}(z_k)$, $k = 1, \dots, m-1$,
and $Y(n-1) = \text{Re}(z_m)$

odd n: $Y(0) = \text{Re}(z_0)$, $Y(2*k-1) = \text{Re}(z_k)$, $Y(2*k) = \text{Im}(z_k)$, $k = 1, \dots, m$

Perm format:

even n: $Y(0) = \text{Re}(z_0)$, $Y(1) = \text{Re}(z_m)$, $Y(2*k) = \text{Re}(z_k)$, $Y(2*k+1) = \text{Im}(z_k)$,
 $k = 1, \dots, m-1$,

odd n: $Y(0) = \text{Re}(z_0)$, $Y(2*k-1) = \text{Re}(z_k)$, $Y(2*k) = \text{Im}(z_k)$, $k = 1, \dots, m$.

Notice that if the stride of the output array is not set to the default value unit stride, the real and imaginary parts of one complex element will be placed with this stride.

For example:

CCS format: $Y(2*k*s) = \text{Re}(z_k)$, $Y((2*k+1)*s) = \text{Im}(z_k)$, $k = 0, 1, \dots, m$, s - stride.

See [Example C-16a](#) and [Example C-17a](#).

Input and output data exchange the roles in the backward transform.

2. DFTI_REAL_REAL for real domain, **DFTI_COMPLEX_COMPLEX** for conjugate even domain. It is used for 1D, 2D and 3D REAL FFT. The CCE format is set by default. You must explicitly set the storage scheme in this case, because its value is not the default one.

- A typical usage of in-place transform is as follows:

```
// m = floor( n/2 )

REAL :: X(0:m*2)

...some other code...

...assuming in-place transform...

Status = DftiSetValue( Desc_Handle, DFTI_CONJUGATE_EVEN_STORAGE, DFTI_COMPLEX_COMPLEX)

...

Status = DftiComputeForward( Desc_Handle, X)
```

On input,

$X(j) = w_j, j = 0, 1, \dots, n-1.$

On output,

$X(2*k) = \text{Re}(z_k), X(2*k+1) = \text{Im}(z_k), k = 0, 1, \dots, m.$

See [Example C-24](#).

Input and output data exchange the roles in the backward transform.

- A typical usage of out-of-place transform is as follows:

```
// m = floor( n/2 )

REAL :: X(0:n-1)

COMPLEX :: Y(0:m)

...some other code...

...assuming out-of-place transform...

Status = DftiSetValue( Desc_Handle, DFTI_CONJUGATE_EVEN_STORAGE, DFTI_COMPLEX_COMPLEX)

...

Status = DftiComputeForward( Desc_Handle, X, Y )
```

On input,

$X(j) = w_j, j = 0, 1, \dots, n-1.$

On output,

$$Y(k) = z_k, \quad k = 0, 1, \dots, m.$$

See [Example C-24a](#) and [Example C-25](#)

Input and output data exchange the roles in the backward transform.

- 3. DFTI_REAL_COMPLEX** for real domain, **DFTI_COMPLEX_COMPLEX** for conjugate even domain. It is not used in the current version. See [Note in the “FFT Interface” section](#) for details. A typical usage is as follows:

```
// m = floor( n/2 )  
COMPLEX :: X(0:m)  
...some other code...  
...inplace transform...  
Status = DftiComputeForward( Desc_Handle, X )
```

On input,

$$X(j) = w_j, \quad j = 0, 1, \dots, n-1.$$

That is, the imaginary parts of $X(j)$ are zero.

On output,

$$Y(k) = z_k, \quad k = 0, 1, \dots, m,$$

where m is `floor( n/2 )`.

Number of user threads

Customer application can be parallelized by using the following techniques:

- a.** You do not create threads in your application but specify the parallel mode within the FFT module of Intel MKL. See *Intel MKL User's Guide* for more information on how to do this. See also [Example C-26 in Appendix C](#).
- b.** You create threads in the application yourself and have each thread perform all stages of FFT implementation, including descriptor initialization, FFT computation, and descriptor deallocation. In this case, each descriptor is used only within its corresponding thread. It is recommended to set single-threaded mode for Intel MKL. See [Example C-27 in Appendix C](#).
- c.** You create threads in the application yourself after initializing all FFT descriptors. This implies that threading is employed for parallel FFT computation only, and the descriptors are released upon return from the parallel region. In this case, each descriptor is used only within its corresponding thread. It is obligatory to explicitly set the single-threaded mode for Intel

MKL, otherwise, the actual number of threads may differ from one, because the `DftiCommitDescriptor` function is not in a parallel region. See [Example C-27a in Appendix C](#).

- d. You create threads in the application yourself after initializing the only FFT descriptor. This implies that threading is employed for parallel FFT computation only, and the descriptor is released upon return from the parallel region. In this case, each thread uses the same descriptor. See [Example C-28 in Appendix C](#).

In cases "a", "b", and "c", listed above, set the parameter `DFTI_NUMBER_OF_USER_THREADS` to 1 (its default value), since each particular descriptor instance is used only in a single thread.

In case "d", you must use the `DftiSetValue()` function to set the `DFTI_NUMBER_OF_USER_THREADS` to the actual number of FFT computation threads, because multiple threads will be using the same descriptor. If this setting is not done, your program will work incorrectly or fail, since the descriptor contains individual data for each thread.



WARNING.

- It is not recommended to simultaneously parallelize your program and employ the Intel MKL internal threading because this will slow down the performance. Note that in case "d" above, FFT computation is automatically initiated in a single-threading mode.
- You must not change the number of threads after the `DftiCommitDescriptor()` function completed FFT initialization.

Input and output distances

FFT interface in Intel MKL allows the computation of multiple number of transforms. Consequently, the user needs to be able to specify the data distribution of these multiple sets of data. This is accomplished by the distance between the first data element of the consecutive data sets. This parameter is obligatory if multiple number is more than one. The parameter is a value of `Integer` data type in Fortran and `MKL_LONG` data type in C. Data sets don't have any common elements. The following example illustrates the specification. Consider computing the forward FFT on three 32-length complex sequences stored in `X(0:31, 1)`, `X(0:31, 2)`, and `X(0:31, 3)`. Suppose the results are to be stored in the locations `Y(0:31, k)`, $k = 1, 2, 3$, of the array `Y(0:63, 3)`. Thus the input distance is 32, while the output distance is 64. Notice that the data and result parameters in computation functions are all declared as

assumed-size rank-1 array `DIMENSION(0:*)`. Therefore two-dimensional array must be transformed to one-dimensional array by `EQUIVALENCE` statement or other facilities of Fortran. Here is the code fragment:

```
Complex :: X_2D(0:31,3), Y_2D(0:63, 3)

Complex :: X(96), Y(192)
Equivalence (X_2D, X)
Equivalence (Y_2D, Y)
.....
Status = DftiCreateDescriptor(Desc_Handle, DFTI_SINGLE,

    DFTI_COMPLEX, 1, 32)
Status = DftiSetValue(Desc_Handle, DFTI_NUMBER_OF_TRANSFORMS, 3)
Status = DftiSetValue(Desc_Handle, DFTI_INPUT_DISTANCE, 32)
Status = DftiSetValue(Desc_Handle, DFTI_OUTPUT_DISTANCE, 64)
Status = DftiSetValue(Desc_Handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE)
Status = DftiCommitDescriptor(Desc_Handle)
Status = DftiComputeForward(Desc_Handle, X, Y)
Status = DftiFreeDescriptor(Desc_Handle)
```

Strides

In addition to supporting transforms of multiple number of datasets, FFT interface supports non-unit stride distribution of data within each data set. The parameter is an array of values of `Integer` data type in Fortran and `MKL_LONG` data type in C. Consider the following situation where a 32-length FFT is to be computed on the sequence x_j , $0 \leq j < 32$. The actual location of these values are in $X(5)$, $X(7)$, ..., $X(67)$ of an array $X(1:68)$. The stride accommodated by FFT interface consists of a displacement from the first element of the data array $L0$, (4 in this case), and a constant distance of consecutive elements $L1$ (2 in this case). Thus for the Fortran array X

$$x_j = X(1 + L0 + L1 * j) = X(5 + L1 * j).$$

This stride vector (4,2) is provided by a length-2 rank-1 integer array:

```
COMPLEX :: X(68)
INTEGER :: Stride(2)
.....
Status = DftiCreateDescriptor(Desc_Handle, DFTI_SINGLE,
    DFTI_COMPLEX, 1, 32)
Stride = (/ 4, 2 /)
Status = DftiSetValue(Desc_Handle, DFTI_INPUT_STRIDES, Stride)
Status = DftiSetValue(Desc_Handle, DFTI_OUTPUT_STRIDES, Stride)
Status = DftiCommitDescriptor(Desc_Handle)
Status = DftiComputeForward(Desc_Handle, X)
Status = DftiFreeDescriptor(Desc_Handle)
```

In general, for a d -dimensional transform, the stride is provided by a $d+1$ -length integer vector ($L0, L1, L2, \dots, Ld$) with the meaning:

$L0$ = displacement from the first array element
 $L1$ = distance between consecutive data elements in the first dimension
 $L2$ = distance between consecutive data elements in the second dimension
 $\dots = \dots$
 Ld = distance between consecutive data elements in the d -th dimension.

A d -dimensional data sequence

$X_{j_1, j_2, \dots, j_d}, 0 \leq j_i \leq J_i, 1 \leq i \leq d$

will be stored in the rank-1 array X by the mapping

$X_{j_1, j_2, \dots, j_d} = X(\text{first index} + L0 + j_1 L1 + j_2 L2 + \dots + j_d Ld).$

For multiple transforms, the value $L0$ applies to the first data sequence, and $Lj, j = 1, 2, \dots, d$ apply to all the data sequences.

In the case of a single one-dimensional sequence, $L1$ is simply the usual stride. The default setting of strides in the general multi-dimensional situation corresponds to the case where the sequences are distributed tightly into the array:

$$L1 = 1, L2 = J_1, L3 = J_1 J_2, \dots, Ld = \prod_{i=1}^{d-1} J_i$$

Both the input data and output data have a stride associated with it. The default is set in accordance with the data to be stored contiguously in memory in a way that is natural to the language.

Note that in case of a real FFT, where different formats are available, the default value is not the one that seems most natural for certain formats. For example, with the *CCE* format, strides are set by default to $L1 = 1$, $L2 = J1$ for a real transform regardless. However, for a complex matrix, slightly over half of the matrix is actually stored, and you should set strides to $L1 = 1$, $L2 = J1/2 + 1$. In case of an *in-place* transform with the *CCE* data format, even for a real array, you should set strides explicitly: $L1 = 1$, $L2 = (J1/2 + 1) * 2$.

See [Example C-23](#) as an illustration of how to use the configuration parameters discussed above. See [Storage schemes](#) and [Packed formats](#) on how to define arrays for different formats.

Ordering

It is well known that a number of FFT algorithms apply an explicit permutation stage that is time consuming [4]. The exclusion of this step is similar to applying FFT to input data whose order is scrambled, or allowing a scrambled order of the FFT results. In applications such as convolution and power spectrum calculation, the order of result or data is unimportant and thus permission of scrambled order is attractive if it leads to higher performance. The following options are available in Intel MKL:

- `DFTI_ORDERED`: Forward transform data ordered, backward transform data ordered (default option).
- `DFTI_BACKWARD_SCRAMBLLED`: Forward transform data ordered, backward transform data scrambled.

[Table 11-17](#) tabulates the effect on this configuration setting.

Table 11-17 Scrambled Order Transform

	<code>DftiComputeForward</code>	<code>DftiComputeBackward</code>
<code>DFTI_ORDERING</code>	Input → Output	Input → Output
<code>DFTI_ORDERED</code>	ordered → ordered	ordered → ordered

	DftiComputeForward	DftiComputeBackward
DFTI_BACKWARD_SCRAMBLED	ordered → scrambled	scrambled → ordered

Note that meaning of the latter two options are "allow scrambled order if practical." There are situations where in fact allowing out of order data gives no performance advantage, and thus an implementation may choose to ignore the suggestion. Strictly speaking, the normal order is also a scrambled order, the trivial one.

Transposition

This is an option that allows for the result of a high-dimensional transform to be presented in a transposed manner. The default setting is `DFTI_NONE` and can be set to `DFTI_ALLOW`. Similar to that of scrambled order, sometimes in higher dimension transform, performance can be gained if the result is delivered in a transposed manner. FFT interface offers an option for the output be returned in a transposed form if performance gain is possible. Since the generic stride specification is naturally suited for representation of transposition, this option allows the strides for the output to be possibly different from those originally specified by the user. Consider an example where a two-dimensional result

$$Y_{j_1, j_2}, \quad 0 \leq j_i < n_i,$$

is expected. Originally the user specified that the result be distributed in the (flat) array Y with generic strides $L_1 = 1$ and $L_2 = n_1$. With the transposition option, the computation may actually return the result into Y with stride $L_1 = n_2$ and $L_2 = 1$. These strides can be obtained from an appropriate inquiry function. Note also that in dimension 3 and above, transposition means an arbitrary permutation of the dimension.

Cluster FFT Functions

This section describes the cluster Fast Fourier Transform (FFT) functions implemented in Intel® MKL.



NOTE. These functions are available only for the Linux* and Windows* operating systems.

The cluster FFT function library was designed to perform fast Fourier transforms on a cluster, that is, a group of computers interconnected via a network. Each computer (node) in the cluster has its own memory and processor(s). Data interchanges between the nodes are provided by the network.

One or more processes may be running in parallel on each cluster node. To organize communication between different processes, the cluster FFT function library uses the Message Passing Interface (MPI). To avoid dependence on a specific MPI implementation (for example, MPICH, Intel® MPI, and others), the library works with MPI via a message-passing library for linear algebra called BLACS.

Cluster FFT functions of Intel MKL provide one-dimensional, two-dimensional, and multi-dimensional (up to the order of 7) functions and both Fortran and C interfaces for all transform functions.

To develop applications using the cluster FFT functions, you should have basic knowledge of and skills in MPI programming.

The interfaces for the Intel MKL cluster FFT functions are similar to the corresponding interfaces for the conventional Intel MKL [FFT functions](#), described earlier in this chapter. Refer there for details not explained in this section.

[Table 11-18](#) lists cluster FFT functions implemented in Intel MKL:

Table 11-18 Cluster FFT Functions in Intel MKL

Function Name	Operation
Descriptor Manipulation Functions	
DftiCreateDescriptorDM	Allocates memory for the descriptor data structure and preliminarily initializes it.
DftiCommitDescriptorDM	Performs all initialization for the actual FFT computation.
DftiFreeDescriptorDM	Frees memory allocated for a descriptor.
FFT Computation Functions	
DftiComputeForwardDM	Computes the forward FFT.
DftiComputeBackwardDM	Computes the backward FFT.
Descriptor Configuration Functions	
DftiSetValueDM	Sets one particular configuration parameter with the specified configuration value.
DftiGetValueDM	Gets the value of one particular configuration parameter.

Computing Cluster FFT

The cluster FFT functions described later in this section are provided with Fortran and C interfaces. Fortran stands for Fortran 95.

Cluster FFT computation is performed by [DftiComputeForwardDM](#) and [DftiComputeBackwardDM](#) functions, called in a program using MPI, which will be referred to as MPI program. After an MPI program starts, a number of processes is created. MPI identifies each process by its rank. The processes are independent of one another and communicate via MPI. A function called in an MPI program is invoked in all the processes. Each process manipulates data according to its rank. Input or output data for a cluster FFT transform is a sequence of complex values. A cluster FFT computation function operates local part of the input data, i.e. some part of the data to be operated in a particular process, as well as generates local part of the output data. While each process performs its part of computations, running in parallel and communicating through MPI, the processes perform the entire FFT computation. FFT computations using the Intel MKL cluster FFT functions is typically effected by a number of steps listed below:

1. Initiate MPI by calling `MPI_Init` in C/C++ or `MPI_INIT` in Fortran (the function must be called prior to calling any FFT function and any MPI function).
2. Allocate memory for the descriptor and create it by calling [DftiCreateDescriptorDM](#).
3. Specify a value(s) of configuration parameters by a call(s) to [DftiSetValueDM](#).
4. Obtain values of configuration parameters needed to create local data arrays; the values are retrieved by calling [DftiGetValueDM](#).
5. Initialize the descriptor for the FFT computation by calling [DftiCommitDescriptorDM](#).
6. Create arrays for local parts of input and output data and fill the local part of input data with values (For more information, see [Distributing Data among Processes](#).)
7. Compute the transform by calling [DftiComputeForwardDM](#) or [DftiComputeBackwardDM](#).
8. Gather local output data into the global array using MPI functions. (This step is optional because you may need to immediately employ the data differently.)
9. Release memory allocated for the descriptor by calling [DftiFreeDescriptorDM](#).
10. Finalize communication through MPI by calling `MPI_Finalize` in C/C++ or `MPI_FINALIZE` in Fortran (the function must be called after the last call to a cluster FFT function and the last call to an MPI function).

Several code examples in the “[Examples for Cluster FFT Functions](#)” section in Appendix C illustrate cluster FFT computations.

Distributing Data among Processes

The Intel MKL cluster FFT functions store all input and output multi-dimensional arrays (matrices) in one-dimensional arrays (vectors). The arrays are stored in the row-major order in C/C++ and in the column-major order in Fortran. For example, a two-dimensional matrix A of size (m, n) is stored in a vector B of size $m*n$ so that

- $B[i*n+j]=A[i][j]$ in C/C++ ($i=0, \dots, m-1, j=0, \dots, n-1$)
- $B(j*m+i)=A(i, j)$ in Fortran ($i=1, \dots, m, j=1, \dots, n$).



NOTE. Order of FFT dimensions is the same as the order of array dimensions in the programming language. For example, a 3-dimensional FFT with `Lengths=(m,n,l)` can be computed over an array `Ar[m][n][l]` in C/C++ or `AR(m,n,l)` in Fortran.

All MPI processes involved in cluster FFT computation operate their own portions of data. These local arrays make up the virtual global array that the fast Fourier transform is applied to. It is your responsibility to properly allocate local arrays (if needed), fill them with initial data and gather resulting data into an actual global array or process the resulting data differently. To be able to do this, see sections below on how the virtual global array is composed of the local ones.

Multi-dimensional transforms

If the dimension of transform is greater than one, the cluster FFT function library splits data in the dimension whose index changes most slowly, so that the parts contain all elements with several consecutive values of this index. It is the first dimension in C and the last one in Fortran. If the global array is two-dimensional, in C, it gives each process several consecutive rows. The term “rows” will be used regardless of the array dimension and programming language. Local arrays are placed in memory allocated for the virtual global array consecutively, in the order determined by process ranks. For example, in case of two processes, during the computation of a three-dimensional transform whose matrix has size $(11,15,12)$, the processes may store local arrays of sizes $(6,15,12)$ and $(5,15,12)$, respectively.

If p is the number of MPI processes and the matrix of a transform to be computed has size (m, n, l) , in C, each MPI process works with local data array of size (m_q, n, l) , where $\sum_{q=0}^{p-1} m_q = m$, $q=0, \dots, p-1$. Local input arrays must contain appropriate parts of the actual global input array, and then local output arrays will contain appropriate parts of the actual global output array. You can figure out which particular rows of the global array the local array must contain

from the following configuration parameters of the cluster FFT interface: `CDFT_LOCAL_NX`, `CDFT_LOCAL_START_X`, and `CDFT_LOCAL_SIZE`. To retrieve values of the parameters, use the `DftiGetValueDM` function:

- `CDFT_LOCAL_NX` specifies how many rows of the global array the current process receives.
- `CDFT_LOCAL_START_X` specifies which row of the global input or output array corresponds to the first row of the local input or output array. If `A` is a global array and `L` is the appropriate local array, then
 - `L[i][j][k]=A[i+cdft_local_start_x][j][k]`, where $i=0, \dots, m_q-1$, $j=0, \dots, n-1$, $k=0, \dots, l-1$ for C/C++
 - `L(i,j,k)=A(i,j,k+cdft_local_start_x-1)`, where $i=1, \dots, m_q$, $j=1, \dots, n$, $k=1, \dots, l$ for Fortran.

Example C-29 in Appendix C shows how the data is distributed among processes for a two-dimensional cluster FFT computation.

One-dimensional transforms

In this case, input and output data are distributed among processes differently and even the numbers of elements stored in a particular process before and after the transform may be different. Each local array stores a segment of consecutive elements of the appropriate global array. Such segment is determined by the number of elements and a shift with respect to the first array element. So, to specify segments of the global input and output arrays that a particular process receives, *four* configuration parameters are needed: `CDFT_LOCAL_NX`, `CDFT_LOCAL_START_X`, `CDFT_LOCAL_OUT_NX`, and `CDFT_LOCAL_OUT_START_X`. Use the `DftiGetValueDM` function to retrieve their values. The meaning of the four configuration parameters depends upon the type of the transform, as shown in Table 11-19:

Table 11-19 Data Distribution Configuration Parameters for 1D Transforms

Meaning of the Parameter	Forward Transform	Backward Transform
Number of elements in input array	CDFT_LOCAL_NX	CDFT_LOCAL_OUT_NX
Elements shift in input array	CDFT_LOCAL_START_X	CDFT_LOCAL_OUT_START_X
Number of elements in output array	CDFT_LOCAL_OUT_NX	CDFT_LOCAL_NX

Meaning of the Parameter	Forward Transform	Backward Transform
Elements shift in output array	<code>CDFT_LOCAL_OUT_START_X</code>	<code>CDFT_LOCAL_START_X</code>

Memory size for local data

The memory size needed for local arrays cannot be just calculated from `CDFT_LOCAL_NX` (`CDFT_LOCAL_OUT_NX`), because the cluster FFT functions sometimes require allocating a little bit more memory for local data than just the size of the appropriate sub-array. The configuration parameter `CDFT_LOCAL_SIZE` specifies the size of the local input and output array in data elements. Each local input and output arrays must have size not less than `CDFT_LOCAL_SIZE*size_of_element`. Note that in the current implementation of the cluster FFT interface, data elements are complex values, consisting of a real and imaginary parts. If you employ a user-defined workspace for in-place transforms (for more information, refer to [Table 11-20](#)), it must have the same size as the local arrays. [Example C-30](#) in Appendix C illustrates how the cluster FFT functions distribute data among processes in case of a one-dimensional FFT computation effected with a user-defined workspace.

Available Auxiliary Functions

If a global input array is located on one MPI process and you want to obtain its local parts or you want to gather the global output array on one MPI process, you can use functions `MKL_CDFT_ScatterData` and `MKL_CDFT_GatherData` to distribute or gather data among processes, respectively. These functions are defined in a file that is delivered with Intel MKL and located in the following subdirectory of the Intel MKL installation directory:
`examples/cdftc/examples_support.c` for C/C++ and
`examples/cdftf/examples_support.f90` for Fortran.

Restriction on Lengths of Transforms

The algorithm that the Intel MKL cluster FFT functions use to distribute data among processes imposes a restriction on lengths of transforms with respect to the number of MPI processes used for the FFT computation:

- For a multi-dimensional transform, lengths of the first two dimensions in C/C++ or of the last two dimensions in Fortran must be not less than the number of MPI processes.
- Length of a one-dimensional transform must be the product of two integers each of which is not less than the number of MPI processes.

Non-compliance with the restriction causes an error `CDFT_SPREAD_ERROR` (refer to [Error Codes](#) for details). To achieve the compliance, you can change the transform lengths and/or the number of MPI processes, which is specified at start of an MPI program. MPI-2 enables changing the number of processes during execution of an MPI program.



NOTE. The best performance of a one-dimensional FFT is reached when the transform length L is a square of an integer number: $L=N^2$.

Cluster FFT Interface



NOTE. The cluster FFT interface implemented in Intel MKL 9.0 is incompatible with previous versions. For details, refer to *Intel MKL Release Notes*.

To use the cluster FFT functions, you need to access the module `MKL_CDFT` through the "use" statement in Fortran; or access the header file `mkl_cdfti.h` through "include" in C/C++.

The Fortran interface provides a derived type `DFTI_DESCRIPTOR_DM`; a number of named constants representing various names of configuration parameters and their possible values; and a number of overloaded functions through the generic functionality of Fortran 95.

The C interface provides a structure type `DFTI_DESCRIPTOR_DM_HANDLE` and a number of functions, some of which accept a different number of input arguments.

To provide communication between parallel processes through MPI, the following include statement must be present in your code:

- Fortran:

```
INCLUDE "mpif.h"
```

(for some MPI versions, "mpif90.h" header may be used instead).

- C/C++:

```
#include "mpi.h"
```

There are three main categories of the cluster FFT functions in Intel MKL:

1. **Descriptor Manipulation.** There are three functions in this category. The `DftiCreateDescriptorDM` function creates an FFT descriptor whose storage is allocated dynamically. The `DftiCommitDescriptorDM` function "commits" the descriptor to all its settings. The `DftiFreeDescriptorDM` function frees up the memory allocated for the descriptor.
2. **FFT Computation.** There are two functions in this category. The `DftiComputeForwardDM` function performs the forward FFT computation, and the `DftiComputeBackwardDM` function performs the backward FFT computation.
3. **Descriptor Configuration.** There are two functions in this category. The `DftiSetValueDM` function sets one specific configuration value to one of the many configuration parameters. The `DftiGetValueDM` function gets the current value of any of these configuration parameters, all of which are readable. These parameters, though many, are handled one at a time.

Descriptor Manipulation Functions

There are three functions in this category: create a descriptor, commit a descriptor, and free a descriptor.

CreateDescriptorDM

Allocates memory for the descriptor data structure and preliminarily initializes it.

Syntax

Fortran:

```
Status = DftiCreateDescriptorDM(comm, handle, v1, v2, dim, size)
Status = DftiCreateDescriptorDM(comm, handle, v1, v2, dim, sizes)
```

C/C++:

```
status = DftiCreateDescriptorDM(comm, &handle, v1, v2, dim, size );
status = DftiCreateDescriptorDM(comm, &handle, v1, v2, dim, sizes );
```

Input Parameters

<i>comm</i>	MPI communicator, e.g. <code>MPI_COMM_WORLD</code> .
<i>v1</i>	Precision of the transform.

<i>v2</i>	Type of the forward domain. Must be <code>DFTI_COMPLEX</code> for the current version.
<i>dim</i>	Dimension of the transform.
<i>size</i>	Length of the transform in a one-dimensional case.
<i>sizes</i>	Lengths of the transform in a multi-dimensional case.

Output Parameters

<i>handle</i>	Pointer to the descriptor handle of transform. If the function completes successfully, the pointer to the created handle is stored in the variable.

Description

This function allocates memory in a particular MPI process for the descriptor data structure and instantiates it with default configuration settings with respect to the precision, domain, dimension, and length of the desired transform. The domain is understood to be the domain of the forward transform. The result is a pointer to the created descriptor. This function is slightly different from the "initialization" function `DftiCommitDescriptorDM` in a more traditional software packages or libraries used for computing the FFT. This function does not perform any significant computation work, such as twiddle factors computation, because the default configuration settings can still be changed using the function `DftiSetValueDM`.

The value of the parameter *v1* is specified through named constants `DFTI_SINGLE` and `DFTI_DOUBLE`. It corresponds to precision of input data, output data, and computation. A setting of `DFTI_SINGLE` indicates single-precision floating-point data type and a setting of `DFTI_DOUBLE` indicates double-precision floating-point data type.

The parameter *dim* is a simple positive integer indicating the dimension of the transform.

In C/C++, for one-dimensional transforms, length is a single integer value of the parameter *size* having type `long`; for multi-dimensional transforms, length is supplied with the parameter *sizes*, which is an array of integers having type `long`. In Fortran, length is an integer or an array of integers.

Return Values

The function returns `DFTI_NO_ERROR` when completes successfully. In this case, the pointer to the created descriptor handle is stored in *handle*. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the [Error Codes](#) section).

Interface and Prototype

```
! Fortran Interface
INTERFACE DftiCreateDescriptorDM
    INTEGER(4) FUNCTION DftiCreateDescriptorDMn(C,H,P1,P2,D,L)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: H
        INTEGER(4) C,P1,P2,D,L(*)
    END FUNCTION
    INTEGER(4) FUNCTION DftiCreateDescriptorDM1(C,H,P1,P2,D,L)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: H
        INTEGER(4) C,P1,P2,D,L
    END FUNCTION
END INTERFACE

/* C/C++ prototype */
long DftiCreateDescriptorDM(MPI_Comm,DFTI_DESCRIPTOR_DM_HANDLE*,
    enum DFTI_CONFIG_VALUE,enum DFTI_CONFIG_VALUE,long,...);
```

CommitDescriptorDM

Performs all initialization for the actual FFT computation.

Syntax

Fortran:

```
Status = DftiCommitDescriptorDM(handle)
```

C/C++:

```
status = DftiCommitDescriptorDM(handle);
```

Input Parameters

handle

The descriptor handle. Must be valid, that is, created in a call to [DftiCreateDescriptorDM](#).

Description

The cluster FFT interface requires a function that commits a previously created descriptor be invoked before the descriptor can be used for FFT computations in a particular MPI process. The [DftiCommitDescriptorDM](#) function performs all initialization that facilitates the actual FFT computation. For the current implementation, it may involve exploring many different factorizations of the input length to search for highly efficient computation method.

Any changes of configuration parameters of a committed descriptor via the set value function (see [Descriptor Configuration](#)) requires a re-committal of the descriptor before a computation function can be invoked. Typically, this committal function is called right before a computation function call (see [FFT Computation](#)).

Return Values

The function returns `DFTI_NO_ERROR` when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the [Error Codes](#) section).

Interface and Prototype

```
! Fortran Interface
INTERFACE DftiCommitDescriptorDM
    INTEGER(4) FUNCTION DftiCommitDescriptorDM(handle);
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: handle
    END FUNCTION
END INTERFACE
```

```
/* C/C++ prototype */
```

```
long DftiCommitDescriptorDM(DFTI_DESCRIPTOR_DM_HANDLE handle);
```

FreeDescriptorDM

Frees memory allocated for a descriptor.

Syntax

Fortran:

```
Status = DftiFreeDescriptorDM(handle)
```

C/C++:

```
status = DftiFreeDescriptorDM(&handle);
```

Input Parameters

<i>handle</i>	The descriptor handle. Must be valid, that is, created in a call to DftiCreateDescriptorDM .

Output Parameters

<i>handle</i>	The descriptor handle. Memory allocated for the handle is released on output.

Description

This function frees up all memory allocated for a descriptor in a particular MPI process. Call the `DftiFreeDescriptorDM` function to delete the descriptor handle. Upon successful completion of `DftiFreeDescriptorDM` the descriptor handle is no longer valid.

Return Values

The function returns `DFTI_NO_ERROR` when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the [Error Codes](#) section).

Interface and Prototype

```
! Fortran Interface
INTERFACE DftiFreeDescriptorDM
  INTEGER(4) FUNCTION DftiFreeDescriptorDM(handle)
    TYPE(DFTI_DESCRIPTOR_DM), POINTER :: handle
  END FUNCTION
END INTERFACE

/* C/C++ prototype */

long DftiFreeDescriptorDM(DFTI_DESCRIPTOR_DM_HANDLE *handle);
```

FFT Computation Functions

There are two functions in this category: compute the forward transform and compute the backward transform.

ComputeForwardDM

Computes the forward FFT.

Syntax

Fortran:

```
Status = DftiComputeForwardDM(handle, in_X, out_X)
Status = DftiComputeForwardDM(handle, in_out_X)
```

C/C++:

```
status = DftiComputeForwardDM(handle, in_X, out_X);
status = DftiComputeForwardDM(handle, in_out_X);
```

Input Parameters

<i>handle</i>	The descriptor handle.
<i>in_X, in_out_X</i>	Local part of input data. Array of complex values. Refer to the Distributing Data among Processes section on how to allocate and initialize the array.

Output Parameters

<i>out_X, in_out_X</i>	Local part of output data. Array of complex values. Refer to the Distributing Data among Processes section on how to allocate the array.

Description

The `DftiComputeForwardDM` function computes the forward FFT. Forward FFT is the transform using the factor $e^{-i2\pi/n}$.

Before you call the function, the valid descriptor, created by `DftiCreateDescriptorDM`, must be configured and committed using the `DftiCommitDescriptorDM` function.

The computation is carried out by calling the `DftiComputeForward` function. So, the functions have very much in common, and details not explicitly mentioned below can be found in the description of `DftiComputeForward`.

Local part of input data, as well as local part of the output data, is an appropriate sequence of complex values (each complex value consists of two real numbers: real part and imaginary part) that a particular process stores. See the [Distributing Data among Processes](#) section for details.

Refer to the [Configuration Settings](#) section for the list of configuration parameters that the descriptor passes to the function.

The configuration parameter `DFTI_PRECISION` determines the precision of input data, output data, and transform: a setting of `DFTI_SINGLE` indicates single-precision floating-point data type and a setting of `DFTI_DOUBLE` indicates double-precision floating-point data type.

The configuration parameter `DFTI_PLACEMENT` informs the function whether the computation should be in-place. If the value of this parameter is `DFTI_INPLACE` (default), you must call the function with two parameters, otherwise you must supply three parameters. If `DFTI_PLACEMENT = DFTI_INPLACE` and three parameters are supplied, then the third parameter is ignored.



CAUTION. Even in case of an out-of-place transform, local array of input data `in_X` may be changed. To save data, make its copy before calling `DftiComputeForwardDM`.

In case of an in-place transform, `DftiComputeForwardDM` dynamically allocates and deallocates a work buffer of the same size as the local input/output array requires.



NOTE. You can specify your own workspace of the same size through the configuration parameter `CDFT_WORKSPACE` to avoid redundant memory allocation.

Return Values

The function returns `DFTI_NO_ERROR` when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the [Error Codes](#) section).

Interface and Prototype

```
! Fortran Interface
INTERFACE DftiComputeForwardDM
    INTEGER(4) FUNCTION DftiComputeForwardDM(h, in_X, out_X)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
        COMPLEX(8), DIMENSION(*) :: in_x, out_X
    END FUNCTION DftiComputeForwardDM
    INTEGER(4) FUNCTION DftiComputeForwardDMi(h, in_out_X)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
        COMPLEX(8), DIMENSION(*) :: in_out_X
    END FUNCTION DftiComputeForwardDMi
    INTEGER(4) FUNCTION DftiComputeForwardDMs(h, in_X, out_X)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
        COMPLEX(4), DIMENSION(*) :: in_x, out_X
    END FUNCTION DftiComputeForwardDMs
    INTEGER(4) FUNCTION DftiComputeForwardDMis(h, in_out_X)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
        COMPLEX(4), DIMENSION(*) :: in_out_X
    END FUNCTION DftiComputeForwardDMis
END INTERFACE

/* C/C++ prototype */
long DftiComputeForwardDM(DFTI_DESCRIPTOR_DM_HANDLE handle, void *in_X,...);
```

ComputeBackwardDM

Computes the backward FFT.

Syntax

Fortran:

```
Status = DftiComputeBackwardDM(handle, in_X, out_X)
```

```
Status = DftiComputeBackwardDM(handle, in_out_X)
```

C/C++:

```
status = DftiComputeBackwardDM(handle, in_X, out_X);
```

```
status = DftiComputeBackwardDM(handle, in_out_X);
```

Input Parameters

handle

The descriptor handle.

in_X, in_out_X

Local part of input data. Array of complex values. Refer to the [Distributing Data among Processes](#) section on how to allocate and initialize the array.

Output Parameters

out_X, in_out_X

Local part of output data. Array of complex values. Refer to the [Distributing Data among Processes](#) section on how to allocate the array.

Description

The `DftiComputeBackwardDM` function computes the backward FFT. Backward FFT is the transform using the factor $e^{i2\pi/n}$.

Before you call the function, the valid descriptor, created by `DftiCreateDescriptorDM`, must be configured and committed using the `DftiCommitDescriptorDM` function.

The computation is carried out by calling the `DftiComputeBackward` function. So, the functions have very much in common, and details not explicitly mentioned below can be found in the description of `DftiComputeBackward`.

Local part of input data, as well as local part of the output data, is an appropriate sequence of complex values (each complex value consists of two real numbers: real part and imaginary part) that a particular process stores. See the [Distributing Data among Processes](#) section for details.

Refer to the [Configuration Settings](#) section for the list of configuration parameters that the descriptor passes to the function.

The configuration parameter `DFTI_PRECISION` determines the precision of input data, output data, and transform: a setting of `DFTI_SINGLE` indicates single-precision floating-point data type and a setting of `DFTI_DOUBLE` indicates double-precision floating-point data type.

The configuration parameter `DFTI_PLACEMENT` informs the function whether the computation should be in-place. If the value of this parameter is `DFTI_INPLACE` (default), you must call the function with two parameters, otherwise you must supply three parameters. If `DFTI_PLACEMENT = DFTI_INPLACE` and three parameters are supplied, then the third parameter is ignored.



CAUTION. Even in case of an out-of-place transform, local array of input data `in_X` may be changed. To save data, make its copy before calling `DftiComputeBackwardDM`.

In case of an in-place transform, `DftiComputeBackwardDM` dynamically allocates and deallocates a work buffer of the same size as the local input/output array requires.



NOTE. You can specify your own workspace of the same size through the configuration parameter `CDFT_WORKSPACE` to avoid redundant memory allocation.

Return Values

The function returns `DFTI_NO_ERROR` when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the [Error Codes](#) section).

Interface and Prototype

! Fortran Interface

```

INTERFACE DftiComputeBackwardDM
    INTEGER(4) FUNCTION DftiComputeBackwardDM(h, in_X, out_X)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
        COMPLEX(8), DIMENSION(*) :: in_x, out_X
    END FUNCTION DftiComputeBackwardDM
    INTEGER(4) FUNCTION DftiComputeBackwardDMi(h, in_out_X)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
        COMPLEX(8), DIMENSION(*) :: in_out_X
    END FUNCTION DftiComputeBackwardDMi
    INTEGER(4) FUNCTION DftiComputeBackwardDMs(h, in_X, out_X)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
        COMPLEX(4), DIMENSION(*) :: in_x, out_X
    END FUNCTION DftiComputeBackwardDMs
    INTEGER(4) FUNCTION DftiComputeBackwardDMis(h, in_out_X)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
        COMPLEX(4), DIMENSION(*) :: in_out_X
    END FUNCTION DftiComputeBackwardDMis
END INTERFACE

```

/* C/C++ prototype */

```

long DftiComputeBackwardDM(DFTI_DESCRIPTOR_DM_HANDLE handle, void *in_X,...);

```

Descriptor Configuration Functions

There are two functions in this category: the value setting function `DftiSetValueDM` sets one particular configuration parameter to an appropriate value, the value getting function `DftiGetValueDM` reads the value of one particular configuration parameter.

Some configuration parameters used by cluster FFT functions originate from the conventional FFT interface (see [Configuration Settings](#) subsection in the “FFT Functions” section for details).

Other configuration parameters are specific to the cluster FFT. Integer values of these parameters have type `long` in C/C++ and `INTEGER(4)` in Fortran. The exact type of the configuration parameters being floating-point scalars is `float` or `double` in C/C++ and `REAL(4)` or `REAL(8)` in Fortran. The configuration parameters whose values are named constants have the `enum` type in C/C++ and `INTEGER` in Fortran. They are defined in the `mk1_cdft.h` header file in C/C++ and `MKL_CDFT` module in Fortran.

Names of the configuration parameters specific to the cluster FFT interface have prefix `CDFT`.

SetValueDM

Sets one particular configuration parameter with the specified configuration value.

Syntax

Fortran:

```
Status = DftiSetValueDM (handle, param, value)
```

C/C++:

```
status = DftiSetValueDM (handle, param, value);
```

Input Parameters

<i>handle</i>	The descriptor handle. Must be valid, that is, created in a call to <code>DftiCreateDescriptorDM</code> .
<i>param</i>	Name of a parameter to be set up in the descriptor handle. See Table 11-20 for the list of available parameters.
<i>value</i>	Value of the parameter.

Description

This function sets one particular configuration parameter with the specified configuration value. The configuration parameter is one of the named constants listed in the table below, and the configuration value must have the corresponding type. See [Configuration Settings](#) for details of the meaning of the setting. See [Table 11-6](#) for possible values of the parameters that are named constants.

Table 11-20 Settable Configuration Parameters

Parameter Name	Data Type	Description	Default Value
DFTI_FORWARD_SCALE	Floating-point scalar	Scale factor of forward transform.	1.0
DFTI_BACKWARD_SCALE	Floating-point scalar	Scale factor of backward transform.	1.0
DFTI_PLACEMENT	Named constant	Placement of the computation result.	DFTI_INPLACE
DFTI_ORDERING	Named constant	Scrambling of data order.	DFTI_ORDERED
DFTI_WORKSPACE	Array of an appropriate type	Auxiliary buffer, a user-defined workspace. Enables saving memory during in-place computations.	NULL (allocate workspace dynamically).

Return Values

The function returns `DFTI_NO_ERROR` when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the [Error Codes](#) section).

Interface and Prototype

```
! Fortran Interface
INTERFACE DftiSetValueDM
    INTEGER(4) FUNCTION DftiSetValueDM(h, p, v)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
        INTEGER(4) :: p, v
    END FUNCTION
    INTEGER(4) FUNCTION DftiSetValueDMd(h, p, v)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
        INTEGER(4) :: p
        REAL(8) :: v
    END FUNCTION
    INTEGER(4) FUNCTION DftiSetValueDMs(h, p, v)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
        INTEGER(4) :: p
        REAL(4) :: v
    END FUNCTION
    INTEGER(4) FUNCTION DftiSetValueDMsw(h, p, v)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
        INTEGER(4) :: p
        COMPLEX(4) :: v(*)
    END FUNCTION
    INTEGER(4) FUNCTION DftiSetValueDMdw(h, p, v)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
        INTEGER(4) :: p
        COMPLEX(8) :: v(*)
    END FUNCTION
```



```
END INTERFACE
```

```
/* C/C++ prototype */
```

```
long DftiSetValueDM(DFTI_DESCRIPTOR_DM_HANDLE handle, int param,...);
```

GetValueDM

Gets the value of one particular configuration parameter.

Syntax

Fortran:

```
Status = DftiGetValueDM(handle, param, value)
```

C/C++:

```
status = DftiGetValueDM(handle, param, &value);
```

Input Parameters

<i>handle</i>	The descriptor handle. Must be valid, that is, created in a call to DftiCreateDescriptorDM .
<i>param</i>	Name of a parameter to be retrieved from the descriptor. See Table 11-21 for the list of available parameters.

Output Parameters

<i>value</i>	Value of the parameter.

Description

This function gets the configuration value of one particular configuration parameter. The configuration parameter is one of the named constants listed in the table below, and the configuration value is the corresponding appropriate type, which can be a named constant or a native type. Possible values of the named constants can be found in [Table 11-6](#).

Table 11-21 Retrievable Configuration Parameters

Parameter Name	Data Type	Description
DFTI_PRECISION	Named constant	Precision of computation, input data and output data.
DFTI_DIMENSION	Integer scalar	Dimension of the transform
DFTI_LENGTHS	Array of integer values	Array of lengths of the transform. Number of lengths corresponds to the dimension of the transform.
DFTI_FORWARD_SCALE	Floating-point scalar	Scale factor of forward transform.
DFTI_BACKWARD_SCALE	Floating-point scalar	Scale factor of backward transform.
DFTI_PLACEMENT	Named constant	Placement of the computation result.
DFTI_COMMIT_STATUS	Named constant	Shows whether descriptor has been committed.
DFTI_FORWARD_DOMAIN	Named constant	Forward domain of transforms (In the current implementation of the cluster FFT interface, the value is always DFTI_COMPLEX.).
DFTI_ORDERING	Named constant	Scrambling of data order.
CDFT_MPI_COMM	Type of MPI communicator	MPI communicator used for transforms.
CDFT_LOCAL_SIZE	Integer scalar	Necessary size of input, output, and buffer arrays in data elements.
CDFT_LOCAL_X_START	Integer scalar	Row/element number of the global array that corresponds to the first row/element of the local array. For more information, see Distributing Data among Processes .

Parameter Name	Data Type	Description
CDFT_LOCAL_NX	Integer scalar	The number of rows/elements of the global array stored in the local array. For more information, see Distributing Data among Processes .
CDFT_LOCAL_OUT_X_START	Integer scalar	Element number of the appropriate global array that corresponds to the first element of the input or output local array in a 1D case. For details, see Distributing Data among Processes .
CDFT_LOCAL_OUT_NX	Integer scalar	The number of elements of the appropriate global array that are stored in the input or output local array in a 1D case. For details, see Distributing Data among Processes .

Return Values

The function returns `DFTI_NO_ERROR` when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the [Error Codes](#) section).

Interface and Prototype

```
! Fortran Interface
INTERFACE DftiGetValueDM
    INTEGER(4) FUNCTION DftiGetValueDM(h, p, v)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
        INTEGER(4) :: p, v
    END FUNCTION
    INTEGER(4) FUNCTION DftiGetValueDMar(h, p, v)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
        INTEGER(4) :: p, v(*)
    END FUNCTION
    INTEGER(4) FUNCTION DftiGetValueDMd(h, p, v)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
        INTEGER(4) :: p
        REAL(8) :: v
    END FUNCTION
    INTEGER(4) FUNCTION DftiGetValueDMs(h, p, v)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
        INTEGER(4) :: p
        REAL(4) :: v
    END FUNCTION
END INTERFACE

/* C/C++ prototype */
long DftiGetValueDM(DFTI_DESCRIPTOR_DM_HANDLE handle, int param,...);
```

Error Codes

All the cluster FFT functions return an integer value denoting the status of the operation. These values are identified by named constants. Each function returns `DFTI_NO_ERROR` if no errors were encountered during execution. Otherwise, a function generates an error code. In addition to FFT error codes, the cluster FFT interface has its own ones. Named constants specific to the cluster FFT interface have prefix “`CDFT`” in names. [Table 11-22](#) lists error codes that the cluster FFT functions may return.

Table 11-22 Error Codes that Cluster FFT Functions Return

Named Constants	Comments
<code>DFTI_NO_ERROR</code>	No error.
<code>DFTI_MEMORY_ERROR</code>	Usually associated with memory allocation.
<code>DFTI_INVALID_CONFIGURATION</code>	Invalid settings of one or more configuration parameters.
<code>DFTI_INCONSISTENT_CONFIGURATION</code>	Inconsistent configuration or input parameters.
<code>DFTI_NUMBER_OF_THREADS_ERROR</code>	Number of OMP threads in the computation function is not equal to the number of OMP threads in the initialization stage (commit function).
<code>DFTI_MULTITHREADED_ERROR</code>	Usually associated with a value that OMP routines return in case of errors.
<code>DFTI_BAD_DESCRIPTOR</code>	Descriptor is unusable for computation.
<code>DFTI_UNIMPLEMENTED</code>	Unimplemented legitimate settings; implementation dependent.
<code>DFTI_MKL_INTERNAL_ERROR</code>	Internal library error.
<code>DFTI_1D_LENGTH_EXCEEDS_INT32</code>	Length of one of dimensions exceeds $2^{32} - 1$ (4 bytes).
<code>CDFT_SPREAD_ERROR</code>	Data cannot be distributed (For more information, see Distributing Data among Processes .)
<code>CDFT_MPI_ERROR</code>	MPI error. Occurs when calling MPI.

Interval Linear Solvers

Intel® MKL Interval Linear solver routines can be used for:

- solving systems of interval linear equations $Ax = b$ with an interval matrix $A = (a_{ij})$ and interval right-hand side vector $b = (b_i)$;
- checking properties of interval matrices.

For more information on key concepts of interval linear systems, see [Appendix A, “Linear Solvers Basics”](#).

Routines described below are subdivided according to the problems they solve into the following groups:

[Routines for Fast Solution of Interval Systems](#)

[Routines for Sharp Solution of Interval Systems](#)

[Routines for Inverting Interval Matrices](#)

[Routines for Checking Properties of Interval Matrices](#)

[Auxiliary and Utility Routines](#)

Table 12-1 contains the full list of Intel MKL routines for solving interval linear systems.

Table 12-1 Intel MKL Interval Linear Solver Routines

Routine Name	Description
?trtrs	Solves a triangular system of interval linear equations by backward substitution procedure.
?gegas	Solves a system of interval linear equations by interval Gauss method.
?gehss	Solves a system of interval linear equations by interval Householder method.
?gekws	Solves a system of interval linear equations by Krawczyk iteration method.
?gegss	Solves a system of interval linear equations by interval Gauss-Seidel iteration.
?gehbs	Solves a system of interval linear equations by Hansen-Bliek-Rohn procedure.
?gepps	Solves a system of interval linear equations by a parameter partitioning method.
?gepss	Solves a system of interval linear equations by a solution partitioning method.
?trtri	Computes inverse interval matrix to a triangular interval matrix.
?geszi	Computes inverse interval matrix by Schulz interval iterative procedure.
?gerbr	Tests regularity of an interval matrix by Ris-Beeck and Rex-Rohn criteria
?gesvr	Tests regularity/singularity of an interval matrix by Rump and Rex-Rohn singular value criteria.
?gemip	Performs midpoint-inverse preconditioning of an interval linear system.

Routine Naming Conventions

For the routines introduced below, the LAPACK-like naming conventions are used. Specifically, all the routine names have the structure `xyyzzz`, where the first letters `xx` indicate the data types:

<code>si</code>	real interval, single precision
<code>di</code>	real interval, double precision
<code>cr</code>	complex rectangular interval, single precision
<code>zr</code>	complex rectangular interval, double precision
<code>cc</code>	complex circular interval, single precision
<code>zc</code>	complex circular interval, double precision

The third and fourth letters `yy` indicate the matrix type:

<code>ge</code>	general
<code>tr</code>	triangular

The last three letters `zzz` indicate the computational procedure performed by the routine:

<code>trs</code>	backward substitution solver for triangular interval linear systems
<code>gas</code>	interval Gauss solver for interval linear systems
<code>hss</code>	interval Householder solver for interval linear systems
<code>kws</code>	iterative Krawczyk solver for interval linear systems
<code>gss</code>	interval Gauss-Seidel iteration solver for interval linear systems
<code>hbs</code>	Hansen-Bliek-Rohn solver for interval linear systems
<code>pps</code>	parameter partitioning method-based solver for interval linear systems
<code>pss</code>	solution partitioning method-based solver for interval linear systems
<code>tri</code>	inverting triangular interval matrix based on backward substitution
<code>szi</code>	inverting general interval matrix by Schulz iterative method
<code>rbr</code>	testing regularity/singularity of interval matrix by Ris-Beeck criterion
<code>svr</code>	testing regularity/singularity of interval matrix by Rump and Rex-Rohn singular value criteria
<code>mip</code>	midpoint-inverse preconditioning of interval linear system

The question mark in the routine group name corresponds to different character codes indicating the data type (`si`, `di`, `cr`, `zr`, `cc`, or `zc`). For example, `?trtri` denotes the group name for either of the routines `sitrtri`, `ditrtri`, `crttrtri`, `zrttrtri`, `cctrtri`, or `zcctrtri`.

Routines for Fast Solution of Interval Systems

?trtrs

Solves a triangular system of interval linear equations by backward substitution procedure.

Syntax

```
call sitrtrs(uplo, trans, diag, n, nrhs, a, lda, b, ldb, info)
call ditrtrs(uplo, trans, diag, n, nrhs, a, lda, b, ldb, info)
call crtrtrs(uplo, trans, diag, n, nrhs, a, lda, b, ldb, info)
call zrtrtrs(uplo, trans, diag, n, nrhs, a, lda, b, ldb, info)
call cctrtrs(uplo, trans, diag, n, nrhs, a, lda, b, ldb, info)
call zctrtrs(uplo, trans, diag, n, nrhs, a, lda, b, ldb, info)
```

Description

The routine ?trtrs solves for X the following systems of interval linear equations with a triangular matrix A and multiple right-hand sides stored in B :

$$A^*X = B, \text{ if } trans = 'N'$$

$$A^T * X = B, \text{ if } trans = 'T'$$

$$A^H * X = B, \text{ if } trans = 'C' \text{ (for complex matrices only).}$$

The routine implements backward substitution algorithm and produces optimal enclosures of the solution sets to interval linear systems, which is due to the simple structure of the matrix A .

Input Parameters

<i>uplo</i>	CHARACTER(1). Must be one of 'U', 'L', 'u', or 'l'. Indicates whether A is upper or lower triangular. If <i>uplo</i> = 'U' or 'u', then A is upper triangular. If <i>uplo</i> = 'L' or 'l', then A is lower triangular.
<i>trans</i>	CHARACTER(1). Must be one of 'N', 'T', 'C', 'n', 't', or 'c'. If <i>trans</i> = 'N' or 'n', then $A^*X = B$ is solved for X .

	If $trans = 'T'$ or $'t'$, then $A^T * X = B$ is solved for X. If $trans = 'C'$ or $'c'$, then $A^H * X = B$ is solved for X.
<i>diag</i>	CHARACTER(1). Must be one of 'N', 'U', 'n', or 'u'. If $diag = 'N'$ or $'n'$, then A is not a unit triangular matrix. If $diag = 'U'$ or $'u'$, then A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array a.
<i>n</i>	INTEGER. The order of A , the number of rows in B ($n \geq 0$).
<i>nrhs</i>	INTEGER. The number of right-hand sides ($nrhs \geq 0$).
<i>a, b</i>	S_INTERVAL for sitrtrs. D_INTERVAL for ditrtrs. CR_INTERVAL for crrtrs. ZR_INTERVAL for zrrtrs. CC_INTERVAL for cctrtrs. ZC_INTERVAL for zcctrtrs. Arrays: $a(la, *)$, $b(lb, *)$. The array a contains the matrix A. The array b contains the matrix B, whose columns are the right-hand sides for the systems of equations. The second dimension of a must be at least $\max(1, n)$ and the second dimension of b must be at least $\max(1, nrhs)$.
<i>lda</i>	INTEGER. The first dimension of a , $lda \geq \max(1, n)$.
<i>ldb</i>	INTEGER. The first dimension of b , $ldb \geq \max(1, nrhs)$.

Output Parameters

<i>b</i>	Overwritten by the solution matrix X .
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info \geq 1$, the execution is not successful. If $info = -i$, the i-th parameter has an illegal value.

?gegas

Solves a system of interval linear equations by interval Gauss method.

Syntax

```
call sigegas(trans, n, nrhs, a, lda, b, ldb, info)
call digegas(trans, n, nrhs, a, lda, b, ldb, info)
call crgegas(trans, n, nrhs, a, lda, b, ldb, info)
call zrgegas(trans, n, nrhs, a, lda, b, ldb, info)
call ccgegas(trans, n, nrhs, a, lda, b, ldb, info)
call zcgegas(trans, n, nrhs, a, lda, b, ldb, info)
```

Description

The routine ?gegas uses the interval Gauss method to compute an enclosure of the solution set to the following interval linear system of equations:

$A^*X = B$, if $trans = 'N'$

$A^T * X = B$, if $trans = 'T'$

$A^H * X = B$, if $trans = 'C'$ (for complex matrices only).

Input Parameters

<i>trans</i>	CHARACTER(1). Must be one of 'N', 'T', 'C', 'n', 't', or 'c'. Indicates the form of the equations system: If $trans = 'N'$ or 'n', then $A^*X = B$ is solved for X . If $trans = 'T'$ or 't', then $A^T * X = B$ is solved for X . If $trans = 'C'$ or 'c', then $A^H * X = B$ is solved for X .
<i>n</i>	INTEGER. The order of A , the number of rows in B ($n \geq 0$).
<i>nrhs</i>	INTEGER. The number of right-hand sides ($nrhs \geq 0$).
<i>a, b</i>	REAL for sigegas. DOUBLE PRECISION for digegas. Arrays: a ($lda, *$), b ($ldb, *$). The array a contains the matrix A .

The array b contains the matrix B , whose columns are the right-hand sides for the systems of equations.

The second dimension of a must be at least $\max(1, n)$ and the second dimension of b must be at least $\max(1, nrhs)$.

lda INTEGER. The first dimension of a , $lda \geq \max(1, n)$.

ldb INTEGER. The first dimension of b , $ldb \geq \max(1, n)$.

Output Parameters

b Overwritten by the solution matrix x .

$info$ INTEGER.

If $info = 0$, the execution is successful.

If $info \geq 0$, the execution is not successful.

If $info = -i$, the i -th parameter has an illegal value.

Example 12-1 Fortran 90 Code for Interval Gauss Method

The following piece of Fortran code presents an example of using the routine `digegas` to compute, by an interval Gauss method, an enclosure of the solution set to the interval linear system of equations:

$$\begin{pmatrix} [2, 3] & [0, 1] \\ [1, 2] & [2, 3] \end{pmatrix} x = \begin{pmatrix} [0, 120] \\ [60, 240] \end{pmatrix}$$

```

-----
. . . . .
USE INTERVAL_ARITHMETIC
. . . . .
TYPE(D_INTERVAL)          :: A(2,2), B(2)
INTEGER                   :: N, INFO
CHARACTER(1)              :: TRANS = 'n'
. . . . .
N = 2
A(1,1) = DINTERVAL(2.,3.);   A(1,2) = DINTERVAL(0.,1.)
A(2,1) = DINTERVAL(1.,2.);   A(2,2) = DINTERVAL(2.,3.)
B(1,1) = DINTERVAL(0.,120.); B(2,1) = DINTERVAL(60.,240.)
. . . . .
CALL DIGEGAS( TRANS, N, 1, A, 2, B, 2, INFO )
-----

```

Note that assigning double-precision intervals to the entries of the matrix *A* and right-hand side vector *B* is carried out by `DINTERVAL` function supplied by `INTERVAL_ARITHMETIC` module.

?gehss

Solves a system of interval linear equations by interval Householder method.

Syntax

```

call sigehss(trans, n, nrhs, a, lda, b, ldb, info)
call digehss(trans, n, nrhs, a, lda, b, ldb, info)

```

Description

The routine `?gehss` uses the interval Householder method to compute an enclosure of the solution set to the following interval linear system of equations:

$$A * X = B, \text{ if } trans = 'N',$$

$A^T * X = B$, if *trans* = 'T' or 'C'.

Input Parameters

<i>trans</i>	CHARACTER(1). Must be one of 'N', 'T', 'C', 'n', 't', or 'c'. Indicates the form of the equations system: If <i>trans</i> = 'N' or 'n', then $A * X = B$ is solved for X . If <i>trans</i> = 'T' or 'C' or 't' or 'c', then $A^T * X = B$ is solved for X .
<i>n</i>	INTEGER. The order of A , the number of rows in B ($n \geq 0$).
<i>nrhs</i>	INTEGER. The number of right-hand sides ($nrhs \geq 0$).
<i>a</i> , <i>b</i>	REAL for <i>sgehss</i> . DOUBLE PRECISION for <i>dgehss</i> . Arrays: <i>a</i> (<i>lda</i> , *), <i>b</i> (<i>ldb</i> , *). The array <i>a</i> contains the matrix A . The array <i>b</i> contains the matrix B , whose columns are the right-hand sides for the systems of equations. The second dimension of <i>a</i> must be at least $\max(1, n)$ and the second dimension of <i>b</i> must be at least $\max(1, nrhs)$.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> , $lda \geq \max(1, n)$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> , $ldb \geq \max(1, nrhs)$.

Output Parameters

<i>b</i>	Overwritten by an enclosure of the solution matrix X .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> ≥ 1 , the execution is not successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter has an illegal value.

?gekws

Solves a system of interval linear equations by Krawczyk iteration method.

Syntax

```
call sigekws(trans, n, nrhs, a, lda, b, ldb, epsilon, info)
call digekws(trans, n, nrhs, a, lda, b, ldb, epsilon, info)
call crgekws(trans, n, nrhs, a, lda, b, ldb, epsilon, info)
call zrgekws(trans, n, nrhs, a, lda, b, ldb, epsilon, info)
call ccgekws(trans, n, nrhs, a, lda, b, ldb, epsilon, info)
call zcgekws(trans, n, nrhs, a, lda, b, ldb, epsilon, info)
```

Description

The routine ?gekws uses the Krawczyk interval iteration to compute an enclosure of the solution set to the following interval linear system of equations:

$A^*X = B$, if $trans = 'N'$

$A^T * X = B$, if $trans = 'T'$

$A^H * X = B$, if $trans = 'C'$ (for complex matrices only).

Input Parameters

<i>trans</i>	CHARACTER(1). Must be one of 'N', 'T', 'C', 'n', 't', or 'c'. Indicates the form of the equations system: If $trans = 'N'$ or 'n', then $A^*X = B$ is solved for X . If $trans = 'T'$ or 't', then $A^T * X = B$ is solved for X . If $trans = 'C'$ or 'c', then $A^H * X = B$ is solved for X .
<i>n</i>	INTEGER. The order of A , the number of rows in B ($n \geq 0$).
<i>nrhs</i>	INTEGER. The number of right-hand sides ($nrhs \geq 0$).
<i>a, b</i>	S_INTERVAL for sigekws. D_INTERVAL for digekws. CR_INTERVAL for crgekws. ZR_INTERVAL for zrgekws.

CC_INTERVAL for ccgekws.

ZC_INTERVAL for zcgekws.

Arrays: a ($lda, *$), b ($ldb, *$).

The array a contains the matrix A .

The array b contains the matrix B , whose columns are the right-hand sides for the systems of equations.

lda INTEGER. The first dimension of a , $lda \geq \max(1, n)$.

ldb INTEGER. The first dimension of b , $ldb \geq \max(1, n)$.

$epsilon$ REAL for sigekws, crgekws, and ccgekws.
DOUBLE PRECISION for digekws, zrgekws, and zcgekws.
The prescribed accuracy of the estimate.

Output Parameters

b Overwritten by an enclosure of the solution matrix x .

$info$ INTEGER.
If $info = 0$, the execution is successful.
If $info \geq 0$, the execution is not successful.
If $info = -i$, the i -th parameter has an illegal value.

Application Notes

Krawczyk interval iteration already incorporates midpoint inverse preconditioning, so that additional application of ?gemip routines is not necessary and does not improve the overall efficiency.

?gegss

Solves a system of interval linear equations by interval Gauss-Seidel iteration.

Syntax

```
call sigegss(trans, n, nrhs, a, lda, b, ldb, encl, epsilon, nits, info)
call digegss(trans, n, nrhs, a, lda, b, ldb, encl, epsilon, nits, info)
call crgegss(trans, n, nrhs, a, lda, b, ldb, encl, epsilon, nits, info)
call zrgegss(trans, n, nrhs, a, lda, b, ldb, encl, epsilon, nits, info)
call ccgegss(trans, n, nrhs, a, lda, b, ldb, encl, epsilon, nits, info)
call zcgegss(trans, n, nrhs, a, lda, b, ldb, encl, epsilon, nits, info)
```

Description

The routine ?gegss uses the interval Gauss-Seidel iteration to compute an enclosure of a portion of the solution set to the following interval linear system of equations:

$A^*X = B$, if $trans = 'N'$

$A^T * X = B$, if $trans = 'T'$

$A^H * X = B$, if $trans = 'C'$ (for complex matrices only).

See in Appendix C of this manual for example code on using this routine.

Input Parameters

<i>trans</i>	CHARACTER(1). Must be one of 'N', 'T', 'C', 'n', 't', or 'c'. Indicates the form of the equations system: If $trans = 'N'$ or 'n', then $A^*X = B$ is solved for X . If $trans = 'T'$ or 't', then $A^T * X = B$ is solved for X . If $trans = 'C'$ or 'c', then $A^H * X = B$ is solved for X .
<i>n</i>	INTEGER. The order of A , the number of rows in B ($n \geq 0$).
<i>nrhs</i>	INTEGER. The number of right-hand sides ($nrhs \geq 0$).
<i>a, b</i>	S_INTERVAL for sigegss. D_INTERVAL for digegss. CR_INTERVAL for crgegss.

	ZR_INTERVAL for zrgegss. CC_INTERVAL for ccgegss. ZC_INTERVAL for zcgegss. Arrays: <i>a</i> (<i>lda</i> ,*), <i>b</i> (<i>ldb</i> ,*). The array <i>a</i> contains the matrix <i>A</i> . The array <i>b</i> contains the matrix <i>B</i> , whose columns are the right-hand sides for the systems of equations.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> , $lda \geq \max(1, n)$.
<i>ldb</i>	INTEGER. The first dimension of <i>b</i> , $ldb \geq \max(1, n)$.
<i>encl</i>	S_INTERVAL for sigegss. D_INTERVAL for digegss. CR_INTERVAL for crgegss. ZR_INTERVAL for zrgegss. CC_INTERVAL for ccgegss. ZC_INTERVAL for zcgegss. Array: <i>encl</i> (<i>ldb</i> ,*). The array <i>encl</i> defines the interval box bounding the portion of the solution set that the routine estimates.
<i>epsilon</i>	REAL for sigegss, crgegss, and ccgegss. DOUBLE PRECISION for digegss, zrgegss, and zcgegss. The prescribed accuracy of the estimate.
<i>nits</i>	INTEGER. The number of Gauss-Seidel iterations, $nits \geq 0$.

Output Parameters

<i>b</i>	Overwritten by an enclosure of the solution matrix <i>x</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>i</i> ≥ 0, then the diagonal element <i>a</i> (<i>i</i> , <i>i</i>) of the matrix contains zero. The execution of the routine did not fail, but it is recommended to interchange the rows and/or columns of the matrix so as to exclude zero-containing elements from its main diagonal. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter has an illegal value.

Application Notes

Interval Gauss-Seidel iteration is a *local solver* of interval linear systems, which means that it is mainly intended for computing enclosures of portions of the solution set bounded by a given interval box in the space R^n .

If the goal is to compute, by `?gegs`, an enclosure of the entire solution set to an interval linear system of equations, then its initial (crude) enclosure should be provided through `encl` argument.

?gehbs

Solves a system of interval linear equations by Hansen-Bliek-Rohn procedure.

Syntax

```
call sigehts(trans, n, a, lda, b, ldb, info)
call digehts(trans, n, a, lda, b, ldb, info)
```

Description

The routine `?gehts` uses the Hansen-Bliek-Rohn procedure to compute an enclosure of the solution set to the following interval linear system of equations:

$A^*X = B$, if `trans = 'N'`,

$A^T * X = B$, if `trans = 'T' or 'C'`.

See in Appendix C of this manual for example code on using this routine.

Input Parameters

<code>trans</code>	CHARACTER(1). Must be one of 'N', 'T', 'C', 'n', 't', or 'c'. Indicates the form of the equations system: If <code>trans = 'N'</code> or 'n', then $A^*X = B$ is solved for X . If <code>trans = 'T'</code> or 'C' or 't' or 'c', then $A^T * X = B$ is solved for X .
<code>n</code>	INTEGER. The order of A , the number of rows in B ($n \geq 0$).
<code>a, b</code>	REAL for <code>sigehts</code> . DOUBLE PRECISION for <code>digehts</code> . Arrays: <code>a (lda,*)</code> , <code>b (ldb,*)</code> . The array <code>a</code> contains the matrix A .

The array *b* contains the matrix *B*, whose columns are the right-hand sides for the systems of equations.

lda INTEGER. The first dimension of *a*, $lda \geq \max(1, n)$.

ldb INTEGER. The first dimension of *b*, $ldb \geq \max(1, n)$.

Output Parameters

b Overwritten by the solution matrix *x*.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* ≥ 0, the execution is not successful.

If *info* = -*i*, the *i*-th parameter has an illegal value.

Application Notes

If the middle matrix of *A* is not close to a diagonal matrix, then the midpoint inverse preconditioning by ?gemip routine may be necessary to correct the interval linear system and yield better results.

Routines for Sharp Solution of Interval Systems

?gepps

Solves a system of interval linear equations by a parameter partitioning method.

Syntax

call sigepps(*trans*, *n*, *a*, *lda*, *b*, *ldb*, *cmpt*, *mode*, *estm*, *epsilon*, *nits*, *info*)

call digepps(*trans*, *n*, *a*, *lda*, *b*, *ldb*, *cmpt*, *mode*, *estm*, *epsilon*, *nits*, *info*)

Description

The routine ?gepps uses the parameter partitioning (PPS) method to compute some (or all) of the sharp outer component-wise estimates of the solution set to the following interval linear system of equations:

$A * X = B$, if *trans* = 'N',

$A^T * X = B$, if $trans = 'T' \text{ or } 'C'$.

Input Parameters

<i>trans</i>	CHARACTER(1). Must be one of 'N', 'T', 'C', 'n', 't', or 'c'. Indicates the form of the equations system: If $trans = 'N' \text{ or } 'n'$, then $A * X = B$ is solved for X. If $trans = 'T' \text{ or } 'C' \text{ or } 't' \text{ or } 'c'$, then $A^T * X = B$ is solved for X.
<i>n</i>	INTEGER. The order of A, the number of rows in B ($n \geq 0$).
<i>a, b</i>	REAL for sigepps. DOUBLE PRECISION for digepps. Arrays: $a(lda, *)$, $b(l db)$. The array a contains the matrix A. The array b contains the vector B of the right-hand sides for the system of equations.
<i>lda</i>	INTEGER. The first dimension of a, $lda \geq \max(1, n)$.
<i>ldb</i>	INTEGER. The first dimension of b, $ldb \geq \max(1, n)$.
<i>cmpt</i>	INTEGER. The number of the component of the solution set to be estimated.
<i>mode</i>	CHARACTER(1). Must be either 'L' or 'U' (or the corresponding lowercase letters). Indicates how to estimate the solution set along the coordinate direction specified by the parameter <i>cmpt</i>: if $mode = 'L' \text{ or } 'l'$, then the routine computes the lower estimate of the solution set over the <i>cmpt</i>-th coordinate; if $mode = 'U' \text{ or } 'u'$, then the routine computes the upper estimate of the solution set over the <i>cmpt</i>-th coordinate.
<i>epsilon</i>	REAL for sigepps. DOUBLE PRECISION for digepps. The prescribed accuracy of the estimate.
<i>nits</i>	INTEGER. The number of iterations of the PPS algorithm, $nits \geq 0$.

Output Parameters

<i>estm</i>	REAL for sigepps.

	DOUBLE PRECISION for <code>digepps</code> .
	Estimate of the solution set along the coordinate axis with the number <code>cmpt</code> . if <code>mode = 'L'</code> , then <code>estm</code> represents the lower estimate of the solution set. if <code>mode = 'U'</code> , then <code>estm</code> is equal to the upper estimate of the solution set.
<code>epsilon</code>	The actual precision of the estimate.
<code>nits</code>	The number of iterations that the algorithm actually executed.
<code>info</code>	INTEGER. If <code>info = 0</code> , the execution is successful. If <code>info = i ≥ 0</code> , the execution is not successful. If <code>info = -i</code> , the <i>i</i> -th parameter has an illegal value.

Application Notes

Routines `?gepps` and `?gepss` implement two mutually dual algorithms for computing optimal estimates of the solution sets to interval linear systems, which use adaptive partitioning of either parameter set or solution set of the system. The choice of either of these methods should be based on the specific features of the problem under solution such as the number of essentially interval parameters, the shape of the solution set, and so on.

For example, if the interval system has few interval parameters, then PPS-method is preferable, while PSS-method works better for systems that have simple shape of the solution set.

Computing optimal (or sharp) enclosures of the solution sets to interval linear systems, as well as enclosures that are guaranteed to be sharp within a prescribed accuracy, is known to be an NP-hard problem.

Therefore, a good choice of the parameters `epsilon` and `nits` becomes crucial for effective work of `?gepps` and `?gepss` routines and for producing desired results.

With this in mind, the recommendation is to organize the whole solution process with `?gepps` or `?gepss` interactively as a sequence of routine calls, starting from a moderate `nits` and a rough `epsilon` and then increasing `nits` and reducing `epsilon` until `?gepps` or `?gepss` still complete their execution.

?gepss

Solves a system of interval linear equations by a solution partitioning method.

Syntax

```
call sigepss( trans, n, a, lda, b, ldb, cmpt, mode, estm, epsilon, nits, info)
call digepss( trans, n, a, lda, b, ldb, cmpt, mode, estm, epsilon, nits, info)
```

Description

The routine ?gepss uses the (PSS) method to compute a sharp outer component-wise estimate, along a prescribed coordinate axis, of the solution set to the following interval linear system of equations:

$A * X = B$, if $trans = 'N'$,

$A^T * X = B$, if $trans = 'T'$ or $'C'$.

Input Parameters

<i>trans</i>	CHARACTER(1). Must be one of 'N', 'T', 'C', 'n', 't', or 'c'. Indicates the form of the equations system: If $trans = 'N'$ or $'n'$, then $A * X = B$ is solved for X . If $trans = 'T'$ or $'C'$ or $'t'$ or $'c'$, then $A^T * X = B$ is solved for X .
<i>n</i>	INTEGER. The order of A , the number of rows in B ($n \geq 0$).
<i>a, b</i>	REAL for sigepss. DOUBLE PRECISION for digepss. Arrays: $a(lda, *)$, $b(ldb)$. The array a contains the matrix A . The array b contains the vector B of the right-hand sides for the system of equations to be solved.
<i>lda</i>	INTEGER. The first dimension of a , $lda \geq \max(1, n)$.
<i>ldb</i>	INTEGER. The first dimension of b , $ldb \geq \max(1, n)$.
<i>cmpt</i>	INTEGER. The number of the component of the solution set to be estimated.
<i>mode</i>	CHARACTER(1). Must be either 'L' or 'U' (or the corresponding lowercase letters).

Indicates how to estimate the solution set along the coordinate direction specified by the parameter *cmpt*:
 if *mode* = 'L' or 'l', then the routine computes the lower estimate of the solution set over the *cmpt*-th coordinate;
 if *mode* = 'U' or 'u', then the routine computes the upper estimate of the solution set over the *cmpt*-th coordinate.

epsilon REAL for *sigepps*.
 DOUBLE PRECISION for *digepps*.
 The prescribed accuracy of the estimate for the solution set.

nits INTEGER. The number of iterations of the PSS algorithm, *nits* ≥ 0.

Output Parameters

estm REAL for *sigepps*.
 DOUBLE PRECISION for *digepps*.
 Estimate of the solution set along the coordinate axis with the number *cmpt*. if *mode* = 'L', then *estm* represents the lower estimate of the solution set. if *mode* = 'U', then *estm* is equal to the upper estimate of the solution set.

epsilon The actual precision of the estimate *estm*.

nits INTEGER. The number of iterations that the algorithm actually executed.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = *i* ≥ 0, the execution is not successful.
 If *info* = -*i*, the *i*-th parameter has an illegal value and the routine outputs the corresponding message.

Application Notes

Routines *?gepps* and *?gepps* implement two mutually dual algorithms for computing optimal estimates of the solution sets to interval linear systems, which use adaptive partitioning of either parameter set or solution set of the system. The choice of either of these methods should be based on the specific features of the problem under solution such as the number of essentially interval parameters, the shape of the solution set, and so on.

For example, if the interval system has few interval parameters, then PPS-method is preferable, while PSS-method works better for systems that have simple shape of the solution set.

Computing optimal (or sharp) enclosures of the solution sets to interval linear systems, as well as enclosures that are guaranteed to be sharp within a prescribed accuracy, is known to be an NP-hard problem.

Therefore, a good choice of the parameters *epsilon* and *nits* becomes crucial for effective work of ?gepps and ?gepss routines and for producing desired results.

With this in mind, the recommendation is to organize the whole solution process with ?gepps or ?gepss interactively as a sequence of routine calls, starting from a moderate *nits* and a rough *epsilon* and then increasing *nits* and reducing *epsilon* until ?gepps or ?gepss still complete their execution.

Example 12-2 Fortran 90 Code for Parameter Partitioning (PPS) Method

Consider a sample problem that requires computing a sharp lower estimate (to within the accuracy of, say, $1.E-4$), along the first coordinate direction, of the solution set to the interval linear system

$$\begin{pmatrix} 3.5 & [0, 2] & [0, 2] \\ [0, 2] & 3.5 & [0, 2] \\ [0, 2] & [0, 2] & 3.5 \end{pmatrix} x = \begin{pmatrix} [-1, 1] \\ [-1, 1] \\ [-1, 1] \end{pmatrix}$$

The problem can be solved by the following Fortran code that implements parameter partitioning method (PPS-method) and uses `sigepps` routine:

```

-----
. . . . .
USE INTERVAL_ARITHMETIC
. . . . .
INTEGER, PARAMETER          :: LDA = 3, LDB = 3
INTEGER                      :: NITS, CMPT, INFO, I, J
CHARACTER(1)                 :: MODE = 'L'
REAL(4)                      :: EPS, ESTM
TYPE(S_INTERVAL)             :: A(3,3), B(3)
. . . . .
DO I = 1, 3
    DO J = 1, 3
        IF( I/=J ) THEN
            A(I,J) = SINTERVAL(0.,2.)
        ELSE
            A(I,J) = SINTERVAL(3.5)
        END IF
        B(I) = SINTERVAL(-1.,1.)
    END DO
END DO

CMPT = 1
NITS = 100
EPS= 1.E-4
. . . . .
CALL SIGEPPS( 'n', 3, A, LDA, B, LDB, CMPT, MODE, ESTM, EPS, NITS, INFO )
-----

```

To guarantee the completion of the algorithm, the value of the parameter `NITS` is set equal to 100 (iterations), which is enough for the above specific example. Note that assigning single-precision intervals to the entries of the matrix `A` and right-hand side vector `B` is carried out by `SINTERVAL` function supplied by `INTERVAL_ARITHMETIC` module.

Routines for Inverting Interval Matrices

?trtri

Computes inverse interval matrix to a triangular interval matrix.

Syntax

```
call sitrtri(uplo, diag, n, a, lda, info)
call ditrtri(uplo, diag, n, a, lda, info)
call crtrtri(uplo, diag, n, a, lda, info)
call ztrtrtri(uplo, diag, n, a, lda, info)
call cctrtri(uplo, diag, n, a, lda, info)
call zctrtri(uplo, diag, n, a, lda, info)
```

Description

The routine `?trtri` computes an interval enclosure of the inverse $\text{inv}(A)$ of an interval triangular matrix `A`.

This routine implements a backward substitution algorithm and produces optimal enclosures of the inverse interval matrix, which is due to the simple structure of the matrix to be inverted.

Input Parameters

<code>uplo</code>	CHARACTER(1). Must be one of 'U', 'L', 'u', or 'l'. Indicates whether <code>A</code> is upper or lower triangular. If <code>uplo</code> = 'U' or 'u', then <code>A</code> is upper triangular. If <code>uplo</code> = 'L' or 'l', then <code>A</code> is lower triangular.
<code>diag</code>	CHARACTER(1). Must be one of 'N', 'U', 'n', or 'u'. If <code>diag</code> = 'N' or 'n', then <code>A</code> is not a unit triangular matrix.

If *diag* = 'U' or 'u', then *A* is unit triangular: diagonal elements of *A* are assumed to be 1 and not referenced in the array *a*.

n INTEGER. The order of the matrix *A* ($n \geq 0$).

a S_INTERVAL for sitrtri.
D_INTERVAL for ditrtri.
CR_INTERVAL for cctrtri.
ZR_INTERVAL for ztrtri.
CC_INTERVAL for cctrtri.
ZC_INTERVAL for zctrtri.
Array: DIMENSION (*lda*, *).
Contains the matrix *A*.
The second dimension of *a* must be at least $\max(1, n)$.

lda INTEGER. The first dimension of *a*, $lda \geq \max(1, n)$.

Output Parameters

a Overwritten by an interval *n*-by-*n* matrix that encloses the inverse matrix $\text{inv}(A)$.

info INTEGER.
If *info* = 0, the execution was successful.
If *info* = -*i*, the *i*-th parameter has an illegal value.
If *info* = *i*, the *i*-th diagonal element of *A* contains zero, *A* is singular, and its inversion could not be completed.

?geszi

Computes inverse interval matrix by Schulz iterative method.

Syntax

```
call sigeszi(n, a, lda, info)
call digeszi(n, a, lda, info)
```

Description

For a general interval square matrix *A*, the routine ?geszi computes an enclosure of the inverse interval matrix $\text{inv}(A)$ by the Schulz iterative method.

See in Appendix C of this manual for example code on using this routine.

Input Parameters

n INTEGER. The order of the matrix *A* ($n \geq 0$).

a REAL for *sigeszi*.
DOUBLE PRECISION for *digeszi*.
Array: DIMENSION (*lda*,*).
Contains the matrix *A*.
The second dimension of *a* must be at least $\max(1, n)$.

lda INTEGER. The first dimension of *a*, $lda \geq \max(1, n)$.

Output Parameters

a Overwritten by an enclosure of the inverse interval matrix $\text{inv}(A)$.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = *i* ≥ 0 , the execution is not successful.
If *info* = -*i*, the *i*-th parameter has an illegal value.

Application Notes

Schulz iteration implemented in *?geszi* routine converges only provided that the interval matrix *A* is not "too wide". Otherwise, when Schulz iteration diverges, the result is set equal to the interval matrix with the elements $[-\text{infty}, +\text{infty}]$, where *infty* is computer infinity of the corresponding kind.

Routines for Checking Properties of Interval Matrices

?gerbr

Tests regularity of an interval matrix by Ris-Beeck and Rex-Rohn criteria.

Syntax

```
call sigerbr( n, a, lda, sr, reg, info )
call digerbr( n, a, lda, sr, reg, info )
```

Description

The routine `?gerbr` checks whether a general interval square matrix A is regular or singular by using a combination of Ris-Beeck spectral criterion and Rex-Rohn test.

Input Parameters

<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>a</i>	REAL for <code>sigerbr</code> . DOUBLE PRECISION for <code>digerbr</code> . Array: DIMENSION (<i>lda</i> ,*). Contains the matrix A . The second dimension of <i>a</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The first dimension of <i>a</i> , $lda \geq \max(1, n)$.

Output Parameters

<i>sr</i>	REAL for <code>sigerbr</code> . DOUBLE PRECISION for <code>digerbr</code> . An upper estimate of the spectral radius of the matrix $(\text{inv}(\text{mid } A) \cdot \text{rad } A)$. This is an additional information about the matrix A , which is crucial for the so-called strong regularity of A .
<i>reg</i>	INTEGER. Displays the results of the singularity test. If $reg \geq 0$, then A is regular. If $reg < 0$, then A is singular. If $reg = 0$, then the result is undetermined, that is, the test was not sufficiently sensitive to detect whether the matrix A is regular or singular.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = i \geq 0$, the execution is not successful. If $info = -i$, the i -th parameter has an illegal value.

Application Notes

The test implemented in the routine `?gerbr` is rather crude, and in critical cases further investigation of the matrix A is recommended. However, the routine may help to determine (by comparing the value of sr with 1) whether an interval matrix is strongly regular or not.

?gesvr

*Tests regularity/singularity of an interval matrix
by Rump and Rex-Rohn singular value criteria.*

Syntax

```
call sigesvr(n, a, lda, msr, rsr, reg, info)
call digesvr(n, a, lda, msr, rsr, reg, info)
```

Description

The routine `?gesvr` checks whether a general interval square matrix A is regular or singular by using Rump and Rex-Rohn singular value criteria.

Input Parameters

n INTEGER. The order of the matrix A ($n \geq 0$).

a REAL for `sigesvr`.
DOUBLE PRECISION for `digesvr`.
Array: DIMENSION (*lda*, *).
Contains the matrix A .
The second dimension of *a* must be at least $\max(1, n)$.

lda INTEGER. The first dimension of *a*, $lda \geq \max(1, n)$.

Output Parameters

msr, rsr S_INTERVAL for `sigesvr`.
D_INTERVAL for `digesvr`.
Additional information about the matrix A .
The intervals represent the ranges of the singular spectra of the midpoint matrix and radius matrix, respectively.

reg INTEGER. Displays results of the singularity test.

If $reg \geq 0$, then A is regular.

If $reg < 0$, then A is singular.

If $reg = 0$, then the result is undetermined, that is, the test was not sufficiently sensitive to detect whether the matrix A is regular or singular.

info

INTEGER.

If $info = 0$, the execution is successful.

If $info = i \geq 0$, the execution is not successful.

If $info = -i$, the i -th parameter has an illegal value.

Application Notes

The routine `?gesvr` implements a test that is only a sufficient condition for a matrix to be either regular or singular. This means that in some boundary cases the test may prove not sensitive enough to determine whether a given matrix is regular or singular, and the routine returns $reg = 0$ on output.

Example 12-3 Fortran 90 Code for Testing Regularity of Interval Matrix by Singular Value Criteria

To test regularity of the interval matrix

$$\begin{pmatrix} [2, 4] & [-1, 2] \\ [-2, 1] & [2, 4] \end{pmatrix}$$

by singular value criteria, the following piece of Fortran 90 code may be helpful:


```

-----
. . . . .
USE INTERVAL_ARITHMETIC

. . . . .
INTEGER, PARAMETER      :: LDA = 2, N = 2
TYPE(D_INTERVAL)        :: A(LDA,N), MSR, RSR
INTEGER                  :: REG, INFO

. . . . .
A(1,1) = DINTERVAL(2.,4.); A(1,2) = DINTERVAL(-2.,1.)
A(2,1) = DINTERVAL(-1.,2.); A(2,2) = DINTERVAL(2.,4.)

. . . . .
CALL DIGESVR( N, A, LDA, MSR, RSR, REG, INFO )
-----

```

Mutual disposition of the intervals `MSR` and `RSR` on the real axis can serve to some extent as a measure of how large the regularity margin is (in case of $MSR \geq RSR$), or how far the matrix is from the regular ones (in case of $MSR < RSR$).

Auxiliary and Utility Routines

?gemip

Performs midpoint-inverse preconditioning of an interval linear system.

Syntax

```
call sigemip(n, nrhs, a, lda, b, ldb, info)
call digemip(n, nrhs, a, lda, b, ldb, info)
call crgemip(n, nrhs, a, lda, b, ldb, info)
call zrgemip(n, nrhs, a, lda, b, ldb, info)
call ccgemip(n, nrhs, a, lda, b, ldb, info)
call zcgemip(n, nrhs, a, lda, b, ldb, info)
```

Description

The routine ?gemip performs midpoint-inverse preconditioning of the interval linear system $A \cdot X = B$. This is done through multiplying both matrices A and B by the midpoint-inverse matrix $\text{inv}(\text{mid } A)$ in computer (rounded) interval arithmetic.

Input Parameters

<i>n</i>	INTEGER. The order of the matrix A .
<i>nrhs</i>	INTEGER. The number of right-hand sides ($nrhs \geq 0$).
<i>a, b</i>	S_INTERVAL for sigemip. D_INTERVAL for digemip. CR_INTERVAL for crgemip. ZR_INTERVAL for zrgemip. CC_INTERVAL for ccgemip. ZC_INTERVAL for zcgemip. Arrays: <i>a</i> (<i>lda</i>, *), <i>b</i> (<i>ldb</i>, *). The array <i>a</i> contains the matrix A. The array <i>b</i> contains the matrix B, whose columns are the right-hand sides for the systems of equations.

The second dimension of a must be at least $\max(1, n)$ and the second dimension of b must be at least $\max(1, nrhs)$.

lda INTEGER. The first dimension of a , $lda \geq \max(1, n)$.
 ldb INTEGER. The first dimension of b , $ldb \geq \max(1, n)$.

Output Parameters

a Overwritten by the preconditioned matrix A .
 b Overwritten by the preconditioned matrix B .
 $info$ INTEGER.
 If $info = 0$, the execution is successful.
 If $info \geq 1$, the execution is not successful.
 If $info = -i$, the i -th parameter has an illegal value.

Application Notes

Preconditioning may sometimes extend applicability of the algorithms for the solution of interval linear systems and/or improve the quality of the results produced by these algorithms.

In particular, interval Gauss method, interval Householder method and interval Gauss-Seidel iteration applied to interval linear systems with "wide" matrices that are not diagonally dominant should be preceded by preconditioning to yield better results. For Hansen-Bliek-Rohn procedure, the midpoint-inverse preconditioning is recommended if the middle matrix of the system is far from diagonal.

Example 12-4 Fortran 90 Code for Preconditioning Interval Linear System

The following piece of Fortran code presents an example of how you can perform preconditioning of the interval linear system

$$\begin{pmatrix} [2, 4] & [-2, 1] \\ [-1, 2] & [2, 4] \end{pmatrix} x = \begin{pmatrix} [0, 2] & [-2, 2] \\ [0, 2] & [-2, 2] \end{pmatrix}$$

and then solve it by using the interval Gauss method:

```

-----
. . . . .
USE INTERVAL_ARITHMETIC

. . . . .
TYPE(D_INTERVAL)           :: A(2,2), B(2,2)
INTEGER                    :: N = 2, NRHS = 2, LDA = 2, LDB = 2, INFO
CHARACTER(1)               :: TRANS = 'n'

. . . . .
A(1,1) = DINTERVAL(2.,4.);   A(1,2) = DINTERVAL(-2.,1.)
A(2,1) = DINTERVAL(-1.,2.);  A(2,2) = DINTERVAL(2.,4.)
B(1,1) = DINTERVAL(0.,2.);   B(2,1) = DINTERVAL(0.,2.)
B(1,2) = DINTERVAL(-2.,2.);  B(2,2) = DINTERVAL(-2.,2.)

. . . . .
CALL DIGEMIP( N, NRHS, A, LDA, B, LDB, INFO )
CALL DIGEGAS( TRANS, N, NRHS, A, LDA, B, LDB, INFO )
-----

```

For more code examples on using this routine, see in Appendix C of this manual .

Partial Differential Equations Support

13

The Intel® Math Kernel Library (Intel® MKL) provides tools for solving Partial Differential Equations (PDE). These tools are Trigonometric Transform interface routines (see [Trigonometric Transform Routines](#)) and Poisson Library (see [Poisson Library Routines](#)).

Poisson Library is designed for fast solving of simple Helmholtz, Poisson, and Laplace problems. The solver is based on the Trigonometric Transform interface, which is, in turn, based on the Intel MKL Fast Fourier Transform (FFT) interface (refer to [Fourier Transform Functions](#)), optimized for Intel® processors.

Direct use of the Trigonometric Transform routines may be helpful to those who have already implemented their own solvers similar to the one that the Poisson Library provides. As it may be hard enough to modify the original code so as to make it work with Poisson Library, you are encouraged to use fast (staggered) sine/cosine transforms implemented in the Trigonometric Transform interface to improve performance of your solver.

Both Trigonometric Transform and Poisson Library routines can be called from C and Fortran 90, although the interfaces description uses C convention. Fortran 90 users can find routine calls specifics in the “[Calling PDE Support Routines from Fortran 90](#)” section.

Trigonometric Transform Routines

In addition to the Fast Fourier Transform (FFT) interface, described in chapter “[Fast Fourier Transforms](#)”, Intel® MKL supports the Real Discrete Trigonometric Transforms (sometimes called real-to-real Discrete Fourier Transforms) interface. In this manual, the interface is referred to as TT interface. It implements a group of routines (TT routines) used to compute sine/cosine, staggered sine/cosine, and twice staggered sine/cosine transforms (referred to as staggered2 sine/cosine transforms, for brevity). The TT interface provides much flexibility of use: you can adjust routines to your particular needs at the cost of manual tuning routine parameters or just call routines with default parameter values. The current Intel MKL implementation of the TT interface can be used in solving partial differential equations and contains routines that are helpful for Fast Poisson and similar solvers.

To describe the Intel MKL TT interface, the C convention is used. Fortran users should refer to [Calling PDE Support Routines from Fortran 90](#).

For the list of Trigonometric Transforms currently implemented in Intel MKL TT interface, see [Transforms Implemented](#).

If you have got used to the FFTW interface (www.fftw.org), you can call the TT interface functions through real-to-real FFTW to Intel MKL wrappers without changing FFTW function calls in your code (refer to *FFTW to Intel® MKL Wrappers Technical User Notes for FFTW 3.x* for details). However, you are strongly encouraged to use the native TT interface for better performance. Another reason why

you should use the wrappers cautiously is that TT and the real-to-real FFTW interfaces are not fully compatible and some features of the real-to-real FFTW, such as strides and multidimensional transforms, are not available through wrappers.

Transforms Implemented

TT routines allow computing the following transforms:

Forward sine transform

$$F(k) = \frac{2}{n} \sum_{i=1}^{n-1} f(i) \sin \frac{ki\pi}{n}, k = 1, \dots, n-1$$

Backward sine transform

$$f(i) = \sum_{k=1}^{n-1} F(k) \sin \frac{ki\pi}{n}, i = 1, \dots, n-1$$

Forward staggered sine transform

$$F(k) = \frac{1}{n} \sin \frac{(2k-1)\pi}{2} f(n) + \frac{2}{n} \sum_{i=1}^{n-1} f(i) \sin \frac{(2k-1)i\pi}{2n}, k = 1, \dots, n$$

Backward staggered sine transform

$$f(i) = \sum_{k=1}^n F(k) \sin \frac{(2k-1)i\pi}{2n}, i = 1, \dots, n$$

Forward staggered2 sine transform

$$F(k) = \frac{2}{n} \sum_{i=1}^n f(i) \sin \frac{(2k-1)(2i-1)\pi}{4n}, \quad k = 1, \dots, n$$

Backward staggered2 sine transform

$$f(i) = \sum_{k=1}^n F(k) \sin \frac{(2k-1)(2i-1)\pi}{4n}, \quad i = 1, \dots, n$$

Forward cosine transform

$$F(k) = \frac{1}{n} \left[f(0) + f(n) \cos k\pi \right] + \frac{2}{n} \sum_{i=1}^{n-1} f(i) \cos \frac{ki\pi}{n}, \quad k = 0, \dots, n$$

Backward cosine transform

$$f(i) = \frac{1}{2} \left[F(0) + F(n) \cos i\pi \right] + \sum_{k=1}^{n-1} F(k) \cos \frac{ki\pi}{n}, \quad i = 0, \dots, n$$

Forward staggered cosine transform

$$F(k) = \frac{1}{n} f(0) + \frac{2}{n} \sum_{i=1}^{n-1} f(i) \cos \frac{(2k+1)i\pi}{2n}, \quad k = 0, \dots, n-1$$

Backward staggered cosine transform

$$f(i) = \sum_{k=0}^{n-1} F(k) \cos \frac{(2k+1)i\pi}{2n}, i = 0, \dots, n-1$$

Forward staggered2 cosine transform

$$F(k) = \frac{2}{n} \sum_{i=1}^n f(i) \cos \frac{(2k-1)(2i-1)\pi}{4n}, k = 1, \dots, n$$

Backward staggered2 cosine transform

$$f(i) = \sum_{k=1}^n F(k) \cos \frac{(2k-1)(2i-1)\pi}{4n}, i = 1, \dots, n$$



NOTE. The size of the transform n can be any integer greater or equal to 2.

Sequence of Invoking TT Routines

Computation of a transform using TT interface is conceptually divided into four steps, each of which is performed via a dedicated routine. [Table 13-1](#) lists the routines and briefly describes their purpose and use.

Most TT routines have versions operating with single-precision and double-precision data. Names of such routines begin respectively with "s" and "d". The wildcard "?" stands for either of these symbols in routine names.

Table 13-1 TT Interface Routines

Routine	Description
<code>?_init_trig_transform</code>	Initializes basic data structures of Trigonometric Transforms.
<code>?_commit_trig_transform</code>	Checks consistency and correctness of user-defined data as well as creates a data structure to be used by Intel MKL FFT interface ¹ .
<code>?_forward_trig_transform</code> <code>?_backward_trig_transform</code>	Computes a forward/backward Trigonometric Transform of a specified type using the appropriate formula (see Transforms Implemented).
<code>free_trig_transform</code>	Cleans the memory used by a data structure needed for calling FFT interface ¹ .

¹TT routines call Intel MKL FFT interface for better performance.

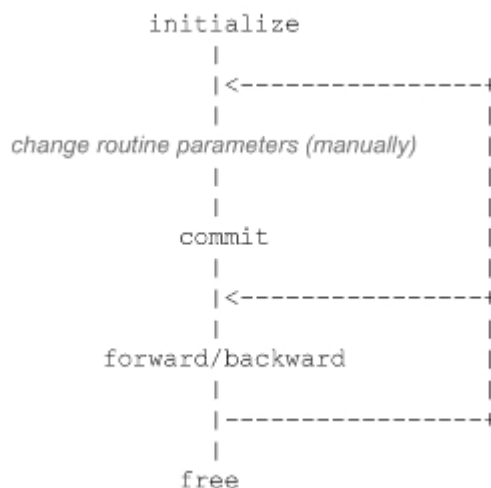
To find a transformed vector for a particular input vector only once, the Intel MKL TT interface routines are normally invoked in the order in which they are listed in [Table 13-1](#).



NOTE. Though the order of invoking TT routines may be changed, it is highly recommended to follow the above order of routine calls.

The diagram in [Figure 13-1](#) indicates the typical order in which TT interface routines can be invoked in a general case (prefixes and suffixes in routine names are omitted).

Figure 13-1 Typical Order of Invoking TT Interface Routines



A general scheme of using TT routines for double-precision computations is shown below. A similar scheme holds for single-precision computations with the only difference in the initial letter of routine names.

```

...
    d_init_trig_transform(&n, &tt_type, ipar, dpar, &ir);
/* Change parameters in ipar if necessary. */
/* Note that the result of the Transform will be in f ! If you want to preserve the data
stored in f,
save them before this place in your code */
    d_commit_trig_transform(f, &handle, ipar, dpar, &ir);
    d_forward_trig_transform(f, &handle, ipar, dpar, &ir);
    d_backward_trig_transform(f, &handle, ipar, dpar, &ir);
    free_trig_transform(&handle, ipar, &ir);
/* here the user may clean the memory used by f, dpar, ipar */
...

```

You can find examples of Fortran 90 and C code that use TT interface routines to solve one-dimensional Helmholtz problem in the “[Trigonometric Transform Code Examples](#)” section in Appendix C.

Interface Description

All types in this documentation are standard C types: `int`, `float`, and `double`. Fortran 90 users can call the routines with `INTEGER`, `REAL`, and `DOUBLE PRECISION` Fortran types, respectively (see examples in the “[Trigonometric Transform Code Examples](#)” section in Appendix C).

The interface description uses the built-in type `int` for integer values. If you employ the ILP64 interface, read this type as `long long int` (or `INTEGER*8` for Fortran). For more information, refer to the *Intel MKL User's Guide*.

Routine Options

All TT routines use parameters to pass various options to one another. These parameters are arrays `ipar`, `dpar` and `spar`. Values for these parameters should be specified very carefully (see [Common Parameters](#)). You can change these values during computations to meet your needs.



WARNING. To avoid failure or wrong results, you must provide correct and consistent parameters to the routines.

User Data Arrays

TT routines take arrays of user data as input. For example, user arrays are passed to the routine `d_forward_trig_transform` to compute a forward Trigonometric Transform. To minimize storage requirements and improve the overall run-time efficiency, Intel MKL TT routines do not make copies of user input arrays.



NOTE. If you need a copy of your input data arrays, save them yourself.

TT Routines

The section gives detailed description of TT routines, their syntax, parameters and values they return. Double-precision and single-precision versions of the same routine are described together.

TT routines call Intel MKL FFT interface (described in section “[FFT Functions](#)” in chapter “Fast Fourier Transforms”), which enhances performance of the routines.

?_init_trig_transform

Initializes basic data structures of a Trigonometric Transform.

Syntax

```
void d_init_trig_transform (int *n, int *tt_type, int ipar[], double dpar[],
int *stat);
```

```
void s_init_trig_transform (int *n, int *tt_type, int ipar[], float spar[],
int *stat);
```

Input Parameters

<i>n</i>	int*. Contains the size of the problem, which should be a positive integer greater than 1. Note that data vector of the transform, which other TT routines will use, must have size <i>n</i> +1 for all but staggered2 transforms. Staggered2 transforms require the vector of size <i>n</i> .
<i>tt_type</i>	int*. Contains the type of transform to compute, defined via a set of named constants. The following constants are available in the current implementation of TT interface: MKL_SINE_TRANSFORM, MKL_STAGGERED_SINE_TRANSFORM, MKL_STAGGERED2_SINE_TRANSFORM; MKL_COSINE_TRANSFORM, MKL_STAGGERED_COSINE_TRANSFORM, MKL_STAGGERED2_COSINE_TRANSFORM.

Output Parameters

<i>ipar</i>	int array of size 128. Contains integer data needed for Trigonometric Transform computations.

<code>dpar</code>	double array of size $5n/2+2$. Contains double-precision data needed for Trigonometric Transform computations.
<code>spar</code>	float array of size $5n/2+2$. Contains single-precision data needed for Trigonometric Transform computations.
<code>stat</code>	int*. Contains the routine completion status, which is also written to <code>ipar[6]</code> . The status should be 0 to proceed to other TT routines.

Description

The routine initializes basic data structures for Trigonometric Transforms of appropriate precision. After a call to `?_init_trig_transform`, all subsequently invoked TT routines use values of `ipar` and `dpar` (`spar`) array parameters returned by `?_init_trig_transform`. The routine initializes the entire array `ipar`. In the `dpar` or `spar` array, `?_init_trig_transform` initializes elements that do not depend upon the type of transform. For a detailed description of arrays `ipar`, `dpar` and `spar`, refer to the [Common Parameters](#) section. You can skip calling the initialization routine in your code. For more information, see [Caveat on Parameter Modifications](#).

Return Values

<code>stat= 0</code>	The routine successfully completed the task. In general, to proceed with computations, the routine should complete with this <code>stat</code> value.
<code>stat= -99999</code>	The routine failed to complete the task.

?_commit_trig_transform

Checks consistency and correctness of user's data as well as initializes certain data structures required to perform the Trigonometric Transform.

Syntax

```
void d_commit_trig_transform (double f[], DFTI_DESCRIPTOR_HANDLE *handle,
int ipar[], double dpar[], int *stat);

void s_commit_trig_transform (float f[], DFTI_DESCRIPTOR_HANDLE *handle, int
ipar[], float spar[], int *stat);
```

Input Parameters

<i>f</i>	double for <code>d_commit_trig_transform</code>, float for <code>s_commit_trig_transform</code>, array of size n for staggered2 transforms and of size $n+1$ for all other transforms, where n is the size of the problem. Contains data vector to be transformed. Note that the following values should be 0.0 up to rounding errors: • $f[0]$ and $f[n]$ for sine transforms • $f[n]$ for staggered cosine transforms • $f[0]$ for staggered sine transforms. Otherwise, the routine will produce a warning, and the result of the computations for sine transforms may be wrong. These restrictions meet the requirements of the Poisson Library (described in the Poisson Library Routines section), which the TT interface is primarily designed for.
<i>ipar</i>	int array of size 128. Contains integer data needed for Trigonometric Transform computations.
<i>dpar</i>	double array of size $5n/2+2$. Contains double-precision data needed for Trigonometric Transform computations. The routine initializes most elements of this array.
<i>spar</i>	float array of size $5n/2+2$. Contains single-precision data needed for Trigonometric Transform computations. The routine initializes most elements of this array.

Output Parameters

<i>handle</i>	DFTI_DESCRIPTOR_HANDLE*. The data structure used by Intel MKL FFT interface (for details, refer to section " FFT Functions " in chapter "Fast Fourier Transforms").
<i>ipar</i>	Contains integer data needed for Trigonometric Transform computations. On output, <i>ipar</i> [6] is updated with the <i>stat</i> value.
<i>dpar</i>	Contains double-precision data needed for Trigonometric Transform computations. On output, the entire array is initialized.

<i>spar</i>	Contains single-precision data needed for Trigonometric Transform computations. On output, the entire array is initialized.
<i>stat</i>	int*. Contains the routine completion status, which is also written to <i>ipar[6]</i> .

Description

The routine `?_commit_trig_transform` checks consistency and correctness of the parameters to be passed to the transform routines `?_forward_trig_transform` and/or `?_backward_trig_transform`. The routine also initializes the following data structures: *handle*, *dpar* in case of `d_commit_trig_transform`, and *spar* in case of `s_commit_trig_transform`. The `?_commit_trig_transform` routine initializes only those elements of *dpar* or *spar* that depend upon the type of transform, defined in the `?_init_trig_transform` routine and passed to `?_commit_trig_transform` with the *ipar* array. The size of the problem *n*, which determines sizes of the array parameters, is also passed to the routine with the *ipar* array and defined in the previously called `?_init_trig_transform` routine. For a detailed description of arrays *ipar*, *dpar* and *spar*, refer to the [Common Parameters](#) section. The routine performs only a basic check for correctness and consistency of the parameters. If you are going to modify parameters of TT routines, see the [Caveat on Parameter Modifications](#) section. Unlike `?_init_trig_transform`, the `?_commit_trig_transform` routine is mandatory, and you cannot skip calling it in your code.

Return Values

<i>stat</i> = 11	The routine produced some warnings and made some changes in the parameters to achieve their correctness and/or consistency. You may proceed with computations by assigning <i>ipar[6]</i> =0 if you are sure that the parameters are correct.
<i>stat</i> = 10	The routine made some changes in the parameters to achieve their correctness and/or consistency. You may proceed with computations by assigning <i>ipar[6]</i> =0 if you are sure that the parameters are correct.
<i>stat</i> = 1	The routine produced some warnings. You may proceed with computations by assigning <i>ipar[6]</i> =0 if you are sure that the parameters are correct.
<i>stat</i> = 0	The routine completed the task normally.

`stat= -100`

The routine stopped for any of the following reasons:

- An error in the user's data was encountered.
- Data in *ipar*, *dpar* or *spar* parameters became incorrect and/or inconsistent as a result of modifications.

`stat= -1000`

The routine stopped because of an FFT interface error.

`stat= -10000`

The routine stopped because the initialization failed to complete or the parameter *ipar*[0] was altered by mistake.



NOTE. Although positive values of *stat* usually indicate minor problems with the input data and Trigonometric Transform computations can be continued, you are highly recommended to investigate the problem first and achieve *stat*=0.

?_forward_trig_transform

Computes the forward Trigonometric Transform of type specified by the parameter.

Syntax

```
void d_forward_trig_transform (double f[], DFTI_DESCRIPTOR_HANDLE *handle,
int ipar[], double dpar[], int *stat);
```

```
void s_forward_trig_transform (float f[], DFTI_DESCRIPTOR_HANDLE *handle,
int ipar[], float spar[], int *stat);
```

Input Parameters

f

double for `d_forward_trig_transform`,
float for `s_forward_trig_transform`,
array of size *n* for staggered2 transforms and of size *n*+1
for all other transforms, where *n* is the size of the problem.
On input, contains data vector to be transformed. Note that
the following values should be 0.0 up to rounding errors:

- *f*[0] and *f*[*n*] for sine transforms

- $f[n]$ for staggered cosine transforms
- $f[0]$ for staggered sine transforms.

Otherwise, the routine will produce a warning, and the result of the computations for sine transforms may be wrong. The above restrictions meet the requirements of the Poisson Library (described in the [Poisson Library Routines](#) section), which the TT interface is primarily designed for.

<i>handle</i>	DFTI_DESCRIPTOR_HANDLE*. The data structure used by Intel MKL FFT interface (for details, refer to section “ FFT Functions ” in chapter “Fast Fourier Transforms”).
<i>ipar</i>	int array of size 128. Contains integer data needed for Trigonometric Transform computations.
<i>dpar</i>	double array of size $5n/2+2$. Contains double-precision data needed for Trigonometric Transform computations.
<i>spar</i>	float array of size $5n/2+2$. Contains single-precision data needed for Trigonometric Transform computations.

Output Parameters

<i>f</i>	Contains the transformed vector on output.
<i>ipar</i>	Contains integer data needed for Trigonometric Transform computations. On output, <i>ipar</i> [6] is updated with the <i>stat</i> value.
<i>stat</i>	int*. Contains the routine completion status, which is also written to <i>ipar</i> [6].

Description

The routine computes the forward Trigonometric Transform of type defined in the [?_init_trig_transform](#) routine and passed to [?_forward_trig_transform](#) with the *ipar* array. The size of the problem n , which determines sizes of the array parameters, is also passed to the routine with the *ipar* array and defined in the previously called [?_init_trig_transform](#) routine. The other data that facilitates the computation is created by [?_commit_trig_transform](#) and supplied in *dpar* or *spar*. For a detailed description of arrays *ipar*, *dpar* and *spar*, refer to the [Common Parameters](#) section. The routine has a commit step, which calls the [?_commit_trig_transform](#) routine. The transform is computed according to formulas given in the [Transforms Implemented](#) section. The routine replaces the input vector *f* with the transformed vector.



NOTE. If you need a copy of the data vector f to be transformed, make the copy before calling the `?_forward_trig_transform` routine.

Return Values

`stat= 0`

The routine completed the task normally.

`stat= -100`

The routine stopped for any of the following reasons:

- An error in the user's data was encountered.
- Data in `ipar`, `dpar` or `spar` parameters became incorrect and/or inconsistent as a result of modifications.

`stat= -1000`

The routine stopped because of an FFT interface error.

`stat= -10000`

The routine stopped because its commit step failed to complete or the parameter `ipar[0]` was altered by mistake.

?_backward_trig_transform

Computes the backward Trigonometric Transform of type specified by the parameter.

Syntax

```
void d_backward_trig_transform (double f[], DFTI_DESCRIPTOR_HANDLE *handle,
int ipar[], double dpar[], int *stat);
```

```
void s_backward_trig_transform (float f[], DFTI_DESCRIPTOR_HANDLE *handle,
int ipar[], float spar[], int *stat);
```

Input Parameters

f

double for `d_backward_trig_transform`,
float for `s_backward_trig_transform`,
array of size n for staggered2 transforms and of size $n+1$
for all other transforms, where n is the size of the problem.
On input, contains data vector to be transformed. Note that
the following values should be 0.0 up to rounding errors:

- $f[0]$ and $f[n]$ for sine transforms
- $f[n]$ for staggered cosine transforms
- $f[0]$ for staggered sine transforms.

Otherwise, the routine will produce a warning, and the result of the computations for sine transforms may be wrong. The above restrictions meet the requirements of the Poisson Library (described in the [Poisson Library Routines](#) section), which the TT interface is primarily designed for.

<i>handle</i>	DFTI_DESCRIPTOR_HANDLE*. The data structure used by Intel MKL FFT interface (for details, refer to section “ FFT Functions ” in chapter “Fast Fourier Transforms”).
<i>ipar</i>	int array of size 128. Contains integer data needed for Trigonometric Transform computations.
<i>dpar</i>	double array of size $5n/2+2$. Contains double-precision data needed for Trigonometric Transform computations.
<i>spar</i>	float array of size $5n/2+2$. Contains single-precision data needed for Trigonometric Transform computations.

Output Parameters

<i>f</i>	Contains the transformed vector on output.
<i>ipar</i>	Contains integer data needed for Trigonometric Transform computations. On output, <i>ipar</i> [6] is updated with the <i>stat</i> value.
<i>stat</i>	int*. Contains the routine completion status, which is also written to <i>ipar</i> [6].

Description

The routine computes the backward Trigonometric Transform of type defined in the [?_init_trig_transform](#) routine and passed to [?_backward_trig_transform](#) with the *ipar* array. The size of the problem n , which determines sizes of the array parameters, is also passed to the routine with the *ipar* array and defined in the previously called [?_init_trig_transform](#) routine. The other data that facilitates the computation is created by [?_commit_trig_transform](#) and supplied in *dpar* or *spar*. For a detailed description of arrays *ipar*, *dpar* and *spar*, refer to the [Common Parameters](#) section. The routine has a commit step, which calls the

`?_commit_trig_transform` routine. The transform is computed according to formulas given in the [Transforms Implemented](#) section. The routine replaces the input vector f with the transformed vector.



NOTE. If you need a copy of the data vector f to be transformed, make the copy before calling the `?_backward_trig_transform` routine.

Return Values

`stat= 0`

The routine completed the task normally.

`stat= -100`

The routine stopped for any of the following reasons:

- An error in the user's data was encountered.
- Data in `ipar`, `dpar` or `spar` parameters became incorrect and/or inconsistent as a result of modifications.

`stat= -1000`

The routine stopped because of an FFT interface error.

`stat= -10000`

The routine stopped because its commit step failed to complete or the parameter `ipar[0]` was altered by mistake.

free_trig_transform

Cleans the memory allocated for the data structure used by the FFT interface.

Syntax

```
void free_trig_transform (DFTI_DESCRIPTOR_HANDLE *handle, int ipar[], int *stat);
```

Input Parameters

`ipar`

`int` array of size 128. Contains integer data needed for Trigonometric Transform computations.

handle DFTI_DESCRIPTOR_HANDLE*. The data structure used by Intel MKL FFT interface (for details, refer to section “[FFT Functions](#)” in chapter “Fast Fourier Transforms”).

Output Parameters

handle The data structure used by Intel MKL FFT interface. Memory allocated for the structure is released on output.

ipar Contains integer data needed for Trigonometric Transform computations. On output, *ipar[6]* is updated with the *stat* value.

stat int*. Contains the routine completion status, which is also written to *ipar[6]*.

Description

The routine cleans the memory used by the *handle* structure, needed for Intel MKL FFT functions. To release the memory allocated for other parameters, include cleaning of the memory in your code.

Return Values

<i>stat</i> = 0	The routine completed the task normally.
<i>stat</i> = -1000	The routine stopped because of an FFT interface error.
<i>stat</i> = -99999	The routine failed to complete the task.

Common Parameters

This section provides description of array parameters that hold TT routine options: *ipar*, *dpar* and *spar*.



NOTE. Initial values are assigned to the array parameters by the appropriate [?_init_trig_transform](#) and [?_commit_trig_transform](#) routines.

ipar int array of size 128, holds integer data needed for Trigonometric Transform computations. Its elements are described in [Table 13-2](#):

Table 13-2 Elements of the *ipar* Array

Index	Description
0	Contains the size of the problem to solve. The <code>?_init_trig_transform</code> routine sets <code>ipar[0]=n</code> , and all subsequently called TT routines use <code>ipar[0]</code> as the size of the transform.
1	Contains error messaging options: • <code>ipar[1]=-1</code> indicates that all error messages will be printed to the file <code>MKL_Trig_Transforms_log.txt</code> in the folder from which the routine is called. If the file does not exist, the routine tries to create it. If the attempt fails, the routine prints information that the file cannot be created to the standard output device. • <code>ipar[1]=0</code> indicates that no error messages will be printed. • <code>ipar[1]=1</code> (default) indicates that all error messages will be printed to the preconnected default output device (usually, screen). In case of errors, each TT routine assigns a non-zero value to <code>stat</code> regardless of the <code>ipar[1]</code> setting.
2	Contains warning messaging options: • <code>ipar[2]=-1</code> indicates that all warning messages will be printed to the file <code>MKL_Trig_Transforms_log.txt</code> in the directory from which the routine is called. If the file does not exist, the routine tries to create it. If the attempt fails, the routine prints information that the file cannot be created to the standard output device. • <code>ipar[2]=0</code> indicates that no warning messages will be printed. • <code>ipar[2]=1</code> (default) indicates that all warning messages will be printed to the preconnected default output device (usually, screen). In case of warnings, the <code>stat</code> parameter will acquire a non-zero value regardless of the <code>ipar[2]</code> setting.
3 through 4	Reserved for future use.
5	Contains the type of the transform. The <code>?_init_trig_transform</code> routine sets <code>ipar[5]=tt_type</code> , and all subsequently called TT routines use <code>ipar[5]</code> as the type of the transform.
6	Contains the <code>stat</code> value returned by the last completed TT routine. Used to check that the previous call to a TT routine completed with <code>stat=0</code> .

Index	Description
7	Informs the <code>?_commit_trig_transform</code> routines whether to initialize data structures <code>dpar</code> (<code>spar</code>) and <code>handle</code>. <code>ipar[7]=0</code> indicates that the routine should skip the initialization and only check correctness and consistency of the parameters. Otherwise, the routine initializes the data structures. The default value is 1. The possibility to check correctness and consistency of input data without initializing data structures <code>dpar</code>, <code>spar</code> and <code>handle</code> enables avoiding performance losses in a repeated use of the same transform for different data vectors. Note that you can benefit from the opportunity that <code>ipar[7]</code> gives only if you are sure to have supplied proper tolerance value in the <code>dpar</code> or <code>spar</code> array. Otherwise, avoid tuning this parameter.
8	Contains message style options for TT routines. If <code>ipar[8]=0</code> then TT routines print all error and warning messages in Fortran-style notations. Otherwise, TT routines print the messages in C-style notations. The default value is 1. When selecting between these notations, mind that by default, numbering of elements in C arrays starts from 0 and in Fortran, it starts from 1. For example, for a C-style message "<i>parameter ipar[0]=3 should be an even integer</i>", the corresponding Fortran-style message will be "<i>parameter ipar(1)=3 should be an even integer</i>". The use of <code>ipar[8]</code> enables you to view messages in a more convenient style.
9	Specifies the number of OpenMP threads to run TT routines in the OpenMP environment of the Poisson Library. The default value is 1. You are highly recommended not to alter this value. See also Caveat on Parameter Modifications.
10	Specifies the mode of compatibility with FFTW. The default value is 0. Set the value to 1 to invoke compatibility with FFTW. In the latter case, results will not be normalized, because FFTW does not do this. It is highly recommended not to alter this value, but rather use real-to-real FFTW to MKL wrappers, described in <i>FFTW to Intel® MKL Wrappers Technical User Notes for FFTW 3.x</i>. See also Caveat on Parameter Modifications.
11 through 127	Reserved for future use.



NOTE. You may declare the `ipar` array in your code as `int ipar[11]`. However, for compatibility with later versions of Intel MKL TT interface, which may require more `ipar` values, it is highly recommended to declare `ipar` as `int ipar[128]`.

Arrays *dpar* and *spar* are the same except in the data precision:

<i>dpar</i>	double array of size $5n/2+2$, holds data needed for double-precision routines to perform TT computations. This array is initialized in the <code>d_init_trig_transform</code> and <code>d_commit_trig_transform</code> routines.
<i>spar</i>	float array of size $5n/2+2$, holds data needed for single-precision routines to perform TT computations. This array is initialized in the <code>s_init_trig_transform</code> and <code>s_commit_trig_transform</code> routines.

As *dpar* and *spar* have similar elements in respective positions, the elements are described together in Table 13-3:

Table 13-3 Elements of the *dpar* and *spar* Arrays

Index	Description
0	Contains the first absolute tolerance used by the appropriate <code>?_commit_trig_transform</code> routine. For a staggered cosine or a sine transform, $f[n]$ should be equal to 0.0 and for a staggered sine or a sine transform, $f[0]$ should be equal to 0.0. The <code>?_commit_trig_transform</code> routine checks whether absolute values of these parameters are below $dpar[0]*n$ or $spar[0]*n$, depending on the routine precision. To suppress warnings resulting from tolerance checks, set $dpar[0]$ or $spar[0]$ to a sufficiently large number.
1	Reserved for future use.
2 through $5n/2+1$	Contain tabulated values of trigonometric functions. Contents of the elements depend upon the type of transform <i>tt_type</i>, set up in the <code>?_commit_trig_transform</code> routine: • If <i>tt_type</i>=MKL_SINE_TRANSFORM, the transform uses only the first $n/2$ array elements, which contain tabulated sine values. • If <i>tt_type</i>=MKL_STAGGERED_SINE_TRANSFORM, the transform uses only the first $3n/2$ array elements, which contain tabulated sine and cosine values. • If <i>tt_type</i>=MKL_STAGGERED2_SINE_TRANSFORM, the transform uses all the $5n/2$ array elements, which contain tabulated sine and cosine values. • If <i>tt_type</i>=MKL_COSINE_TRANSFORM, the transform uses only the first n array elements, which contain tabulated cosine values. • If <i>tt_type</i>=MKL_STAGGERED_COSINE_TRANSFORM, the transform uses only the first $3n/2$ elements, which contain tabulated sine and cosine values.

Index	Description
	If <code>tt_type=MKL_STAGGERED2_COSINE_TRANSFORM</code>, the transform uses all the $5n/2$ elements, which contain tabulated sine and cosine values.



NOTE. To save memory, you can define the array size depending upon the type of transform.

Caveat on Parameter Modifications

Flexibility of the TT interface enables you to skip calling the `?_init_trig_transform` routine and to initialize the basic data structures explicitly in your code. You may also need to modify the contents of `ipar`, `dpar` and `spar` arrays after initialization. When doing so, provide correct and consistent data in the arrays. Mistakenly altered arrays cause errors or wrong computation. You can perform a basic check for correctness and consistency of parameters by calling the `?_commit_trig_transform` routine; however, this does not ensure the correct result of a transform but only reduces the chance of errors or wrong results.



NOTE. To supply correct and consistent parameters to TT routines, you should have considerable experience in using the TT interface and good understanding of elements that the `ipar`, `spar` and `dpar` arrays contain and dependencies between values of these elements.

However, in rare occurrences, even advanced users might fail to compute a transform using TT routines after the parameter modifications. In cases like these, refer for technical support at <http://www.intel.com/software/products/support/> .



WARNING. The only way that ensures proper computation of the Trigonometric Transforms is to follow a typical sequence of invoking the routines and not change the default set of parameters. So, avoid modifications of `ipar`, `dpar` and `spar` arrays unless a strong need arises.

Implementation Details

Several aspects of the Intel MKL TT interface are platform-specific and language-specific. To promote portability across platforms and ease of use across different languages, users are provided with the TT language-specific header files to include in their code. Currently, the following of them are available:

- `mkl_trig_transforms.h`, to be used together with `mkl_dfti.h`, for C programs.
- `mkl_trig_transforms.f90`, to be used together with `mkl_dfti.f90`, for Fortran 90 programs.



NOTE. Use of the Intel MKL TT software without including one of the above header files is not supported.

C-specific Header File

The C-specific header file defines the following function prototypes:

```
void d_init_trig_transform(int *, int *, int *, double *, int *);
void d_commit_trig_transform(double *, DFTI_DESCRIPTOR_HANDLE *, int *, double *, int *);

void d_forward_trig_transform(double *, DFTI_DESCRIPTOR_HANDLE *, int *, double *, int *);
void d_backward_trig_transform(double *, DFTI_DESCRIPTOR_HANDLE *, int *, double *, int *);

void s_init_trig_transform(int *, int *, int *, float *, int *);
void s_commit_trig_transform(float *, DFTI_DESCRIPTOR_HANDLE *, int *, float *, int *);

void s_forward_trig_transform(float *, DFTI_DESCRIPTOR_HANDLE *, int *, float *, int *);
void s_backward_trig_transform(float *, DFTI_DESCRIPTOR_HANDLE *, int *, float *, int *);

void free_trig_transform(DFTI_DESCRIPTOR_HANDLE *, int *, int *);
```

Fortran-Specific Header File

The Fortran90-specific header file defines the following function prototypes:

```

SUBROUTINE D_INIT_TRIG_TRANSFORM(n, tt_type, ipar, dpar, stat)
    INTEGER, INTENT(IN) :: n, tt_type
    INTEGER, INTENT(INOUT) :: ipar(*)
    REAL(8), INTENT(INOUT) :: dpar(*)
    INTEGER, INTENT(OUT) :: stat
END SUBROUTINE D_INIT_TRIG_TRANSFORM

SUBROUTINE D_COMMIT_TRIG_TRANSFORM(f, handle, ipar, dpar, stat)
    REAL(8), INTENT(INOUT) :: f(*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: handle
    INTEGER, INTENT(INOUT) :: ipar(*)
    REAL(8), INTENT(INOUT) :: dpar(*)
    INTEGER, INTENT(OUT) :: stat
END SUBROUTINE D_COMMIT_TRIG_TRANSFORM

SUBROUTINE D_FORWARD_TRIG_TRANSFORM(f, handle, ipar, dpar, stat)
    REAL(8), INTENT(INOUT) :: f(*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: handle
    INTEGER, INTENT(INOUT) :: ipar(*)
    REAL(8), INTENT(INOUT) :: dpar(*)
    INTEGER, INTENT(OUT) :: stat
END SUBROUTINE D_FORWARD_TRIG_TRANSFORM

SUBROUTINE D_BACKWARD_TRIG_TRANSFORM(f, handle, ipar, dpar, stat)
    REAL(8), INTENT(INOUT) :: f(*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: handle

```

```

        INTEGER, INTENT(INOUT) :: ipar(*)
        REAL(8), INTENT(INOUT) :: dpar(*)
        INTEGER, INTENT(OUT) :: stat
END SUBROUTINE D_BACKWARD_TRIG_TRANSFORM

SUBROUTINE S_INIT_TRIG_TRANSFORM(n, tt_type, ipar, spar, stat)
    INTEGER, INTENT(IN) :: n, tt_type
    INTEGER, INTENT(INOUT) :: ipar(*)
    REAL(4), INTENT(INOUT) :: spar(*)
    INTEGER, INTENT(OUT) :: stat
END SUBROUTINE S_INIT_TRIG_TRANSFORM

SUBROUTINE S_COMMIT_TRIG_TRANSFORM(f, handle, ipar, spar, stat)
    REAL(4), INTENT(INOUT) :: f(*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: handle
    INTEGER, INTENT(INOUT) :: ipar(*)
    REAL(4), INTENT(INOUT) :: spar(*)
    INTEGER, INTENT(OUT) :: stat
END SUBROUTINE S_COMMIT_TRIG_TRANSFORM

SUBROUTINE S_FORWARD_TRIG_TRANSFORM(f, handle, ipar, spar, stat)
    REAL(4), INTENT(INOUT) :: f(*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: handle
    INTEGER, INTENT(INOUT) :: ipar(*)
    REAL(4), INTENT(INOUT) :: spar(*)
    INTEGER, INTENT(OUT) :: stat
END SUBROUTINE S_FORWARD_TRIG_TRANSFORM

```

```

SUBROUTINE S_BACKWARD_TRIG_TRANSFORM(f, handle, ipar, spar, stat)
    REAL(4), INTENT(INOUT) :: f(*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: handle
    INTEGER, INTENT(INOUT) :: ipar(*)
    REAL(4), INTENT(INOUT) :: spar(*)
    INTEGER, INTENT(OUT) :: stat
END SUBROUTINE S_BACKWARD_TRIG_TRANSFORM

```

```

SUBROUTINE FREE_TRIG_TRANSFORM(handle, ipar, stat)
    INTEGER, INTENT(INOUT) :: ipar(*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: handle
    INTEGER, INTENT(OUT) :: stat
END SUBROUTINE FREE_TRIG_TRANSFORM

```

Fortran 90 specifics of the TT routines usage are similar for all Intel MKL PDE support tools and described in the [Calling PDE Support Routines from Fortran 90](#) section.

Poisson Library Routines

In addition to Real Discrete Trigonometric Transforms (TT) interface (refer to [Trigonometric Transform Routines](#)), Intel® MKL supports the Poisson Library interface, referred to as PL interface. The interface implements a group of routines (PL routines) used to compute a solution of Laplace, Poisson, and Helmholtz problems of special kind using discrete Fourier transforms. Laplace and Poisson problems are special cases of a more general Helmholtz problem. The problems being solved are defined more exactly in the [Poisson Library Implemented](#) subsection. The PL interface provides much flexibility of use: you can adjust routines to your particular needs at the cost of manual tuning routine parameters or just call routines with default parameter values. The interface can adjust style of error and warning messages to C or Fortran notations by setting up a dedicated parameter. This adds convenience to debugging, because users can read information in the way that is natural for their code. The Intel MKL PL interface currently contains only routines that implement the following solvers:

- Fast Laplace, Poisson and Helmholtz solvers in a Cartesian coordinate system
- Fast Poisson and Helmholtz solvers in a spherical coordinate system.

To describe the Intel MKL PL interface, the C convention is used. Fortran usage specifics can be found in the [Calling PDE Support Routines from Fortran 90](#) section.

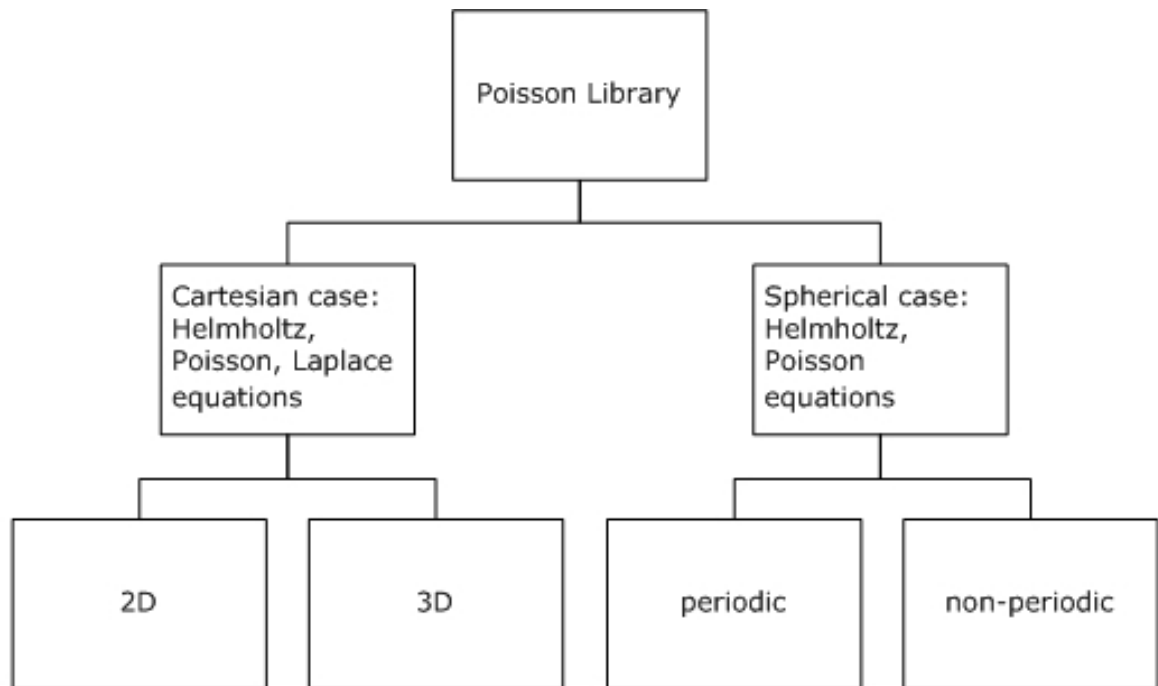


NOTE. Fortran users should mind that respective array indices in Fortran increase by 1.

Poisson Library Implemented

PL routines enable approximate solving of certain two-dimensional and three-dimensional problems. [Figure 13-2](#) shows the general structure of the Poisson Library.

Figure 13-2 Structure of the Poisson Library



Sections below provide details of the problems that can be solved using Intel MKL PL.

Two-Dimensional Problems

Notational Conventions

The PL interface description uses the following notation for boundaries of a rectangular domain $a_x < x < b_x, a_y < y < b_y$ on a Cartesian plane:

$$bd\_a_x = \{x = a_x, a_y \leq y \leq b_y\}, bd\_b_x = \{x = b_x, a_y \leq y \leq b_y\}$$

$$bd\_a_y = \{a_x \leq x \leq b_x, y = a_y\}, bd\_b_y = \{a_x \leq x \leq b_x, y = b_y\}.$$

The wildcard "+" may stand for any of the symbols a_x, b_x, a_y, b_y , so that bd_+ denotes any of the above boundaries.

The PL interface description uses the following notation for boundaries of a rectangular domain $a_\phi < \phi < b_\phi, a_\theta < \theta < b_\theta$ on a sphere $0 \leq \phi \leq 2\pi, 0 \leq \theta \leq \pi$:

$$bd\_a_\phi = \{\phi = a_\phi, a_\theta \leq \theta \leq b_\theta\}, bd\_b_\phi = \{\phi = b_\phi, a_\theta \leq \theta \leq b_\theta\}$$

$$bd\_a_\theta = \{a_\phi \leq \phi \leq b_\phi, \theta = a_\theta\}, bd\_b_\theta = \{a_\phi \leq \phi \leq b_\phi, \theta = b_\theta\}.$$

The wildcard "~" may stand for any of the symbols $a_\phi, b_\phi, a_\theta, b_\theta$, so that $bd\_~$ denotes any of the above boundaries.

Two-dimensional (2D) Helmholtz problem on a Cartesian plane

The 2D Helmholtz problem is to find an approximate solution of the Helmholtz equation

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} + qu = f(x, y), q = \text{const} \geq 0$$

in a rectangle, that is, a rectangular domain $a_x < x < b_x, a_y < y < b_y$, with one of the following boundary conditions on each boundary bd_+ :

- The Dirichlet boundary condition

$$u(x, y) = G(x, y)$$

- The Neumann boundary condition

$$\frac{\partial u}{\partial n}(x, y) = g(x, y)$$

where

$n = -x$ on bd_a_x , $n = x$ on bd_b_x ,

$n = -y$ on bd_a_y , $n = y$ on bd_b_y .

Two-dimensional (2D) Poisson problem on a Cartesian plane

The Poisson problem is a special case of the Helmholtz problem, when $q=0$. The 2D Poisson problem is to find an approximate solution of the Poisson equation

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = f(x, y)$$

in a rectangle $a_x < x < b_x$, $a_y < y < b_y$ with the Dirichlet or Neumann boundary condition on each boundary bd_+ . In case of a problem with the Neumann boundary condition on the entire boundary, you can find the solution of the problem only up to a constant. In this case, the Poisson Library will compute the solution that provides the minimal Euclidean norm of a residual.

Two-dimensional (2D) Laplace problem on a Cartesian plane

The Laplace problem is a special case of the Helmholtz problem, when $q=0$ and $f(x, y)=0$. The 2D Laplace problem is to find an approximate solution of the Laplace equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

in a rectangle $a_x < x < b_x, a_y < y < b_y$ with the Dirichlet or Neumann boundary condition on each boundary $bd_{\sim}$.

Helmholtz problem on a sphere

The Helmholtz problem on a sphere is to find an approximate solution of the Helmholtz equation

$$-\Delta_s u + qu = f, \quad q = const \geq 0,$$
$$\Delta_s = \frac{1}{\sin^2 \theta} \frac{\partial^2}{\partial \phi^2} + \frac{1}{\sin \theta} \frac{\partial}{\partial \theta} \left(\sin \theta \frac{\partial}{\partial \theta} \right)$$

in a spherical rectangle that is, a domain bounded by angles $a_\phi \leq \phi \leq b_\phi, a_\theta \leq \theta \leq b_\theta$, with boundary conditions for particular domains listed in Table 13-4.

Table 13-4 Details of Helmholtz Problem on a Sphere

Domain on a sphere	Boundary condition	Periodic/non-periodic case
Rectangular, that is, $b_\phi - a_\phi < 2 \pi$ and $b_\theta - a_\theta < \pi$	Homogeneous Dirichlet boundary conditions on each boundary $bd_{\sim}$	<i>non-periodic</i>
Where $a_\phi = 0, b_\phi = 2 \pi$, and $b_\theta - a_\theta < \pi$	Homogeneous Dirichlet boundary conditions on the boundaries bd_{a_θ} and bd_{b_θ}	<i>periodic</i>
Entire sphere, that is, $a_\phi = 0, b_\phi = 2 \pi, a_\theta = 0$, and $b_\theta = \pi$	Boundary condition	<i>periodic</i>

$$\left(\sin \theta \frac{\partial u}{\partial \theta} \right)_{\substack{\theta \rightarrow 0 \\ \theta \rightarrow \pi}} = 0$$

at the poles.

Poisson problem on a sphere

The Poisson problem is a special case of the Helmholtz problem, when $q=0$. The Poisson problem on a sphere is to find an approximate solution of the Poisson equation

$$-\Delta_s u = f, \quad \Delta_s = \frac{1}{\sin^2 \theta} \frac{\partial^2}{\partial \varphi^2} + \frac{1}{\sin \theta} \frac{\partial}{\partial \theta} \left(\sin \theta \frac{\partial}{\partial \theta} \right)$$

in a spherical rectangle $a_\phi \leq \phi \leq b_\phi, a_\theta \leq \theta \leq b_\theta$ in cases listed in [Table 13-4](#). The solution to the Poisson problem on the entire sphere can be found up to a constant only. In this case, Poisson Library will compute the solution that provides the minimal Euclidean norm of a residual.

Approximation of 2D problems

To find an approximate solution for any of the 2D problems, a uniform mesh is built in the rectangular domain:

$$\left\{ x_i = a_x + ih_x, y_j = a_y + jh_y \right\},$$

$$i = 0, \dots, n_x, j = 0, \dots, n_y, h_x = \frac{b_x - a_x}{n_x}, h_y = \frac{b_y - a_y}{n_y}$$

in the Cartesian case and

$$\left\{ \varphi_i = a_\varphi + ih_\varphi, \theta_j = a_\theta + jh_\theta \right\},$$

$$i = 0, \dots, n_\varphi, j = 0, \dots, n_\theta, h_\varphi = \frac{b_\varphi - a_\varphi}{n_\varphi}, h_\theta = \frac{b_\theta - a_\theta}{n_\theta}$$

in the spherical case.

Poisson Library uses the standard five-point finite difference approximation on this mesh to compute the approximation to the solution:

- In the Cartesian case, the values of the approximate solution will be computed in the mesh points (x_i, y_j) provided that the user knows the values of the right-hand side $f(x, y)$ in these points and the values of the appropriate boundary functions $G(x, y)$ and/or $g(x, y)$ in the mesh points laying on the boundary of the rectangular domain.
- In the spherical case, the values of the approximate solution will be computed in the mesh points (ϕ_i, θ_j) provided that the user knows the values of the right-hand side $f(\phi, \theta)$ in these points.



NOTE. The number of mesh intervals n_ϕ in the ϕ direction of a spherical mesh must be even in the periodic case. The current implementation of the Poisson Library does not support meshes with the number of intervals that does not meet this condition.

Three-Dimensional Problems

Notational Conventions

The PL interface description uses the following notation for boundaries of a parallelepiped domain $a_x < x < b_x, a_y < y < b_y, a_z < z < b_z$:

$$bd\_a_x = \{x = a_x, a_y \leq y \leq b_y, a_z \leq z \leq b_z\}, bd\_b_x = \{x = b_x, a_y \leq y \leq b_y, a_z \leq z \leq b_z\}$$

$$bd\_a_y = \{a_x \leq x \leq b_x, y = a_y, a_z \leq z \leq b_z\}, bd\_b_y = \{a_x \leq x \leq b_x, y = b_y, a_z \leq z \leq b_z\}$$

$$bd\_a_z = \{a_x \leq x \leq b_x, a_y \leq y \leq b_y, z = a_z\}, bd\_b_z = \{a_x \leq x \leq b_x, a_y \leq y \leq b_y, z = b_z\}.$$

The wildcard "+" may stand for any of the symbols $a_x, b_x, a_y, b_y, a_z, b_z$, so that bd_+ denotes any of the above boundaries.

Three-dimensional (3D) Helmholtz problem

The 3D Helmholtz problem is to find an approximate solution of the Helmholtz equation

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} - \frac{\partial^2 u}{\partial z^2} + qu = f(x, y, z), q = \text{const} \geq 0$$

in a parallelepiped, that is, a parallelepiped domain $a_x < x < b_x, a_y < y < b_y, a_z < z < b_z$, with one of the following boundary conditions on each boundary bd_+ :

- The Dirichlet boundary condition

$$u(x, y, z) = G(x, y, z)$$

- The Neumann boundary condition

$$\frac{\partial u}{\partial n}(x, y, z) = g(x, y, z)$$

where

$$n = -x \text{ on } bd\_a_x, n = x \text{ on } bd\_b_x,$$

$$n = -y \text{ on } bd\_a_y, n = y \text{ on } bd\_b_y,$$

$$n = -z \text{ on } bd\_a_z, n = z \text{ on } bd\_b_z.$$

Three-dimensional (3D) Poisson problem

The Poisson problem is a special case of the Helmholtz problem, when $q=0$. The 3D Poisson problem is to find an approximate solution of the Poisson equation

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} - \frac{\partial^2 u}{\partial z^2} = f(x, y, z)$$

in a parallelepiped $a_x < x < b_x$, $a_y < y < b_y$, $a_z < z < b_z$ with Dirichlet or Neumann boundary condition on each boundary $bd_{\pm}$.

Three-dimensional (3D) Laplace problem

The Laplace problem is a special case of the Helmholtz problem, when $q=0$ and $f(x, y, z)=0$. The 3D Laplace problem is to find an approximate solution of the Laplace equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 0$$

in a parallelepiped $a_x < x < b_x$, $a_y < y < b_y$, $a_z < z < b_z$ with the Dirichlet or Neumann boundary condition on each boundary $bd_{\pm}$.

Approximation of 3D problems

To find an approximate solution for each of the 3D problems, a uniform mesh is built in the parallelepiped domain

$$\{x_i = a_x + ih_x, y_j = a_y + jh_y, z_k = a_z + kh_z\}$$

where

$$i = 0, \dots, n_x, j = 0, \dots, n_y, k = 0, \dots, n_z$$

$$h_x = \frac{b_x - a_x}{n_x}, h_y = \frac{b_y - a_y}{n_y}, h_z = \frac{b_z - a_z}{n_z}$$

The Poisson Library uses the standard seven-point finite difference approximation on this mesh to compute the approximation to the solution. The values of the approximate solution will be computed in the mesh points (x_i, y_j, z_k) , provided that the user knows the values of the right-hand side $f(x, y, z)$ in these points and the values of the appropriate boundary functions $G(x, y, z)$ and/or $g(x, y, z)$ in the mesh points laying on the boundary of the parallelepiped domain.

Sequence of Invoking PL Routines



NOTE. This description always considers the solution process for the Helmholtz problem, because Fast Poisson Solver and Fast Laplace Solver are special cases of Fast Helmholtz Solver (see [Poisson Library Implemented](#)).

Computation of a solution of the Helmholtz problem using the PL interface is conceptually divided into four steps, each of which is performed via a dedicated routine. [Table 13-5](#) lists the routines and briefly describes their purpose.

Most PL routines have versions operating with single-precision and double-precision data. Names of such routines begin respectively with "s" and "d". The wildcard "?" stands for either of these symbols in routine names. The routines for Cartesian coordinate system have 2D and 3D versions. Their names end respectively in "2D" and "3D". The routines for spherical coordinate system have periodic and non-periodic versions. Their names end respectively in "p" and "np"

Table 13-5 PL Interface Routines

Routine	Description
<code>?_init_Helmholtz_2D/?_init_Helmholtz_3D/ ?_init_sph_p/?_init_sph_np</code>	Initializes basic data structures of Poisson Library for Fast Helmholtz Solver in the 2D/3D/periodic/non-periodic case, respectively.
<code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D/ ?_commit_sph_p/?_commit_sph_np</code>	Checks consistency and correctness of user's data, creates and initializes data structures to be used by the Intel MKL FFT interface ¹ , as well as other data structures needed for the solver.

Routine	Description
<code>?_Helmholtz_2D/?_Helmholtz_3D/ ?_sph_p/?_sph_np</code>	Computes an approximate solution of 2D/3D/periodic/non-periodic Helmholtz problem (see Poisson Library Implemented) specified by the parameters.
<code>free_Helmholtz_2D/free_Helmholtz_3D/ free_sph_p/free_sph_np</code>	Cleans the memory used by the data structures needed for calling the Intel MKL FFT interface ¹ .

¹ PL routines call the Intel MKL FFT interface for better performance.

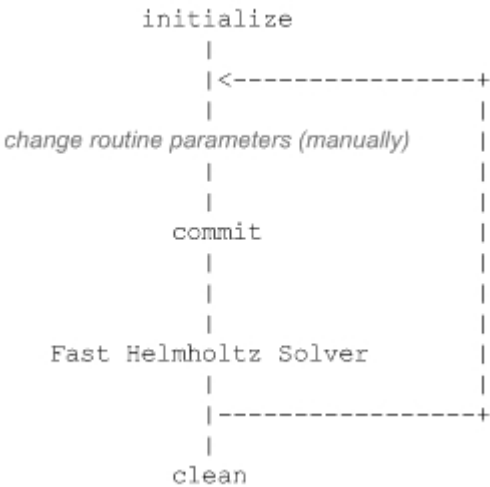
To find an approximate solution of Helmholtz problem only once, the Intel MKL PL interface routines are normally invoked in the order in which they are listed in [Table 13-5](#).



NOTE. Though the order of invoking PL routines may be changed, it is highly recommended to follow the above order of routine calls.

The diagram in [Figure 13-3](#) indicates the typical order in which PL routines can be invoked in a general case.

Figure 13-3 Typical Order of Invoking PL Routines



A general scheme of using PL routines for double-precision computations in a 3D Cartesian case is shown below. A similar scheme holds for single-precision computations with the only difference in the initial letter of routine names. The general scheme in a 2D Cartesian case differs from the one below in the set of routine parameters and the ending of routine names: "2D" instead of "3D".

```
...
d_init_Helmholtz_3D(&ax, &bx, &ay, &by, &az, &bz, &nx, &ny, &nz, Bctype, ipar, dpar, &stat);
/* change parameters in ipar and/or dpar if necessary. */
/* note that the result of the Fast Helmholtz Solver will be in f! If you want to keep the
data stored in f,
save it before the function call below */
d_commit_Helmholtz_3D(f, bd_ax, bd_bx, bd_ay, bd_by, bd_az, bd_bz, &xhandle, &yhandle, ipar,
dpar, &stat);
d_Helmholtz_3D(f, bd_ax, bd_bx, bd_ay, bd_by, bd_az, bd_bz, &xhandle, &yhandle, ipar, dpar,
&stat);
free_Helmholtz_3D (&xhandle, &yhandle, ipar, &stat);
/* here you may clean the memory used by f, dpar, ipar */
...
```

A general scheme of using PL routines for double-precision computations in a spherical periodic case is shown below. Similar scheme holds for single-precision computations with the only difference in the initial letter of routine names. The general scheme in a spherical non-periodic case differs from the one below in the set of routine parameters and the ending of routine names: "np" instead of "p".

```
...
d_init_sph_p(&ap, &bp, &at, &bt, &np, &nt, &q, ipar, dpar, &stat);
/* change parameters in ipar and/or dpar if necessary. */
/* note that the result of the Fast Helmholtz Solver will be in f! If you want to
keep the data stored in f, save it before the function call below */
d_commit_sph_p(f, &handle_s, &handle_c, ipar, dpar, &stat);
d_sph_p(f, &handle_s, &handle_c, ipar, dpar, &stat);
free_sph_p(&handle_s, &handle_c, ipar, &stat);
/* here you may clean the memory used by f, dpar, ipar */
...
```


You can find examples of Fortran 90 and C code that use PL routines to solve Helmholtz problem (in both Cartesian and spherical cases) in the “[Poisson Library Code Examples](#)” section in Appendix C.

Interface Description

All types in this documentation are standard C types: `int`, `float`, and `double`. Fortran 90 users can call the routines with `INTEGER`, `REAL`, and `DOUBLE PRECISION` Fortran types, respectively (see examples in the “[Poisson Library Code Examples](#)” section in Appendix C).

The interface description uses the built-in type `int` for integer values. If you employ the ILP64 interface, read this type as `long long int` (or `INTEGER*8` for Fortran). For more information, refer to the *Intel MKL User's Guide*.

Routine Options

All PL routines use parameters for passing various options to the routines. These parameters are arrays `ipar`, `dpar` and `spar`. Values for these parameters should be specified very carefully (see [Common Parameters](#)). You can change these values during computations to meet your needs.



WARNING. To avoid failure or wrong results, you must provide correct and consistent parameters to the routines.

User Data Arrays

PL routines take arrays of user data as input. For example, user arrays are passed to the routine `d_Helmholtz_3D` to compute an approximate solution to 3D Helmholtz problem. To minimize storage requirements and improve the overall run-time efficiency, Intel MKL PL routines do not make copies of user input arrays.



NOTE. If you need a copy of your input data arrays, save them yourself.

PL Routines for the Cartesian Solver

The section gives detailed description of Cartesian PL routines, their syntax, parameters and values they return. All flavors of the same routine, namely, double-precision and single-precision, 2D and 3D, are described together.



NOTE. Some of the routine parameters are used only in the 3D Fast Helmholtz Solver.

PL routines call the Intel MKL FFT interface (described in section “[FFT Functions](#)” in chapter “Fast Fourier Transforms”), which enhances performance of the routines.

?_init_Helmholtz_2D/?_init_Helmholtz_3D

Initializes basic data structures of the Fast 2D/3D Helmholtz Solver.

Syntax

```
void d_init_Helmholtz_2D(double* ax, double* bx, double* ay, double* by,
int* nx, int* ny, char* BCtype, double* q, int* ipar, double* dpar, int*
stat);

void s_init_Helmholtz_2D(float* ax, float* bx, float* ay, float* by, int*
nx, int* ny, char* BCtype, float* q, int* ipar, float* spar, int* stat);

void d_init_Helmholtz_3D(double* ax, double* bx, double* ay, double* by,
double* az, double* bz, int* nx, int* ny, int* nz, char* BCtype, double* q,
int* ipar, double* dpar, int* stat);

void s_init_Helmholtz_3D(float* ax, float* bx, float* ay, float* by, float*
az, float* bz, int* nx, int* ny, int* nz, char* BCtype, float* q, int* ipar,
float* spar, int* stat);
```

Input Parameters

<code>ax</code>	double* for d_init_Helmholtz_2D/d_init_Helmholtz_3D, float* for s_init_Helmholtz_2D/s_init_Helmholtz_3D. The coordinate of the leftmost boundary of the domain along x-axis.

<i>bx</i>	double* for d_init_Helmholtz_2D/d_init_Helmholtz_3D, float* for s_init_Helmholtz_2D/s_init_Helmholtz_3D. The coordinate of the rightmost boundary of the domain along x-axis.
<i>ay</i>	double* for d_init_Helmholtz_2D/d_init_Helmholtz_3D, float* for s_init_Helmholtz_2D/s_init_Helmholtz_3D. The coordinate of the leftmost boundary of the domain along y-axis.
<i>by</i>	double* for d_init_Helmholtz_2D/d_init_Helmholtz_3D, float* for s_init_Helmholtz_2D/s_init_Helmholtz_3D. The coordinate of the rightmost boundary of the domain along y-axis.
<i>az</i>	double* for d_init_Helmholtz_2D/d_init_Helmholtz_3D, float* for s_init_Helmholtz_2D/s_init_Helmholtz_3D. The coordinate of the leftmost boundary of the domain along z-axis. This parameter is needed only for the ?_init_Helmholtz_3D routine.
<i>bz</i>	double* for d_init_Helmholtz_2D/d_init_Helmholtz_3D, float* for s_init_Helmholtz_2D/s_init_Helmholtz_3D. The coordinate of the rightmost boundary of the domain along z-axis. This parameter is needed only for the ?_init_Helmholtz_3D routine.
<i>nx</i>	int*. The number of mesh intervals along x-axis.
<i>ny</i>	int*. The number of mesh intervals along y-axis.
<i>nz</i>	int*. The number of mesh intervals along z-axis. This parameter is needed only for the ?_init_Helmholtz_3D routine.
<i>BCtype</i>	char*. Contains the type of boundary conditions on each boundary. Must contain four characters for ?_init_Helmholtz_2D and six characters for ?_init_Helmholtz_3D. Each of the characters can be 'N' (Neumann boundary condition) or 'D' (Dirichlet boundary

condition). Types of boundary conditions for the boundaries should be specified in the following order: *bd_ax*, *bd_bx*, *bd_ay*, *bd_by*, *bd_az*, *bd_bz*. Boundary condition types for the last two boundaries should be specified only in the 3D case.

q double* for
 d_init_Helmholtz_2D/d_init_Helmholtz_3D,
 float* for s_init_Helmholtz_2D/s_init_Helmholtz_3D

.
 The constant Helmholtz coefficient. Note that to solve Poisson or Laplace problem, you should set the value of *q* to 0.

Output Parameters

ipar int array of size 128. Contains integer data to be used by Fast Helmholtz Solver (for details, refer to [Common Parameters](#)).

dpar double array of size $5 \cdot n_x / 2 + 7$ in the 2D case or $5 \cdot (n_x + n_y) / 2 + 9$ in the 3D case. Contains double-precision data to be used by Fast Helmholtz Solver (for details, refer to [Common Parameters](#)).

spar float array of size $5 \cdot n_x / 2 + 7$ in the 2D case or $5 \cdot (n_x + n_y) / 2 + 9$ in the 3D case. Contains single-precision data to be used by Fast Helmholtz Solver (for details, refer to [Common Parameters](#)).

stat int*. Routine completion status, which is also written to *ipar[0]*. The status should be 0 to proceed to other PL routines.

Description

The routines `?_init_Helmholtz_2D/?_init_Helmholtz_3D` are called to initialize basic data structures for Poisson Library computations of the appropriate precision. All routines invoked after a call to a `?_init_Helmholtz_2D/?_init_Helmholtz_3D` routine use values of the *ipar*, *dpar* and *spar* array parameters returned by the routine. Detailed description of the array parameters can be found in [Common Parameters](#).



WARNING. Data structures initialized and created by 2D/3D flavors of the routine cannot be used by 3D/2D flavors of any PL routines, respectively.

You can skip calling this routine in your code. However, see [Caveat on Parameter Modifications](#) before doing so.

Return Values

`stat= 0`

The routine successfully completed the task. In general, to proceed with computations, the routine should complete with this `stat` value.

`stat= -99999`

The routine failed to complete the task because of a fatal error.

?_commit_Helmholtz_2D/?_commit_Helmholtz_3D

Checks consistency and correctness of user's data as well as initializes certain data structures required to solve 2D/3D Helmholtz problem.

Syntax

```
void d_commit_Helmholtz_2D(double* f, double* bd_ax, double* bd_bx, double*
bd_ay, double* bd_by, DFTI_DESCRIPTOR* xhandle, int* ipar, double* dpar, int*
stat);
```

```
void s_commit_Helmholtz_2D (float* f, float* bd_ax, float* bd_bx, float*
bd_ay, float* bd_by, DFTI_DESCRIPTOR* xhandle, int* ipar, float* spar, int*
stat);
```

```
void d_commit_Helmholtz_3D(double* f, double* bd_ax, double* bd_bx, double*
bd_ay, double* bd_by, double* bd_az, double* bd_bz, DFTI_DESCRIPTOR* xhandle,
DFTI_DESCRIPTOR* yhandle, int* ipar, double* dpar, int* stat);
```

```
void s_commit_Helmholtz_3D(float* f, float* bd_ax, float* bd_bx, float*
bd_ay, float* bd_by, float* bd_az, float* bd_bz, DFTI_DESCRIPTOR* xhandle,
DFTI_DESCRIPTOR* yhandle, int* ipar, float* spar, int* stat);
```

Input Parameters

<i>f</i>	double* for d_commit_Helmholtz_2D/d_commit_Helmholtz_3D, float* for s_commit_Helmholtz_2D/s_commit_Helmholtz_3D. Contains the right-hand side of the problem packed in a single vector. The size of the vector for the 2D problem is $(nx+1)*(ny+1)$. In this case, the value of the right-hand side in the mesh point (i, j) is stored in $f[i+j*(nx+1)]$. The size of the vector for the 3D problem is $(nx+1)*(ny+1)*(nz+1)$. In this case, value of the right-hand side in the mesh point (i, j, k) is stored in $f[i+j*(nx+1)+k*(nx+1)*(ny+1)]$. Note that to solve the Laplace problem, you should set all the elements of the array <i>f</i> to 0. Note also that the array <i>f</i> may be altered by the routine. To preserve the vector, save it in another memory location.
<i>ipar</i>	int array of size 128. Contains integer data to be used by Fast Helmholtz Solver (for details, refer to Common Parameters).
<i>dpar</i>	double array of size $5*nx/2+7$ in the 2D case or $5*(nx+ny)/2+9$ in the 3D case. Contains double-precision data to be used by Fast Helmholtz Solver (for details, refer to Common Parameters).
<i>spar</i>	float array of size $5*nx/2+7$ in the 2D case or $5*(nx+ny)/2+9$ in the 3D case. Contains single-precision data to be used by Fast Helmholtz Solver (for details, refer to Common Parameters).
<i>bd_ax</i>	double* for d_commit_Helmholtz_2D/d_commit_Helmholtz_3D, float* for s_commit_Helmholtz_2D/s_commit_Helmholtz_3D. Contains values of the boundary condition on the leftmost boundary of the domain along <i>x</i>-axis.

For `?_commit_Helmholtz_2D`, the size of the array is $ny+1$. In case of the Dirichlet boundary condition (value of `BCtype[0]` is 'D'), it contains values of the function $G(ax, y_j)$, $j=0, \dots, ny$. In case of the Neumann boundary condition (value of `BCtype[0]` is 'N'), it contains values of the function $g(ax, y_j)$, $j=0, \dots, ny$. The value corresponding to the index j is placed in `bd_ax[j]`.

For `?_commit_Helmholtz_3D`, the size of the array is $(ny+1)*(nz+1)$. In case of the Dirichlet boundary condition (value of `BCtype[0]` is 'D'), it contains values of the function $G(ax, y_j, z_k)$, $j=0, \dots, ny, k=0, \dots, nz$. In case of the Neumann boundary condition (value of `BCtype[0]` is 'N'), it contains the values of the function $g(ax, y_j, z_k)$, $j=0, \dots, ny, k=0, \dots, nz$. The values are packed in the array so that the value corresponding to indices (j, k) is placed in `bd_ax[j+k*(ny+1)]`.

`bd_bx`

`double* for`
`d_commit_Helmholtz_2D/d_commit_Helmholtz_3D,`
`float* for`
`s_commit_Helmholtz_2D/s_commit_Helmholtz_3D.`
 Contains values of the boundary condition on the rightmost boundary of the domain along x -axis.

For `?_commit_Helmholtz_2D`, the size of the array is $ny+1$. In case of the Dirichlet boundary condition (value of `BCtype[1]` is 'D'), it contains values of the function $G(bx, y_j)$, $j=0, \dots, ny$. In case of the Neumann boundary condition (value of `BCtype[1]` is 'N'), it contains values of the function $g(bx, y_j)$, $j=0, \dots, ny$. The value corresponding to the index j is placed in `bd_bx[j]`.

For `?_commit_Helmholtz_3D`, the size of the array is $(ny+1)*(nz+1)$. In case of the Dirichlet boundary condition (value of `BCtype[1]` is 'D'), it contains values of the function $G(bx, y_j, z_k)$, $j=0, \dots, ny, k=0, \dots, nz$. In case of the Neumann boundary condition (value of `BCtype[1]` is 'N'), it contains the values of the function $g(bx, y_j, z_k)$, $j=0, \dots, ny, k=0, \dots, nz$. The values are packed in the array so that the value corresponding to indices (j, k) is placed in `bd_bx[j+k*(ny+1)]`.

bd_ay

double* for
d_commit_Helmholtz_2D/d_commit_Helmholtz_3D,
float* for
s_commit_Helmholtz_2D/s_commit_Helmholtz_3D.
Contains values of the boundary condition on the leftmost boundary of the domain along *y*-axis.
For ?_commit_Helmholtz_2D, the size of the array is *nx*+1. In case of the Dirichlet boundary condition (value of *BCtype*[2] is 'D'), it contains values of the function $G(x_i, ay)$, $i=0, \dots, nx$. In case of the Neumann boundary condition (value of *BCtype*[2] is 'N'), it contains values of the function $g(x_i, ay)$, $i=0, \dots, nx$. The value corresponding to the index *i* is placed in *bd_ay*[*i*].
For ?_commit_Helmholtz_3D, the size of the array is $(nx+1)*(nz+1)$. In case of the Dirichlet boundary condition (value of *BCtype*[2] is 'D'), it contains values of the function $G(x_i, ay, z_k)$, $i=0, \dots, nx, k=0, \dots, nz$. In case of the Neumann boundary condition (value of *BCtype*[2] is 'N'), it contains the values of the function $g(x_i, ay, z_k)$, $i=0, \dots, nx, k=0, \dots, nz$. The values are packed in the array so that the value corresponding to indices (*i, k*) is placed in *bd_ay*[*i+k*(nx+1)*].

bd_by

double* for
d_commit_Helmholtz_2D/d_commit_Helmholtz_3D,
float* for
s_commit_Helmholtz_2D/s_commit_Helmholtz_3D.
Contains values of the boundary condition on the rightmost boundary of the domain along *y*-axis.
For ?_commit_Helmholtz_2D, the size of the array is *nx*+1. In case of the Dirichlet boundary condition (value of *BCtype*[3] is 'D'), it contains values of the function $G(x_i, by)$, $i=0, \dots, nx$. In case of the Neumann boundary condition (value of *BCtype*[3] is 'N'), it contains values of the function $g(x_i, by)$, $i=0, \dots, nx$. The value corresponding to the index *i* is placed in *bd_by*[*i*].
For ?_commit_Helmholtz_3D, the size of the array is $(nx+1)*(nz+1)$. In case of the Dirichlet boundary condition (value of *BCtype*[3] is 'D'), it contains values of the function

$G(x_i, by, z_k)$, $i=0, \dots, nx$, $k=0, \dots, nz$. In case of the Neumann boundary condition (value of `BCType[3]` is 'N'), it contains the values of the function $g(x_i, by, z_k)$, $i=0, \dots, nx$, $k=0, \dots, nz$. The values are packed in the array so that the value corresponding to indices (i, k) is placed in `bd_by[i+k*(nx+1)]`.

`bd_az`

`double* for`
`d_commit_Helmholtz_2D/d_commit_Helmholtz_3D,`
`float* for`
`s_commit_Helmholtz_2D/s_commit_Helmholtz_3D.`
 This parameter is needed only for `?_commit_Helmholtz_3D`. Contains values of the boundary condition on the leftmost boundary of the domain along z -axis.
 The size of the array is $(nx+1)*(ny+1)$. In case of the Dirichlet boundary condition (value of `BCType[4]` is 'D'), it contains values of the function $G(x_i, y_j, az)$, $i=0, \dots, nx$, $j=0, \dots, ny$. In case of the Neumann boundary condition (value of `BCType[4]` is 'N'), it contains the values of the function $g(x_i, y_j, az)$, $i=0, \dots, nx$, $j=0, \dots, ny$. The values are packed in the array so that the value corresponding to indices (i, j) is placed in `bd_az[i+j*(nx+1)]`.

`bd_bz`

`double* for`
`d_commit_Helmholtz_2D/d_commit_Helmholtz_3D,`
`float* for`
`s_commit_Helmholtz_2D/s_commit_Helmholtz_3D.`
 This parameter is needed only for `?_commit_Helmholtz_3D`. Contains values of the boundary condition on the rightmost boundary of the domain along z -axis.
 The size of the array is $(nx+1)*(ny+1)$. In case of the Dirichlet boundary condition (value of `BCType[5]` is 'D'), it contains values of the function $G(x_i, y_j, bz)$, $i=0, \dots, nx$, $j=0, \dots, ny$. In case of the Neumann boundary condition (value of `BCType[5]` is 'N'), it contains the values of the function $g(x_i, y_j, bz)$, $i=0, \dots, nx$, $j=0, \dots, ny$. The values are packed in the array so that the value corresponding to indices (i, j) is placed in `bd_bz[i+j*(nx+1)]`.

Output Parameters

<i>f</i>	Vector of the right-hand side of the problem. Possibly, altered on output.
<i>ipar</i>	Contains integer data to be used by Fast Helmholtz Solver. Modified on output as explained in Common Parameters .
<i>dpar</i>	Contains double-precision data to be used by Fast Helmholtz Solver. Modified on output as explained in Common Parameters .
<i>spar</i>	Contains single-precision data to be used by Fast Helmholtz Solver. Modified on output as explained in Common Parameters .
<i>xhandle, yhandle</i>	DESCRIPTOR_HANDLE*. Data structures used by the Intel MKL FFT interface (for details, refer to section “ FFT Functions ” in chapter “Fast Fourier Transforms”). <i>yhandle</i> is used only by <code>?_commit_Helmholtz_3D</code> .
<i>stat</i>	int*. Routine completion status, which is also written to <i>ipar[0]</i> . The status should be 0 to proceed to other PL routines.

Description

The routines `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D` check consistency and correctness of the parameters to be passed to the solver routines `?_Helmholtz_2D/?_Helmholtz_3D`. They also initialize data structures *xhandle, yhandle* as well as arrays *ipar* and *dpar/spar*, depending upon the routine precision. Refer to [Common Parameters](#) to find out which particular array elements the `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D` routines initialize and what values are written there.

The routines perform only a basic check for correctness and consistency. If you are going to modify parameters of PL routines, see the [Caveat on Parameter Modifications](#) section. Unlike `?_init_Helmholtz_2D/?_init_Helmholtz_3D`, the routines `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D` are mandatory, and you cannot skip calling them in your code. Values of *ax, bx, ay, by, az, bz, nx, ny, nz*, and *BCtype* are passed to each of the routines with the *ipar* array and defined in a previous call to the appropriate `?_init_Helmholtz_2D/?_init_Helmholtz_3D` routine.

Return Values

<code>stat= 1</code>	The routine completed without errors and produced some warnings.
<code>stat= 0</code>	The routine successfully completed the task.
<code>stat= -100</code>	The routine stopped because an error in the user's data was found or the data in the <code>dpar</code> , <code>spar</code> or <code>ipar</code> array was altered by mistake.
<code>stat= -1000</code>	The routine stopped because of an Intel MKL FFT or TT interface error.
<code>stat= -10000</code>	The routine stopped because the initialization failed to complete or the parameter <code>ipar[0]</code> was altered by mistake.
<code>stat= -99999</code>	The routine failed to complete the task because of a fatal error.

?_Helmholtz_2D/?_Helmholtz_3D

Computes the solution of 2D/3D Helmholtz problem specified by the parameters.

Syntax

```
void d_Helmholtz_2D(double* f, double* bd_ax, double* bd_bx, double* bd_ay,
double* bd_by, DFTI_DESCRIPTOR* xhandle, int* ipar, double* dpar, int* stat);
void s_Helmholtz_2D (float* f, float* bd_ax, float* bd_bx, float* bd_ay,
float* bd_by, DFTI_DESCRIPTOR* xhandle, int* ipar, float* spar, int* stat);
void d_Helmholtz_3D(double* f, double* bd_ax, double* bd_bx, double* bd_ay,
double* bd_by, double* bd_az, double* bd_bz, DFTI_DESCRIPTOR* xhandle,
DFTI_DESCRIPTOR* yhandle, int* ipar, double* dpar, int* stat);
void s_Helmholtz_3D(float* f, float* bd_ax, float* bd_bx, float* bd_ay,
float* bd_by, float* bd_az, float* bd_bz, DFTI_DESCRIPTOR* xhandle,
DFTI_DESCRIPTOR* yhandle, int* ipar, float* spar, int* stat);
```

Input Parameters

<code>f</code>	double* for d_Helmholtz_2D/d_Helmholtz_3D, float* for s_Helmholtz_2D/s_Helmholtz_3D.

Contains the right-hand side of the problem packed in a single vector and modified by the appropriate

[?_commit_Helmholtz_2D/?_commit_Helmholtz_3D](#) routine. Note that an attempt to substitute the original right-hand side vector at this point will result in a wrong solution.

The size of the vector for the 2D problem is $(nx+1)*(ny+1)$. In this case, value of the right-hand side in the mesh point (i, j) is stored in $f[i+j*(nx+1)]$. The size of the vector for the 3D problem is $(nx+1)*(ny+1)*(nz+1)$. In this case, value of the right-hand side in the mesh point (i, j, k) is stored in $f[i+j*(nx+1)+k*(nx+1)*(ny+1)]$.

<i>xhandle, yhandle</i>	DESCRIPTOR_HANDLE*. Data structures used by the Intel MKL FFT interface (for details, refer to section “ FFT Functions ” in chapter “Fast Fourier Transforms”). <i>yhandle</i> is used only by ?_Helmholtz_3D .
<i>ipar</i>	int array of size 128. Contains integer data to be used by Fast Helmholtz Solver (for details, refer to Common Parameters).
<i>dpar</i>	double array of size $5*nx/2+7$ in the 2D case or $5*(nx+ny)/2+9$ in the 3D case. Contains double-precision data to be used by Fast Helmholtz Solver (for details, refer to Common Parameters).
<i>spar</i>	float array of size $5*nx/2+7$ in the 2D case or $5*(nx+ny)/2+9$ in the 3D case. Contains single-precision data to be used by Fast Helmholtz Solver (for details, refer to Common Parameters).
<i>bd_ax</i>	double* for d_Helmholtz_2D/d_Helmholtz_3D , float* for s_Helmholtz_2D/s_Helmholtz_3D . Contains values of the boundary condition on the leftmost boundary of the domain along x-axis. For ?_Helmholtz_2D , the size of the array is $ny+1$. In case of the Dirichlet boundary condition (value of BCtype[0] is 'D'), it contains values of the function $G(ax, y_j)$, $j=0, \dots, ny$. In case of the Neumann boundary condition (value

of $BCType[0]$ is 'N'), it contains values of the function $g(ax, y_j)$, $j=0, \dots, ny$. The value corresponding to the index j is placed in $bd\_ax[j]$.

For $?\_Helmholtz\_3D$, the size of the array is $(ny+1)*(nz+1)$. In case of the Dirichlet boundary condition (value of $BCType[0]$ is 'D'), it contains values of the function $G(ax, y_j, z_k)$, $j=0, \dots, ny$, $k=0, \dots, nz$. In case of the Neumann boundary condition (value of $BCType[0]$ is 'N'), it contains the values of the function $g(ax, y_j, z_k)$, $j=0, \dots, ny$, $k=0, \dots, nz$. The values are packed in the array so that the value corresponding to indices (j, k) is placed in $bd\_ax[j+k*(ny+1)]$.

bd_bx

double* for $d\_Helmholtz\_2D/d\_Helmholtz\_3D$,
float* for $s\_Helmholtz\_2D/s\_Helmholtz\_3D$.

Contains values of the boundary condition on the rightmost boundary of the domain along x -axis.

For $?\_Helmholtz\_2D$, the size of the array is $ny+1$. In case of the Dirichlet boundary condition (value of $BCType[1]$ is 'D'), it contains values of the function $G(bx, y_j)$, $j=0, \dots, ny$. In case of the Neumann boundary condition (value of $BCType[1]$ is 'N'), it contains values of the function $g(bx, y_j)$, $j=0, \dots, ny$. The value corresponding to the index j is placed in $bd\_bx[j]$.

For $?\_Helmholtz\_3D$, the size of the array is $(ny+1)*(nz+1)$. In case of the Dirichlet boundary condition (value of $BCType[1]$ is 'D'), it contains values of the function $G(bx, y_j, z_k)$, $j=0, \dots, ny$, $k=0, \dots, nz$. In case of the Neumann boundary condition (value of $BCType[1]$ is 'N'), it contains the values of the function $g(bx, y_j, z_k)$, $j=0, \dots, ny$, $k=0, \dots, nz$. The values are packed in the array so that the value corresponding to indices (j, k) is placed in $bd\_bx[j+k*(ny+1)]$.

bd_ay

double* for $d\_Helmholtz\_2D/d\_Helmholtz\_3D$,
float* for $s\_Helmholtz\_2D/s\_Helmholtz\_3D$.

Contains values of the boundary condition on the leftmost boundary of the domain along y -axis.

For `?_Helmholtz_2D`, the size of the array is $nx+1$. In case of the Dirichlet boundary condition (value of `BCtype[2]` is 'D'), it contains values of the function $g(x_i, ay)$, $i=0, \dots, nx$. In case of the Neumann boundary condition (value of `BCtype[2]` is 'N'), it contains values of the function $g(x_i, ay)$, $i=0, \dots, nx$. The value corresponding to the index i is placed in `bd_ay[i]`.

For `?_Helmholtz_3D`, the size of the array is $(nx+1)*(nz+1)$. In case of the Dirichlet boundary condition (value of `BCtype[2]` is 'D'), it contains values of the function $G(x_i, ay, z_k)$, $i=0, \dots, nx$, $k=0, \dots, nz$. In case of the Neumann boundary condition (value of `BCtype[2]` is 'N'), it contains the values of the function $g(x_i, ay, z_k)$, $i=0, \dots, nx$, $k=0, \dots, nz$. The values are packed in the array so that the value corresponding to indices (i, k) is placed in `bd_ay[i+k*(nx+1)]`.

`bd_by`

`double* for d_Helmholtz_2D/d_Helmholtz_3D,`
`float* for s_Helmholtz_2D/s_Helmholtz_3D.`

Contains values of the boundary condition on the rightmost boundary of the domain along y -axis.

For `?_Helmholtz_2D`, the size of the array is $nx+1$. In case of the Dirichlet boundary condition (value of `BCtype[3]` is 'D'), it contains values of the function $g(x_i, by)$, $i=0, \dots, nx$. In case of the Neumann boundary condition (value of `BCtype[3]` is 'N'), it contains values of the function $g(x_i, by)$, $i=0, \dots, nx$. The value corresponding to the index i is placed in `bd_by[i]`.

For `?_Helmholtz_3D`, the size of the array is $(nx+1)*(nz+1)$. In case of the Dirichlet boundary condition (value of `BCtype[3]` is 'D'), it contains values of the function $G(x_i, by, z_k)$, $i=0, \dots, nx$, $k=0, \dots, nz$. In case of the Neumann boundary condition (value of `BCtype[3]` is 'N'), it contains the values of the function $g(x_i, by, z_k)$, $i=0, \dots, nx$, $k=0, \dots, nz$. The values are packed in the array so that the value corresponding to indices (i, k) is placed in `bd_by[i+k*(nx+1)]`.

`bd_az`

`double* for d_Helmholtz_2D/d_Helmholtz_3D,`

`float* for s_Helmholtz_2D/s_Helmholtz_3D.`
 This parameter is needed only for `?_Helmholtz_3D`.
 Contains values of the boundary condition on the leftmost boundary of the domain along z -axis.
 The size of the array is $(nx+1)*(ny+1)$. In case of the Dirichlet boundary condition (value of `BCtype[4]` is 'D'), it contains values of the function $G(x_i, y_j, az)$, $i=0, \dots, nx$, $j=0, \dots, ny$. In case of the Neumann boundary condition (value of `BCtype[4]` is 'N'), it contains the values of the function $g(x_i, y_j, az)$, $i=0, \dots, nx$, $j=0, \dots, ny$. The values are packed in the array so that the value corresponding to indices (i, j) is placed in `bd_az[i+j*(nx+1)]`.

`bd_bz` `double* for d_Helmholtz_2D/d_Helmholtz_3D,`
`float* for s_Helmholtz_2D/s_Helmholtz_3D.`
 This parameter is needed only for `?_Helmholtz_3D`.
 Contains values of the boundary condition on the rightmost boundary of the domain along z -axis.
 The size of the array is $(nx+1)*(ny+1)$. In case of the Dirichlet boundary condition (value of `BCtype[5]` is 'D'), it contains values of the function $G(x_i, y_j, bz)$, $i=0, \dots, nx$, $j=0, \dots, ny$. In case of the Neumann boundary condition (value of `BCtype[5]` is 'N'), it contains the values of the function $g(x_i, y_j, bz)$, $i=0, \dots, nx$, $j=0, \dots, ny$. The values are packed in the array so that the value corresponding to indices (i, j) is placed in `bd_bz[i+j*(nx+1)]`.



NOTE. To avoid wrong computation results, do not change arrays `bd_ax`, `bd_bx`, `bd_ay`, `bd_by`, `bd_az`, `bd_bz` between a call to the `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D` routine and a subsequent call to the appropriate `?_Helmholtz_2D/?_Helmholtz_3D` routine.

Output Parameters

`f` On output, contains the approximate solution to the problem packed the same way as the right-hand side of the problem was packed on input.

`xhandle, yhandle` Data structures used by the Intel MKL FFT interface.

<i>ipar</i>	Contains integer data to be used by Fast Helmholtz Solver. Modified on output as explained in Common Parameters .
<i>dpar</i>	Contains double-precision data to be used by Fast Helmholtz Solver. Modified on output as explained in Common Parameters
<i>spar</i>	Contains single-precision data to be used by Fast Helmholtz Solver. Modified on output as explained in Common Parameters .
<i>stat</i>	<code>int*</code> . Routine completion status, which is also written to <i>ipar</i> [0]. The status should be 0 to proceed to other PL routines.

Description

The routines compute the approximate solution of Helmholtz problem defined in the previous calls to the corresponding initialization and commit routines. The solution is computed according to formulas given in the [Poisson Library Implemented](#) section. The *f* parameter, which initially holds the packed vector of the right-hand side of the problem, is replaced by the computed solution packed in the same way. Values of *ax*, *bx*, *ay*, *by*, *az*, *bz*, *nx*, *ny*, *nz*, and *BCtype* are passed to each of the routines with the *ipar* array and defined in the previous call to the appropriate `?_init_Helmholtz_2D/?_init_Helmholtz_3D` routine.

Return Values

<i>stat</i> = 1	The routine completed without errors and produced some warnings.
<i>stat</i> = 0	The routine successfully completed the task.
<i>stat</i> = -2	The routine stopped because division by zero occurred. It usually happens if the data in the <i>dpar</i> or <i>spar</i> array was altered by mistake.
<i>stat</i> = -3	The routine stopped because the memory was insufficient to complete the computations.
<i>stat</i> = -100	The routine stopped because an error in the user's data was found or the data in the <i>dpar</i> , <i>spar</i> or <i>ipar</i> array was altered by mistake.
<i>stat</i> = -1000	The routine stopped because of the Intel MKL FFT or TT interface error.

`stat= -10000`

The routine stopped because the initialization failed to complete or the parameter `ipar[0]` was altered by mistake.

`stat= -99999`

The routine failed to complete the task because of a fatal error.

free_Helmholtz_2D/free_Helmholtz_3D

Cleans the memory allocated for the data structures used by the FFT interface.

Syntax

```
void free_Helmholtz_2D(DFTI_DESCRIPTOR* xhandle, int* ipar, int* stat);
void free_Helmholtz_3D (DFTI_DESCRIPTOR* xhandle, DFTI_DESCRIPTOR* yhandle,
int* ipar, int* stat);
```

Input Parameters

<code>xhandle, yhandle</code>	DESCRIPTOR_HANDLE*. Data structures used by the Intel MKL FFT interface (for details, refer to section “ FFT Functions ” in chapter “Fast Fourier Transforms”). The structure <code>yhandle</code> is used only by <code>free_Helmholtz_3D</code> .
<code>ipar</code>	int array of size 128. Contains integer data to be used by Fast Helmholtz Solver (for details, refer to Common Parameters).

Output Parameters

<code>xhandle, yhandle</code>	Data structures used by the Intel MKL FFT interface. Memory allocated for the structures is released on output.
<code>ipar</code>	Contains integer data to be used by Fast Helmholtz Solver. Status of the routine call is written to <code>ipar[0]</code> .
<code>stat</code>	int*. Routine completion status, which is also written to <code>ipar[0]</code> .

Description

The routine cleans the memory used by the *xhandle* and *yhandle* structures, needed for calling the Intel MKL FFT functions. To release memory allocated for other parameters, include cleaning of the memory in your code.

Return Values

<code>stat= 0</code>	The routine successfully completed the task.
<code>stat= -1000</code>	The routine stopped because of an Intel MKL FFT or TT interface error.
<code>stat= -99999</code>	The routine failed to complete the task because of a fatal error.

PL Routines for the Spherical Solver

The section gives detailed description of spherical PL routines, their syntax, parameters and values they return. All flavors of the same routine, namely, double-precision and single-precision, periodic (having names ending in "p") and non-periodic (having names ending in "np"), are described together.

These PL routines also call the Intel MKL FFT interface (described in section "[FFT Functions](#)" in chapter "Fast Fourier Transforms"), which enhances performance of the routines.

?_init_sph_p/?_init_sph_np

Initializes basic data structures of the Fast periodic and non-periodic Helmholtz Solver on a sphere.

Syntax

```
void d_init_sph_p(double* ap, double* at, double* bp, double* bt, int* np,
int* nt, double* q, int* ipar, double* dpar, int* stat);

void s_init_sph_np(float* ap, float* at, float* bp, float* bt, int* np, int*
nt, float* q, int* ipar, float* spar, int* stat);

void d_init_sph_np(double* ap, double* at, double* bp, double* bt, int* np,
int* nt, double* q, int* ipar, double* dpar, int* stat);

void s_init_sph_np(float* ap, float* at, float* bp, float* bt, int* np, int*
nt, float* q, int* ipar, float* spar, int* stat);
```

Input Parameters

<i>ap</i>	double* for <code>d_init_sph_p/d_init_sph_np</code> , float* for <code>s_init_sph_p/s_init_sph_np</code> . The coordinate (angle) of the leftmost boundary of the domain along ϕ -axis.
<i>bp</i>	double* for <code>d_init_sph_p/d_init_sph_np</code> , float* for <code>s_init_sph_p/s_init_sph_np</code> . The coordinate (angle) of the rightmost boundary of the domain along ϕ -axis.
<i>at</i>	double* for <code>d_init_sph_p/d_init_sph_np</code> , float* for <code>s_init_sph_p/s_init_sph_np</code> . The coordinate (angle) of the leftmost boundary of the domain along θ -axis.
<i>bt</i>	double* for <code>d_init_sph_p/d_init_sph_np</code> , float* for <code>s_init_sph_p/s_init_sph_np</code> . The coordinate (angle) of the rightmost boundary of the domain along θ -axis.
<i>np</i>	int*. The number of mesh intervals along ϕ -axis. Must be even in the periodic case.
<i>nt</i>	int*. The number of mesh intervals along θ -axis.
<i>q</i>	double* for <code>d_init_sph_p/d_init_sph_np</code> , float* for <code>s_init_sph_p/s_init_sph_np</code> . The constant Helmholtz coefficient. Note that to solve Poisson problem, you should set the value of <i>q</i> to 0.

Output Parameters

<i>ipar</i>	int array of size 128. Contains integer data to be used by Fast Helmholtz Solver on a sphere (for details, refer to Common Parameters).
<i>dpar</i>	double array of size $5*np/2+nt+10$. Contains double-precision data to be used by Fast Helmholtz Solver on a sphere (for details, refer to Common Parameters).

<i>spar</i>	float array of size $5 \cdot np/2 + nt + 10$. Contains single-precision data to be used by Fast Helmholtz Solver on a sphere (for details, refer to Common Parameters).
<i>stat</i>	int*. Routine completion status, which is also written to <i>ipar[0]</i> . The status should be 0 to proceed to other PL routines.

Description

The routines `?_init_sph_p/?_init_sph_np` are called to initialize basic data structures for Poisson Library computations of the appropriate precision. All routines invoked after a call to a `?_init_Helmholtz_2D/?_init_Helmholtz_3D` routine use values of the *ipar*, *dpar* and *spar* array parameters returned by the routine. Detailed description of the array parameters can be found in [Common Parameters](#).



WARNING. Data structures initialized and created by periodic/non-periodic flavors of the routine cannot be used by non-periodic/periodic flavors of any PL routines for Helmholtz Solver on a sphere, respectively.

You can skip calling this routine in your code. However, see [Caveat on Parameter Modifications](#) before doing so.

Return Values

<i>stat</i> = 0	The routine successfully completed the task. In general, to proceed with computations, the routine should complete with this <i>stat</i> value.
<i>stat</i> = -99999	The routine failed to complete the task because of fatal error.

?_commit_sph_p/?_commit_sph_np

Checks consistency and correctness of user's data as well as initializes certain data structures required to solve periodic/non-periodic Helmholtz problem on a sphere.

Syntax

```
void d_commit_sph_p(double* f, DFTI_DESCRIPTOR* handle_s, DFTI_DESCRIPTOR*
handle_c, int* ipar, double* dpar, int* stat);

void s_commit_sph_p(float* f, DFTI_DESCRIPTOR* handle_s, DFTI_DESCRIPTOR*
handle_c, int* ipar, float* spar, int* stat);

void d_commit_sph_np(double* f, DFTI_DESCRIPTOR* handle, int* ipar, double*
dpar, int* stat);

void s_commit_sph_np(float* f, DFTI_DESCRIPTOR* handle, int* ipar, float*
spar, int* stat);
```

Input Parameters

<i>f</i>	double* for d_commit_sph_p/d_commit_sph_np, float* for s_commit_sph_p/s_commit_sph_np. Contains the right-hand side of the problem packed in a single vector. The size of the vector is $(n_p+1)*(n_t+1)$ and value of the right-hand side in the mesh point (i, j) is stored in $f[i+j*(n_p+1)]$. Note that the array <i>f</i> may be altered by the routine. Please save this vector in another memory location if you want to preserve it.
<i>ipar</i>	int array of size 128. Contains integer data to be used by Fast Helmholtz Solver on a sphere (for details, refer to Common Parameters).
<i>dpar</i>	double array of size $5*n_p/2+n_t+10$. Contains double-precision data to be used by Fast Helmholtz Solver on a sphere (for details, refer to Common Parameters).
<i>spar</i>	float array of size $5*n_p/2+n_t+10$. Contains single-precision data to be used by Fast Helmholtz Solver on a sphere (for details, refer to Common Parameters).

Output Parameters

<i>f</i>	Vector of the right-hand side of the problem. Possibly, altered on output.
<i>ipar</i>	Contains integer data to be used by Fast Helmholtz Solver on a sphere. Modified on output as explained in Common Parameters .
<i>dpar</i>	Contains double-precision data to be used by Fast Helmholtz Solver on a sphere. Modified on output as explained in Common Parameters .
<i>spar</i>	Contains single-precision data to be used by Fast Helmholtz Solver on a sphere. Modified on output as explained in Common Parameters .
<i>handle_s</i> , <i>handle_c</i> , <i>handle</i>	DESCRIPTOR_HANDLE*. Data structures used by the Intel MKL FFT interface (for details, refer to section “ FFT Functions ” in chapter “Fast Fourier Transforms”). <i>handle_s</i> and <i>handle_c</i> are used only in <code>?_commit_sph_p</code> and <i>handle</i> is used only in <code>?_commit_sph_np</code> .
<i>stat</i>	int*. Routine completion status, which is also written to <i>ipar</i> [0]. The status should be 0 to proceed to other PL routines.

Description

The routines `?_commit_sph_p/?_commit_sph_np` check consistency and correctness of the parameters to be passed to the solver routines `?_sph_p/?_sph_np`, respectively. They also initialize certain data structures. The routine `?_commit_sph_p` initializes structures *handle_s* and *handle_c*, and `?_commit_sph_np` initializes *handle*. The routines also initialize arrays *ipar* and *dpar/spar*, depending upon the routine precision. Refer to [Common Parameters](#) to find out which particular array elements the `?_commit_sph_p/?_commit_sph_np` routines initialize and what values are written there.

The routines perform only a basic check for correctness and consistency. If you are going to modify parameters of PL routines, see the [Caveat on Parameter Modifications](#) section. Unlike `?_init_sph_p/?_init_sph_np`, the routines `?_commit_sph_p/?_commit_sph_np` are mandatory, and you cannot skip calling them in your code. Values of *np* and *nt* are passed to each of the routines with the *ipar* array and defined in a previous call to the appropriate `?_init_sph_p/?_init_sph_np` routine.

Return Values

<code>stat= 1</code>	The routine completed without errors and produced some warnings.
<code>stat= 0</code>	The routine successfully completed the task.
<code>stat= -100</code>	The routine stopped because an error in the user's data was found or the data in the <code>dpar</code> , <code>spar</code> or <code>ipar</code> array was altered by mistake.
<code>stat= -1000</code>	The routine stopped because of an Intel MKL FFT or TT interface error.
<code>stat= -10000</code>	The routine stopped because the initialization failed to complete or the parameter <code>ipar[0]</code> was altered by mistake.
<code>stat= -99999</code>	The routine failed to complete the task because of a fatal error.

?_sph_p/?_sph_np

Computes the solution of a spherical Helmholtz problem specified by the parameters.

Syntax

```
void d_sph_p(double* f, DFTI_DESCRIPTOR* handle_s, DFTI_DESCRIPTOR* handle_c,
int* ipar, double* dpar, int* stat);

void s_sph_p(float* f, DFTI_DESCRIPTOR* handle_s, DFTI_DESCRIPTOR* handle_c,
int* ipar, float* spar, int* stat);

void d_sph_np(double* f, DFTI_DESCRIPTOR* handle, int* ipar, double* dpar,
int* stat);

void s_sph_np(float* f, DFTI_DESCRIPTOR* handle, int* ipar, float* spar, int*
stat);
```

Input Parameters

`f` double* for `d_sph_p/d_sph_np`,
 float* for `s_sph_p/s_sph_np`.

Contains the right-hand side of the problem packed in a single vector and modified by the appropriate `?_commit_sph_p/?_commit_sph_np` routine. Note that an attempt to substitute the original right-hand side vector at this point will result in a wrong solution. The size of the vector is $(np+1)*(nt+1)$ and value of the right-hand side in the mesh point (i, j) is stored in $f[i+j*(np+1)]$.

<i>handle_s, handle_c, handle</i>	DESCRIPTOR_HANDLE*. Data structures used by Intel MKL FFT interface (for details, refer to section “ FFT Functions ” in chapter “Fast Fourier Transforms”). <i>handle_s</i> and <i>handle_c</i> are used only in <code>?_sph_p</code> and <i>handle</i> is used only in <code>?_sph_np</code> .
<i>ipar</i>	int array of size 128. Contains integer data to be used by Fast Helmholtz Solver on a sphere (for details, refer to Common Parameters).
<i>dpar</i>	double array of size $5*np/2+nt+10$. Contains double-precision data to be used by Fast Helmholtz Solver on a sphere (for details, refer to Common Parameters).
<i>spar</i>	float array of size $5*np/2+nt+10$. Contains single-precision data to be used by Fast Helmholtz Solver on a sphere (for details, refer to Common Parameters).

Output Parameters

<i>f</i>	On output, contains the approximate solution to the problem packed the same way as the right-hand side of the problem was packed on input.
<i>handle_s, handle_c, handle</i>	Data structures used by the Intel MKL FFT interface.
<i>ipar</i>	Contains integer data to be used by Fast Helmholtz Solver on a sphere. Modified on output as explained in Common Parameters .
<i>dpar</i>	Contains double-precision data to be used by Fast Helmholtz Solver on a sphere. Modified on output as explained in Common Parameters .

<i>spar</i>	Contains single-precision data to be used by Fast Helmholtz Solver on a sphere. Modified on output as explained in Common Parameters .
<i>stat</i>	<code>int*</code> . Routine completion status, which is also written to <i>ipar[0]</i> . The status should be 0 to proceed to other PL routines.

Description

The routines compute the approximate solution on a sphere of the Helmholtz problem defined in the previous calls to the corresponding initialization and commit routines. The solution is computed according to formulas given in the [Poisson Library Implemented](#) section. The *f* parameter, which initially holds the packed vector of the right-hand side of the problem, is replaced by the computed solution packed in the same way. Values of *np* and *nt* are passed to each of the routines with the *ipar* array and defined in the previous call to the appropriate `?_init_sph_p/?_init_sph_np` routine.

Return Values

<i>stat</i> = 1	The routine completed without errors and produced some warnings.
<i>stat</i> = 0	The routine successfully completed the task.
<i>stat</i> = -2	The routine stopped because division by zero occurred. It usually happens if the data in the <i>dpar</i> or <i>spar</i> array was altered by mistake.
<i>stat</i> = -3	The routine stopped because the memory was insufficient to complete the computations.
<i>stat</i> = -100	The routine stopped because an error in the user's data was found or the data in the <i>dpar</i> , <i>spar</i> or <i>ipar</i> array was altered by mistake.
<i>stat</i> = -1000	The routine stopped because of an Intel MKL FFT or TT interface error.
<i>stat</i> = -10000	The routine stopped because the initialization failed to complete or the parameter <i>ipar[0]</i> was altered by mistake.
<i>stat</i> = -99999	The routine failed to complete the task because of a fatal error.

free_sph_p/free_sph_np

Cleans the memory allocated for the data structures used by the FFT interface.

Syntax

```
void free_sph_p(DFTI_DESCRIPTOR* handle_s, DFTI_DESCRIPTOR* handle_c, int*
ipar, int* stat);

void free_sph_np(DFTI_DESCRIPTOR* handle, int* ipar, int* stat);
```

Input Parameters

<i>handle_s, handle_c,</i> <i>handle</i>	DESCRIPTOR_HANDLE*. Data structures used by the Intel MKL FFT interface (for details, refer to section “ FFT Functions ” in chapter “Fast Fourier Transforms”). The structures <i>handle_s</i> and <i>handle_c</i> are used only in <i>free_sph_p</i> , and <i>handle</i> is used only in <i>free_sph_np</i> .
<i>ipar</i>	int array of size 128. Contains integer data to be used by Fast Helmholtz Solver on a sphere (for details, refer to Common Parameters).

Output Parameters

<i>handle_s, handle_c,</i> <i>handle</i>	Data structures used by the Intel MKL FFT interface. Memory allocated for the structures is released on output.
<i>ipar</i>	Contains integer data to be used by Fast Helmholtz Solver on a sphere. Status of the routine call is written to <i>ipar[0]</i> .
<i>stat</i>	int*. Routine completion status, which is also written to <i>ipar[0]</i> .

Description

The routine cleans the memory used by the *handle_s*, *handle_c* or *handle* structures, needed for calling the Intel MKL FFT functions. To release memory allocated for other parameters, include cleaning of the memory in your code.

Return Values

<i>stat</i> = 0	The routine successfully completed the task.

<code>stat= -1000</code>	The routine stopped because of the Intel MKL FFT or TT interface error.
<code>stat= -99999</code>	The routine failed to complete the task because of a fatal error.

Common Parameters

This section provides description of array parameters *ipar*, *dpar* and *spar*, which hold PL routine options in both Cartesian and spherical cases.



NOTE. Initial values are assigned to the array parameters by the appropriate `?_init_Helmholtz_2D/?_init_Helmholtz_3D/?_init_sph_p/?_init_sph_np` and `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D/?_commit_sph_p/?_commit_sph_np` routines.

<i>ipar</i>	int array of size 128, holds integer data needed for Fast Helmholtz Solver (both for Cartesian and spherical coordinate systems). Its elements are described in Table 13-6 :

Table 13-6 Elements of the ipar Array

Index	Description
0	Contains status value of the last called PL routine. In general, it should be 0 to proceed with Fast Helmholtz Solver. The element has no predefined values. This element can also be used to inform the <code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D/?_commit_sph_p/?_commit_sph_np</code> routines of how the Commit step of the computation should be carried out (see Figure 13-3). A non-zero value of <code>ipar[0]</code> with decimal representation

$$\overline{abc} = 100a + 10b + c$$

where each of *a*, *b*, and *c* is equal to 0 or 9, indicates that some parts of the Commit step should be omitted.

- If *c*=9, the routine omits checking of parameters and initialization of the data structures.

Index	Description
	• If $b=9$, then in the Cartesian case, the routine omits the adjustment of the right-hand side vector f to the Neumann boundary condition (multiplication of boundary values by 0.5 as well as incorporation of the boundary function g) and/or the Dirichlet boundary condition (setting boundary values to 0 as well as incorporation of the boundary function g). In this case, the routine also omits the adjustment of the right-hand side vector f to the particular boundary functions. For the Helmholtz solver on a sphere, the routine omits computation of the spherical weights for the $dpar/spar$ array. • If $a=9$, then the routine omits the normalization of the right-hand side vector f. In the 2D Cartesian case, it is the multiplication by h_y^2, where h_y is the mesh size in the y direction (for details, see Poisson Library Implemented). In the 3D (Cartesian) case, it is the multiplication by h_z^2, where h_z is the mesh size in the z direction. For the Helmholtz solver on a sphere, it is the multiplication by h_θ^2, where h_θ is the mesh size in the θ direction (for details, see Poisson Library Implemented).

Using `ipar[0]` you can adjust the routine to your needs and gain efficiency in solving multiple Helmholtz problems that differ only in the right-hand side. You must be cautious using this opportunity, because misunderstanding of the commit process may cause wrong results or program failure (see also [Caveat on Parameter Modifications](#)).

- 1 Contains error messaging options:
 - `ipar[1]=-1` indicates that all error messages will be printed to the file `MKL_Poisson_Library_log.txt` in the folder from which the routine is called. If the file does not exist, the routine tries to create it. If the attempt fails, the routine prints information that the file cannot be created to the standard output device.
 - `ipar[1]=0` indicates that no error messages will be printed.
 - `ipar[1]=1` is the default value. It indicates that all error messages will be printed to the preconnected default output device (usually, screen).

In case of errors, the `stat` parameter will acquire a non-zero value regardless of the `ipar[1]` setting.

- 2 Contains warning messaging options:

Index	Description
	• <code>ipar[2]=-1</code> indicates that all warning messages will be printed to the file <code>MKL_Poisson_Library_log.txt</code> in the directory from which the routine is called. If the file does not exist, the routine tries to create it. If the attempt fails, the routine prints information that the file cannot be created to the standard output device. • <code>ipar[2]=0</code> indicates that no warning messages will be printed. • <code>ipar[2]=1</code> is the default value. It indicates that all warning messages will be printed to the preconnected default output device (usually, screen). In case of warnings, the <code>stat</code> parameter will acquire a non-zero value regardless of the <code>ipar[2]</code> setting.
3	Contains the number of the combination of boundary conditions. In the Cartesian case, it corresponds to the value that the <code>BCtype</code> parameter holds: • In the 2D case, 0 corresponds to 'DDDD' 1 corresponds to 'DDDN' ... 15 corresponds to 'NNNN' • In the 3D case, 0 corresponds to 'DDDDDD' 1 corresponds to 'DDDDDN' ... 63 corresponds to 'NNNNNNN'. The Helmholtz solver on a sphere uses this parameter only in a periodic case. The <code>bp</code> and <code>bt</code> parameters of the <code>?_init_sph_p/?_init_sph_np</code> routine, which initializes <code>ipar[3]</code>, determine its value: • 0 corresponds to the problem without poles. • 1 corresponds to the problem on the entire sphere.
Parameters 4 through 9 are used only in Cartesian case.	
4	Takes the value of 1 if <code>BCtype[0]='N'</code> , 0 if <code>BCtype[0]='D'</code> , and -1 otherwise.
5	Takes the value of 1 if <code>BCtype[1]='N'</code> , 0 if <code>BCtype[1]='D'</code> , and -1 otherwise.

Index	Description
6	Takes the value of 1 if <i>BCTYPE</i> [2]='N', 0 if <i>BCTYPE</i> [2]='D', and -1 otherwise.
7	Takes the value of 1 if <i>BCTYPE</i> [3]='N', 0 if <i>BCTYPE</i> [3]='D', and -1 otherwise.
8	Takes the value of 1 if <i>BCTYPE</i> [4]='N', 0 if <i>BCTYPE</i> [4]='D', and -1 otherwise. This parameter is used only in the 3D case.
9	Takes the value of 1 if <i>BCTYPE</i> [5]='N', 0 if <i>BCTYPE</i> [5]='D', and -1 otherwise. This parameter is used only in the 3D case.
10	Takes the value of <i>nx</i>, that is, the number of intervals along <i>x</i>-axis, in the Cartesian case. <i>np</i>, that is, the number of intervals along ϕ-axis, in the spherical case.
11	Takes the value of <i>ny</i>, that is, the number of intervals along <i>y</i>-axis, in the Cartesian case <i>nt</i>, that is, the number of intervals along θ-axis, in the spherical case.
12	Takes the value of <i>nz</i> , the number of intervals along <i>z</i> -axis. This parameter is used only in the 3D case (Cartesian).
13	Takes the value of 6, which specifies the internal partitioning of the <i>dpar/spar</i> array.
14	Takes the value of <i>ipar</i> [13]+ <i>ipar</i> [10]+1, which specifies the internal partitioning of the <i>dpar/spar</i> array.

Subsequent values of *ipar* depend upon the dimension of the problem or upon whether the solver on a sphere is periodic.

	2D case	3D case	Periodic case	Non-periodic case
15	Unused	Takes the value of <i>ipar</i> [14]+1, which specifies the internal partitioning of the <i>dpar/spar</i> array.		
16	Unused	Takes the value of <i>ipar</i> [14]+ <i>ipar</i> [11]+1, which specifies the internal partitioning of the <i>dpar/spar</i> array.		

Index	Description			
17	Takes the value of $ipar[14]+1$, which specifies the internal partitioning of the <i>dpar/spar</i> array.	Takes the value of $ipar[16]+1$, which specifies the internal partitioning of the <i>dpar/spar</i> array.		
18	Takes the value of $ipar[14]+3*ipar[10]/2+1$, which specifies the internal partitioning of the <i>dpar/spar</i> array.	Takes the value of $ipar[16]+3*ipar[10]/2+1$, which specifies the internal partitioning of the <i>dpar/spar</i> array.	Takes the value of $ipar[16]+3*ipar[10]/4+1$, which specifies the internal partitioning of the <i>dpar/spar</i> array.	Takes the value of $ipar[16]+3*ipar[10]/2+1$, which specifies the internal partitioning of the <i>dpar/spar</i> array.
19	Unused	Takes the value of $ipar[18]+1$, which specifies the internal partitioning of the <i>dpar/spar</i> array.		Unused
20	Unused	Takes the value of $ipar[18]+3*ipar[11]/2+1$, which specifies the internal partitioning of the <i>dpar/spar</i> array.	Takes the value of $ipar[18]+3*ipar[10]/4+1$, which specifies the internal partitioning of the <i>dpar/spar</i> array.	Unused

Subsequent values of *ipar* are assigned regardless.

21	Contains message style options: • $ipar[21]=0$ indicates that PL routines print all error and warning messages in Fortran-style notations. • $ipar[21]=1$ (default) indicates that PL routines print the messages in C-style notations.
22	Contains the number of threads to be used for computations in a multithreaded environment. The default value is 1.

Index	Description
23 through 39	Unused in the current implementation of the Poisson Library.
40 through 59	Contain the first twenty elements of the <i>ipar</i> array of the first Trigonometric Transform that the Solver uses. (For details, see Common Parameters in the “Trigonometric Transform Routines” chapter.)
60 through 79	Contain the first twenty elements of the <i>ipar</i> array of the second Trigonometric Transform that the 3D and periodic solvers use. (For details, see Common Parameters in the “Trigonometric Transform Routines” chapter.)



NOTE. You may declare the *ipar* array in your code as `int ipar[80]`. However, for compatibility with later versions of Intel MKL Poisson Library, which may require more *ipar* values, it is highly recommended to declare *ipar* as `int ipar[128]`.

Arrays *dpar* and *spar* are the same except in the data precision:

<i>dpar</i>	Holds data needed for double-precision Fast Helmholtz Solver computations. - For the Cartesian solver, double array of size $5 \cdot n_x/2 + 7$ in the 2D case or $5 \cdot (n_x + n_y)/2 + 9$ in the 3D case; initialized in the <code>d_init_Helmholtz_2D/d_init_Helmholtz_3D</code> and <code>d_commit_Helmholtz_2D/d_commit_Helmholtz_3D</code> routines. - For the spherical solver, double array of size $5 \cdot n_p/2 + n_t + 10$; initialized in the <code>d_init_sph_p/d_init_sph_np</code> and <code>d_commit_sph_p/d_commit_sph_np</code> routines.
<i>spar</i>	Holds data needed for single-precision Fast Helmholtz Solver computations. - For the Cartesian solver, float array of size $5 \cdot n_x/2 + 7$ in the 2D case or $5 \cdot (n_x + n_y)/2 + 9$ in the 3D case; initialized in the <code>s_init_Helmholtz_2D/s_init_Helmholtz_3D</code> and <code>s_commit_Helmholtz_2D/s_commit_Helmholtz_3D</code> routines. - For the spherical solver, float array of size $5 \cdot n_p/2 + n_t + 10$; initialized in the <code>s_init_sph_p/s_init_sph_np</code> and <code>s_commit_sph_p/s_commit_sph_np</code> routines.

As *dpar* and *spar* have similar elements in respective positions, the elements are described together in Table 13-7:

Table 13-7 Elements of the *dpar* and *spar* Arrays

Index	Description
0	In the Cartesian case, contains the length of the interval along x-axis right after a call to the <code>?_init_Helmholtz_2D/?_init_Helmholtz_3D</code> routine or the mesh size h_x in the x direction (for details, see Poisson Library Implemented) after a call to the <code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D</code> routine. In the spherical case, contains the length of the interval along ϕ-axis right after a call to the <code>?_init_sph_p/?_init_sph_np</code> routine or the mesh size h_ϕ in the ϕ direction (for details, see Poisson Library Implemented) after a call to the <code>?_commit_sph_p/?_commit_sph_np</code> routine.
1	In the Cartesian case, contains the length of the interval along y-axis right after a call to the <code>?_init_Helmholtz_2D/?_init_Helmholtz_3D</code> routine or the mesh size h_y in the y direction (for details, see Poisson Library Implemented) after a call to the <code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D</code> routine. In the spherical case, contains the length of the interval along θ-axis right after a call to the <code>?_init_sph_p/?_init_sph_np</code> routine or the mesh size h_θ in the θ direction (for details, see Poisson Library Implemented) after a call to the <code>?_commit_sph_p/?_commit_sph_np</code> routine.
2	In the Cartesian case, contains the length of the interval along z-axis right after a call to the <code>?_init_Helmholtz_2D/?_init_Helmholtz_3D</code> routine or the mesh size h_z in the z direction (for details, see Poisson Library Implemented) after a call to the <code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D</code> routine. In the Cartesian solver, this parameter is used only in the 3D case. In the spherical solver, contains the coordinate of the leftmost boundary along θ-axis after a call to the <code>?_init_sph_p/?_init_sph_np</code> routine.
3	Contains the value of the coefficient q after a call to the <code>?_init_Helmholtz_2D/?_init_Helmholtz_3D/?_init_sph_p/?_init_sph_np</code> routine.

Index	Description
4	Contains the tolerance parameter after a call to the <code>?_init_Helmholtz_2D/?_init_Helmholtz_3D/?_init_sph_p/?_init_sph_np</code> routine. - In the Cartesian case, this value is used only for the pure Neumann boundary conditions (<code>BType="NNNN"</code> in the 2D case; <code>BType="NNNNNN"</code> in the 3D case). This is a special case, because the right-hand side of the problem cannot be arbitrary if the coefficient q is zero. Poisson Library verifies that the classical solution exists (up to rounding errors) using this tolerance. In any case, Poisson Library computes the normal solution, that is, the solution that has the minimal Euclidean norm of residual. Nevertheless, the <code>?_Helmholtz_2D/?_Helmholtz_3D</code> routine informs the user that the solution may not exist in a classical sense (up to rounding errors). - In the spherical case, the value is used for the special case of a periodic problem on the entire sphere. This special case is similar to the above described Cartesian case with pure Neumann boundary conditions. So, here Poisson Library computes the normal solution as well. The parameter is also used to detect whether the problem is periodic up to rounding errors. The default value for this parameter is 1.0E-10 in case of double-precision computations or 1.0E-4 in case of single-precision computations. You can increase the value of the tolerance, for instance, to avoid the warnings that may appear.
<code>ipar[13]-1</code> through <code>ipar[14]-1</code>	In the Cartesian case, contain the spectrum of the 1D problem along x-axis after a call to the <code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D</code> routine. In the spherical case, contains the spectrum of the 1D problem along ϕ-axis after a call to the <code>?_commit_sph_p/?_commit_sph_np</code> routine.
<code>ipar[15]-1</code> through <code>ipar[16]-1</code>	In the Cartesian case, contain the spectrum of the 1D problem along y-axis after a call to the <code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D</code> routine. These elements are used only in the 3D case. In the spherical case, contains the spherical weights after a call to the <code>?_commit_sph_p/?_commit_sph_np</code> routine.

Index	Description
<i>ipar</i> [17]-1 through <i>ipar</i> [18]-1	Take the values of the (staggered) sine/cosine in the mesh points: • along <i>x</i>-axis after a call to the <code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D</code> routine for a Cartesian solver • along ϕ-axis after a call to the <code>?_commit_sph_p/?_commit_sph_np</code> routine for a spherical solver.
<i>ipar</i> [19]-1 through <i>ipar</i> [20]-1	Take the values of the (staggered) sine/cosine in the mesh points: • along <i>y</i>-axis after a call to the <code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D</code> routine for a Cartesian 3D solver • along ϕ-axis after a call to the <code>?_commit_sph_p</code> routine for a spherical periodic solver. These elements are not used in the 2D Cartesian case and in the non-periodic spherical case.



NOTE. You may define the array size depending upon the type of the problem to solve.

Caveat on Parameter Modifications

Flexibility of the PL interface enables you to skip calling the `?_init_Helmholtz_2D/?_init_Helmholtz_3D/?_init_sph_p/?_init_sph_np` routine and to initialize the basic data structures explicitly in your code. You may also need to modify contents of *ipar*, *dpar* and *spar* arrays after initialization. When doing so, provide correct and consistent data in the arrays. Mistakenly altered arrays cause errors or wrong computation. You can perform a basic check for correctness and consistency of parameters by calling the `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D` routine; however, this does not ensure the correct solution but only reduces the chance of errors or wrong results.



NOTE. To supply correct and consistent parameters to PL routines, you should have considerable experience in using the PL interface and good understanding of the solution process as well as elements that the *ipar*, *spar* and *dpar* arrays contain and dependencies between values of these elements.

However, in rare occurrences, even advanced users might fail in tuning parameters for the Fast Helmholtz Solver. In cases like these, refer for technical support at <http://www.intel.com/software/products/support/>.



WARNING. The only way that ensures a proper solution of a Helmholtz problem is to follow a typical sequence of invoking the routines and not change the default set of parameters. So, avoid modifications of *ipar*, *dpar* and *spar* arrays unless a strong need arises.

Implementation Details

Several aspects of the Intel MKL PL interface are platform-specific and language-specific. To promote portability across platforms and ease of use across different languages, users are provided with the PL language-specific header files to include in their code. Currently, the following header files are available:

- `mkl_poisson.h`, to be used together with `mkl_dfti.h`, for C programs.
- `mkl_poisson.f90`, to be used together with `mkl_dfti.f90`, for Fortran 90 programs.



NOTE. Use of the Intel MKL PL software without including one of the above header files is not supported.

The include files define function prototypes for appropriate languages.

C-specific Header File

The C-specific header file defines the following function prototypes for the Cartesian solver:

```
void d_init_Helmholtz_2D(double*, double*, double*, double*, int*, int*, char*, double*,
int*, double*, int*);

void d_commit_Helmholtz_2D(double*, double*, double*, double*, double*,
DFTI_DESCRIPTOR_HANDLE*, int*, double*, int*);

void s_Helmholtz_2D(double*, double*, double*, double*, double*, DFTI_DESCRIPTOR_HANDLE*,
int*, double*, int*);

void s_init_Helmholtz_2D(float*, float*, float*, float*, int*, int*, char*, float*, int*,
float*, int*);

void s_commit_Helmholtz_2D(float*, float*, float*, float*, float*, DFTI_DESCRIPTOR_HANDLE*,
int*, float*, int*);

void s_Helmholtz_2D(float*, float*, float*, float*, float*, DFTI_DESCRIPTOR_HANDLE*, int*,
float*, int*);

void free_Helmholtz_2D(DFTI_DESCRIPTOR_HANDLE*, int*, int*);

void d_init_Helmholtz_3D(double*, double*, double*, double*, double*, double*, int*, int*,
int*, char*, double*, int*, double*, int*);

void d_commit_Helmholtz_3D(double*, double*, double*, double*, double*, double*, double*,
DFTI_DESCRIPTOR_HANDLE*, DFTI_DESCRIPTOR_HANDLE*, int*, double*, int*);

void d_Helmholtz_3D(double*, double*, double*, double*, double*, double*, double*,
DFTI_DESCRIPTOR_HANDLE*, DFTI_DESCRIPTOR_HANDLE*, int*, double*, int*);

void s_init_Helmholtz_3D(float*, float*, float*, float*, float*, float*, int*, int*, int*,
char*, float*, int*, float*, int*);

void s_commit_Helmholtz_3D(float*, float*, float*, float*, float*, float*, float*,
DFTI_DESCRIPTOR_HANDLE*, DFTI_DESCRIPTOR_HANDLE*, int*, float*, int*);

void s_Helmholtz_3D(float*, float*, float*, float*, float*, float*, float*,
DFTI_DESCRIPTOR_HANDLE*, DFTI_DESCRIPTOR_HANDLE*, int*, float*, int*);

void free_Helmholtz_3D(DFTI_DESCRIPTOR_HANDLE*, DFTI_DESCRIPTOR_HANDLE*, int*, int*);
```

The C-specific header file defines the following function prototypes for the spherical solver:

```
void d_init_sph_p(double*, double*, double*, double*, int*, int*, double*, int*, double*,
int*);

void d_commit_sph_p(double*, DFTI_DESCRIPTOR_HANDLE*, DFTI_DESCRIPTOR_HANDLE*, int*, double*,
int*);

void d_sph_p(double*, DFTI_DESCRIPTOR_HANDLE*, DFTI_DESCRIPTOR_HANDLE*, int*, double*,
int*);
```

```
void s_init_sph_p(float*, float*, float*, float*, int*, int*, float*, int*, float*, int*);
void s_commit_sph_p(float*, DFTI_DESCRIPTOR_HANDLE*, DFTI_DESCRIPTOR_HANDLE*, int*, float*,
int*);
void s_sph_p(float*, DFTI_DESCRIPTOR_HANDLE*, DFTI_DESCRIPTOR_HANDLE*, int*, float*, int*);
void free_sph_p(DFTI_DESCRIPTOR_HANDLE*, DFTI_DESCRIPTOR_HANDLE*, int*, int*);
void d_init_sph_np(double*, double*, double*, double*, int*, int*, double*, int*, double*,
int*);
void d_commit_sph_np(double*, DFTI_DESCRIPTOR_HANDLE*, int*, double*, int*);
void d_sph_np(double*, DFTI_DESCRIPTOR_HANDLE*, int*, double*, int*);
void s_init_sph_np(float*, float*, float*, float*, int*, int*, float*, int*, float*, int*);
void s_commit_sph_np(float*, DFTI_DESCRIPTOR_HANDLE*, int*, float*, int*);
void s_sph_np(float*, DFTI_DESCRIPTOR_HANDLE*, int*, float*, int*);
void free_sph_np(DFTI_DESCRIPTOR_HANDLE*, int*, int*);
```

Fortran-Specific Header File

The Fortran90-specific header file defines the following function prototypes for the Cartesian solver:

```
SUBROUTINE D_INIT_HELMHOLTZ_2D (AX, BX, AY, BY, NX, NY, BCTYPE, Q, IPAR, DPAR, STAT)
    USE MKL_DFTI

    INTEGER NX, NY, STAT
    INTEGER IPAR(*)
    DOUBLE PRECISION AX, BX, AY, BY, Q
    DOUBLE PRECISION DPAR(*)
    CHARACTER(4) BCTYPE
END SUBROUTINE

SUBROUTINE D_COMMIT_HELMHOLTZ_2D (F, BD_AX, BD_BX, BD_AY, BD_BY, XHANDLE, IPAR, DPAR, STAT)
    USE MKL_DFTI
```

```

      INTEGER STAT
      INTEGER IPAR(*)
      DOUBLE PRECISION F(IPAR(11)+1,*)

      DOUBLE PRECISION DPAR(*)

      DOUBLE PRECISION BD_AX(*), BD_BX(*), BD_AY(*), BD_BY(*)
      TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE
END SUBROUTINE

SUBROUTINE D_HELMHOLTZ_2D (F, BD_AX, BD_BX, BD_AY, BD_BY, XHANDLE, IPAR, DPAR, STAT)
  USE MKL_DFTI

  INTEGER STAT
  INTEGER IPAR(*)
  DOUBLE PRECISION F(IPAR(11)+1,*)

  DOUBLE PRECISION DPAR(*)

  DOUBLE PRECISION BD_AX(*), BD_BX(*), BD_AY(*), BD_BY(*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE
END SUBROUTINE

SUBROUTINE S_INIT_HELMHOLTZ_2D (AX, BX, AY, BY, NX, NY, BCTYPE, Q, IPAR,
  SPAR,
  STAT)
  USE MKL_DFTI

```

```

    INTEGER NX, NY, STAT
    INTEGER IPAR(*)
    REAL AX, BX, AY, BY, Q
    REAL SPAR(*)
    CHARACTER(4) BCTYPE
END SUBROUTINE

SUBROUTINE S_COMMIT_HELMHOLTZ_2D (F, BD_AX, BD_BX, BD_AY, BD_BY, XHANDLE, IPAR, SPAR, STAT)
    USE MKL_DFTI

    INTEGER STAT
    INTEGER IPAR(*)
    REAL F(IPAR(11)+1,*)

    REAL SPAR(*)

    REAL BD_AX(*), BD_BX(*), BD_AY(*), BD_BY(*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE
END SUBROUTINE

SUBROUTINE S_HELMHOLTZ_2D (F, BD_AX, BD_BX, BD_AY, BD_BY, XHANDLE, IPAR, SPAR, STAT)
    USE MKL_DFTI

    INTEGER STAT
    INTEGER IPAR(*)
    REAL F(IPAR(11)+1,*)

    REAL SPAR(*)

```

```

      REAL BD_AX(*), BD_BX(*), BD_AY(*), BD_BY(*)
      TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE
END SUBROUTINE

SUBROUTINE FREE_HELMHOLTZ_2D (XHANDLE, IPAR, STAT)
  USE MKL_DFTI

  INTEGER STAT
  INTEGER IPAR(*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE
END SUBROUTINE

SUBROUTINE D_INIT_HELMHOLTZ_3D (AX, BX, AY, BY, AZ, BZ, NX, NY, NZ, BCTYPE, Q, IPAR, DPAR,
  STAT)
  USE MKL_DFTI

  INTEGER NX, NY, NZ, STAT
  INTEGER IPAR(*)
  DOUBLE PRECISION AX, BX, AY, BY, AZ, BZ, Q
  DOUBLE PRECISION DPAR(*)

  CHARACTER(6) BCTYPE
END SUBROUTINE

SUBROUTINE D_COMMIT_HELMHOLTZ_3D (F, BD_AX, BD_BX, BD_AY, BD_BY, BD_AZ, BD_BZ, XHANDLE,
  YHANDLE, IPAR, DPAR, STAT)
  USE MKL_DFTI

```

```

    INTEGER STAT

    INTEGER IPAR(*)

    DOUBLE PRECISION F(IPAR(11)+1,IPAR(12)+1,*)

    DOUBLE PRECISION DPAR(*)

    DOUBLE PRECISION BD_AX(IPAR(12)+1,*), BD_BX(IPAR(12)+1,*), BD_AY(IPAR(11)+1,*)
    DOUBLE PRECISION BD_BY(IPAR(11)+1,*), BD_AZ(IPAR(11)+1,*), BD_BZ(IPAR(11)+1,*)

    TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE, YHANDLE

END SUBROUTINE


SUBROUTINE D_HELMHOLTZ_3D (F, BD_AX, BD_BX, BD_AY, BD_BY, BD_AZ, BD_BZ, XHANDLE, YHANDLE,
IPAR, DPAR, STAT)

    USE MKL_DFTI

    INTEGER STAT

    INTEGER IPAR(*)

    DOUBLE PRECISION F(IPAR(11)+1,IPAR(12)+1,*)

    DOUBLE PRECISION DPAR(*)

    DOUBLE PRECISION BD_AX(IPAR(12)+1,*), BD_BX(IPAR(12)+1,*), BD_AY(IPAR(11)+1,*)
    DOUBLE PRECISION BD_BY(IPAR(11)+1,*), BD_AZ(IPAR(11)+1,*), BD_BZ(IPAR(11)+1,*)

    TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE, YHANDLE

END SUBROUTINE


SUBROUTINE S_INIT_HELMHOLTZ_3D (AX, BX, AY, BY, AZ, BZ, NX, NY, NZ, BCTYPE, Q, IPAR, SPAR,
STAT)

    USE MKL_DFTI

```

```

    INTEGER NX, NY, NZ, STAT
    INTEGER IPAR(*)
    REAL AX, BX, AY, BY, AZ, BZ, Q
    REAL SPAR(*)

    CHARACTER(6) BCTYPE
END SUBROUTINE

SUBROUTINE S_COMMIT_HELMHOLTZ_3D (F, BD_AX, BD_BX, BD_AY, BD_BY, BD_AZ, BD_BZ, XHANDLE,
YHANDLE, IPAR, SPAR, STAT)

    USE MKL_DFTI

    INTEGER STAT
    INTEGER IPAR(*)
    REAL F(IPAR(11)+1,IPAR(12)+1,*)
    REAL SPAR(*)
    REAL BD_AX(IPAR(12)+1,*), BD_BX(IPAR(12)+1,*), BD_AY(IPAR(11)+1,*)
    REAL BD_BY(IPAR(11)+1,*), BD_AZ(IPAR(11)+1,*), BD_BZ(IPAR(11)+1,*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE, YHANDLE
END SUBROUTINE

SUBROUTINE S_HELMHOLTZ_3D (F, BD_AX, BD_BX, BD_AY, BD_BY, BD_AZ, BD_BZ, XHANDLE, YHANDLE,
IPAR, SPAR, STAT)

    USE MKL_DFTI

```

```

INTEGER STAT
INTEGER IPAR(*)
REAL F(IPAR(11)+1,IPAR(12)+1,*)
REAL SPAR(*)
REAL BD_AX(IPAR(12)+1,*), BD_BX(IPAR(12)+1,*), BD_AY(IPAR(11)+1,*)
REAL BD_BY(IPAR(11)+1,*), BD_AZ(IPAR(11)+1,*), BD_BZ(IPAR(11)+1,*)
TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE, YHANDLE
END SUBROUTINE

```

```

SUBROUTINE FREE_HELMHOLTZ_3D (XHANDLE, YHANDLE, IPAR, STAT)
    USE MKL_DFTI

    INTEGER STAT
    INTEGER IPAR(*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE, YHANDLE
END SUBROUTINE

```

The Fortran90-specific header file defines the following function prototypes for the spherical solver:

```

SUBROUTINE D_INIT_SPH_P(AP,BP,AT,BT,NP,NT,Q,IPAR,DPAR,STAT)
    USE MKL_DFTI

    INTEGER NP, NT, STAT
    INTEGER IPAR(*)
    DOUBLE PRECISION AP,BP,AT,BT,Q
    DOUBLE PRECISION DPAR(*)

END SUBROUTINE

```

```

SUBROUTINE D_COMMIT_SPH_P(F,HANDLE_S,HANDLE_C,IPAR,DPAR,STAT)
    USE MKL_DFTI

    INTEGER STAT
    INTEGER IPAR(*)
    DOUBLE PRECISION DPAR(*)

    DOUBLE PRECISION F(IPAR(11)+1,*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE_C, HANDLE_S
END SUBROUTINE

```

```

SUBROUTINE D_SPH_P(F,HANDLE_S,HANDLE_C,IPAR,DPAR,STAT)
    USE MKL_DFTI

    INTEGER STAT
    INTEGER IPAR(*)
    DOUBLE PRECISION DPAR(*)

    DOUBLE PRECISION F(IPAR(11)+1,*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE_C, HANDLE_S
END SUBROUTINE

```

```

SUBROUTINE S_INIT_SPH_P(AP,BP,AT,BT,NP,NT,Q,IPAR,SPAR,STAT)
    USE MKL_DFTI

    INTEGER NP, NT, STAT
    INTEGER IPAR(*)
    REAL AP,BP,AT,BT,Q
    REAL SPAR(*)

```

END SUBROUTINE

SUBROUTINE S_COMMIT_SPH_P(F,HANDLE_S,HANDLE_C,IPAR,SPAR,STAT)

USE MKL_DFTI

INTEGER STAT

INTEGER IPAR(*)

REAL SPAR(*)

REAL F(IPAR(11)+1,*)

TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE_C, HANDLE_S

END SUBROUTINE

SUBROUTINE S_SPH_P(F,HANDLE_S,HANDLE_C,IPAR,SPAR,STAT)

USE MKL_DFTI

INTEGER STAT

INTEGER IPAR(*)

REAL SPAR(*)

REAL F(IPAR(11)+1,*)

TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE_C, HANDLE_S

END SUBROUTINE

SUBROUTINE FREE_SPH_P(HANDLE_S,HANDLE_C,IPAR,STAT)

USE MKL_DFTI

```

      INTEGER STAT
      INTEGER IPAR(*)
      TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE_S, HANDLE_C
END SUBROUTINE

SUBROUTINE D_INIT_SPH_NP(AP,BP,AT,BT,NP,NT,Q,IPAR,DPAR,STAT)
  USE MKL_DFTI

  INTEGER NP, NT, STAT
  INTEGER IPAR(*)
  DOUBLE PRECISION AP,BP,AT,BT,Q
  DOUBLE PRECISION DPAR(*)

END SUBROUTINE

SUBROUTINE D_COMMIT_SPH_NP(F,HANDLE,IPAR,DPAR,STAT)
  USE MKL_DFTI

  INTEGER STAT
  INTEGER IPAR(*)
  DOUBLE PRECISION DPAR(*)

  DOUBLE PRECISION F(IPAR(11)+1,*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE
END SUBROUTINE

SUBROUTINE D_SPH_NP(F,HANDLE,IPAR,DPAR,STAT)
  USE MKL_DFTI

```

```

    INTEGER STAT

    INTEGER IPAR(*)

    DOUBLE PRECISION DPAR(*)

    DOUBLE PRECISION F(IPAR(11)+1,*)

    TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE

END SUBROUTINE


SUBROUTINE S_INIT_SPH_NP(AP,BP,AT,BT,NP,NT,Q,IPAR,SPAR,STAT)

    USE MKL_DFTI

    INTEGER NP, NT, STAT

    INTEGER IPAR(*)

    REAL AP,BP,AT,BT,Q

    REAL SPAR(*)

END SUBROUTINE


SUBROUTINE S_COMMIT_SPH_NP(F,HANDLE,IPAR,SPAR,STAT)

    USE MKL_DFTI

    INTEGER STAT

    INTEGER IPAR(*)

    REAL SPAR(*)

    REAL
    F(IPAR(11)+1,*)

    TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE

END SUBROUTINE

```



```

SUBROUTINE S_SPH_NP (F, HANDLE, IPAR, SPAR, STAT)
    USE MKL_DFTI

    INTEGER STAT
    INTEGER IPAR(*)
    REAL SPAR(*)

    REAL F(IPAR(11)+1,*)

    TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE
END SUBROUTINE

```

```

SUBROUTINE FREE_SPH_NP (HANDLE, IPAR, STAT)
    USE MKL_DFTI

    INTEGER STAT
    INTEGER IPAR(*)

    TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE
END SUBROUTINE

```

Fortran 90 specifics of the PL routines usage are similar for all Intel MKL PDE support tools and described in the [Calling PDE Support Routines from Fortran 90](#) section.

Calling PDE Support Routines from Fortran 90

The calling interface for all the Intel MKL TT and PL routines is designed to be easily used in C. However, you can invoke each TT or PL routine directly from Fortran 90 if you are familiar with the inter-language calling conventions of your platform.

The TT or PL interface cannot be invoked from Fortran 77 due to restrictions imposed by the use of the Intel MKL FFT interface.

The inter-language calling conventions include, but are not limited to, the argument passing mechanisms for the language, the data type mappings from C to Fortran 90 and how C external names are decorated on the platform.

To promote portability and relieve a user of dealing with the calling conventions specifics, Fortran 90 header file `mkl_trig_transforms.f90` for TT routines and `mkl_poisson.f90` for PL routines, used together with `mkl_dfti.f90`, declare a set of macros and introduce type definitions intended to hide the inter-language calling conventions and provide an interface to the routines that looks natural in Fortran 90.

For example, consider a hypothetical library routine, `foo`, which takes a double-precision vector of length `n`. C users access such a function as follows:

```
int n;

double *x;

...

foo(x, &n);
```

As noted above, to invoke `foo`, Fortran 90 users would need to know what Fortran 90 data types correspond to C types `int` and `double` (or `float` in case of single-precision), what argument-passing mechanism the C compiler uses and what, if any, name decoration is performed by the C compiler when generating the external symbol `foo`. However, with the Fortran 90 header files `mkl_trig_transforms.f90` / `mkl_poisson.f90` and `mkl_dfti.f90` included, the invocation of `foo` within a Fortran 90 program will look as follows:

- For TT interface,

```
use mkl_dfti
use mkl_trig_transforms

INTEGER n

DOUBLE PRECISION, ALLOCATABLE :: x

...

CALL FOO(x,n)
```

- For PL interface,

```
use mkl_dfti
use mkl_poisson

INTEGER n

DOUBLE PRECISION, ALLOCATABLE :: x

...

CALL FOO(x,n)
```

Note that in the above example, the header files `mkl_trig_transforms.f90` / `mkl_poisson.f90` and `mkl_dfti.f90` provide a definition for the subroutine `FOO`. To ease the use of PL or TT routines in Fortran 90, the general approach of providing Fortran 90 definitions of names is used throughout the libraries. Specifically, if a name from a PL or TT interface is documented as having the C-specific name `foo`, then the Fortran 90 header files provide an appropriate Fortran 90 language type definition `FOO`.

One of the key differences between Fortran 90 and C is the language argument-passing mechanism: C programs use pass-by-value semantics and Fortran 90 programs use pass-by-reference semantics. The Fortran 90 headers ensure proper treatment of this difference. In particular, in the above example, the header files `mkl_trig_transforms.f90` / `mkl_poisson.f90` and `mkl_dfti.f90` hide the difference by defining a macro `FOO` that takes the address of the appropriate arguments.

Optimization Solver Routines

Intel® Math Kernel Library provides tools for solving optimization problems. These tools are routines for solving nonlinear least squares problems through the Trust-Region (TR) algorithms.

Intel MKL provides the optimization solver routines that can be used for:

- solving nonlinear least squares problems without constraints
- solving nonlinear least squares problems with boundary constraints
- computing the Jacobi matrix by central differences for solving nonlinear least squares problem

The first two groups of routines are designed to find only one local minimum point. However problems can have multiple local minima and trying different initial points is recommended for better solutions.

For more information on terms and key concepts required to understand the use of the Intel MKL nonlinear least squares problem solver routines, see [Appendix F, “Optimization Solvers Basics”](#).

Routines described below are subdivided according to their purpose as follows:

[Nonlinear Least Squares Problem without Constraints](#)

[Nonlinear Least Squares Problem with Linear \(Boundary\) Constraints](#)

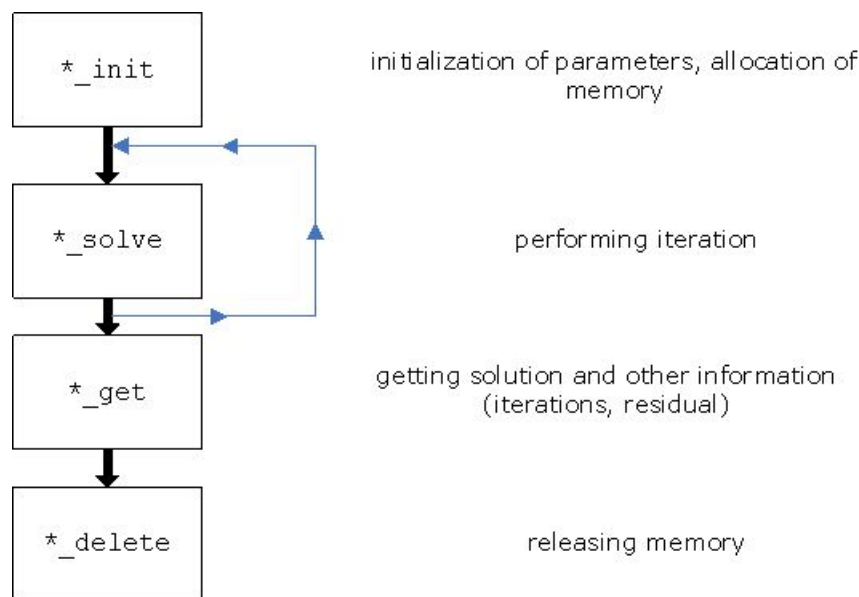
[Jacobi Matrix Calculation Routines](#).

Organization and Implementation

The TR solvers have RCI-based interfaces. The RCI TR interface implements a group of user-callable routines that are used in the step-by-step solving process for nonlinear least squares problem and follows the general RCI scheme described in [[Dong95](#)]. RCI means that the user is supposed to perform certain operations for the solver, for example, to provide values of the objective function

or a Jacobi matrix. When the solver needs the results of such operations, the user should pass them to the solver. This makes the solver independent of specific implementation of the operations. However, this approach requires some additional work by the user.

Figure 14-1 Typical order for invoking RCI solver routines



The Trust-Region solvers are implemented with OpenMP* support. To use the multiprocessing mode, set the number of threads in environment variable `OMP_NUM_THREADS`.

Memory Allocation and Handles

To make the routines easy to use, you are not required to allocate temporary working storage. Any required memory is allocated by the solver. To allow multiple users to access the solver simultaneously, the solver keeps track of the storage allocated for a particular application by using an opaque data object called a *handle*. Each of the Intel RCI TR solver routine creates, uses or deletes a handle.

Handle declaration varies from language to language. This document declares it as to be of `_TRNSP_HANDLE_t` or `_TRNSPBC_HANDLE_t` type.

C and C++ programmers should declare a handle as:

```
#include "mkl_rci.h"

_TRNSP_HANDLE_t handle;

or

_TRNSPBC_HANDLE_t handle;
```

Fortran programmers using compilers that support eight byte integers should declare a handle as:

```
INCLUDE "mkl_rci.fi"

INTEGER*8 handle
```

In addition to the necessary definition, for the correct declaration of a handle, the include file also defines the following:

- function prototypes for languages that support prototypes
- symbolic constants that are used for the error returns
- user options for the solver routines
- message severity.

Nonlinear Least Squares Problem without Constraints

The nonlinear least squares problem without constraints can be described as follows:

$$\min_{x \in R^n} F(x) = \|y - f(x)\|_2^2, y \in R^m, x \in R^n, f: R^n \rightarrow R^m, m > n,$$

where $F(x) : R^n \rightarrow R$ is a twice differentiable function in R^n . Solving of the nonlinear least squares problem is searching for the best approximation to vector y with model function $f_i(x)$, which has nonlinear dependence on variables x . The best approximation means that the sum of squares of residuals $y_i - f_i(x)$ is the lowest possible.

Table 14-1 RCI TR Routines

Routine Name	Operation
dtrnlsp_init	Initializes the solver.

Routine Name	Operation
<code>dtrnlsp_solve</code>	Solves a nonlinear least squares problem on the basis of RCI using the Trust-Region algorithm.
<code>dtrnlsp_get</code>	Retrieves the number of iterations, stop criterion, initial residual, and final residual.
<code>dtrnlsp_delete</code>	Removes data.

dtrnlsp_init

Initializes the solver of nonlinear least squares problem.

Syntax

Fortran:

```
res = dtrnlsp_init(handle, n, m, x, eps, iter1, iter2, rs)
```

C:

```
res = dtrnlsp_init(&handle, &n, &m, x, eps, &iter1, &iter2, &rs);
```

Description

The routine initializes the solver. After initialization all subsequent invocations of the `dtrnlsp_solve` routine should use the values of the handle returned by `dtrnlsp_init`.

The *eps* array contains the stopping tests:

eps(1): $\Delta_k < \text{eps}(1)$

eps(2): $\|F(x)\| < \text{eps}(2)$

eps(3): $\|A(x)_{ij}\| < \text{eps}(3)$

eps(4): $\|s\| < \text{eps}(4)$

eps(5): $\|F(x)\| - \|F(x) - A(x)s\| < \text{eps}(5)$

eps(6): the trial step precision. If *eps*(6) = 0, then *eps*(6) = 1.d-10,

where *A* is the Jacobi matrix, Δ_k is the trust region area, *F*(*x*) is the value of the functional, *s* is the trial step.

Input Parameters

<i>n</i>	INTEGER. Length of x .
<i>m</i>	INTEGER. Length of $F(x)$.
<i>x</i>	DOUBLE PRECISION. Array of size n . Initial guess.
<i>eps</i>	DOUBLE PRECISION. Array of size 6; contains stopping tests. See the values in Description.
<i>iter1</i>	INTEGER. Specifies the maximum number of iterations.
<i>iter2</i>	INTEGER. Specifies the maximum number of iterations of trial-step calculation.
<i>rs</i>	DOUBLE PRECISION. Positive input variable used in determining the initial step bound. In most cases the factor should lie within the interval (0.1, 100.0). The generally recommended value is 100.

Output Parameters

<i>handle</i>	Data object of the <code>_TRNSP_HANDLE_t</code> type in C/C++ and <code>INTEGER*8</code> in FORTRAN.
<i>res</i>	INTEGER. Informs about the task completion. <i>res</i> = <code>TR_SUCCESS</code> means the routine completed the task normally. <i>res</i> = <code>TR_INVALID_OPTION</code> means an error in the input parameters. <i>res</i> = <code>TR_OUT_OF_MEMORY</code> means a memory error.

dtrnlsp_solve

Solves a nonlinear least squares problem using the Trust-Region algorithm.

Syntax

Fortran:

```
res = dtrnlsp_solve(handle, fvec, fjac, RCI_Request)
```

C:

```
res = dtrnlsp_solve(&handle, fvec, fjac, &RCI_Request);
```

Description

The `dtrnlsp_solve` routine, based on Reverse Communication Interface (RCI), uses the Trust-Region algorithm to solve nonlinear least squares problems.

The problem is stated as follows:

$$\min_{x \in \mathbb{R}^n} \frac{1}{2} F(x)^T F(x) = \frac{1}{2} \sum_{i=1}^m f_i(x)^2,$$

where $m \geq n$, $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$ and $f_i(x)$ is the i -th component function of $F(x)$. From a current point, the algorithm uses the trust-region approach:

$$\min_{x \in \mathbb{R}^n} \|F(x_c) + J(x_c)(x_n - x_c)\| \quad \text{subject to} \quad \|x_n - x_c\|_2 \leq \Delta_c$$

to get the new point x_n as the solution to the following problem:

$$\min_{x \in \mathbb{R}^n} \|J^T(x)J(x) + J^T F(x)s\|_2,$$

where s is the trial step, and $\|s\|_2 \leq \Delta_c$,

then $x_n = x_c + s$.

The `RCI_Request` parameter informs about the task completion and may have the following values:

`RCI_Request`= 2 that indicates the requirement to calculate the Jacobian matrix and put the result into `fjac`.

`RCI_Request`= 1 that indicates the requirement to recalculate the function at vector `x` and put the result into `fvec`.

`RCI_Request`= 0 that indicates successful completion of the task

`RCI_Request`= -1 indicating that the algorithm has exceeded the maximal number of iterations.

$RCI\_Request = -2$ indicating that $\Delta_k < eps(1)$

$RCI\_Request = -3$ indicating that $\|F(x)\| < eps(2)$

$RCI\_Request = -4$ indicating that $\|A(x)_{ij}\| < eps(3)$

$RCI\_Request = -5$ indicating that $\|s\| < eps(4)$

$RCI\_Request = -6$ indicating that $\|F(x)\| - \|F(x) - A(x)s\| < eps(5)$,

where A is the Jacobi matrix, Δ_k is the trust region area, $F(x)$ is the value of the functional, s is the trial step.

Input Parameters

<i>handle</i>	Data object of the <code>_TRNSP_HANDLE_t</code> type in C/C++ and <code>INTEGER*8</code> in FORTRAN.
<i>fvec</i>	DOUBLE PRECISION. Array of size m . Contains the function values at X , where $fvec(i) = (y_i - f_i(x))$.
<i>fjac</i>	DOUBLE PRECISION. Array of size (m,n) . Contains the Jacobi matrix of the function.

Output Parameters

<i>fvec</i>	DOUBLE PRECISION. Array of size m . Contains the updated function values at x
<i>RCI_Request</i>	INTEGER. Informs about the task completion. When equal to 0, the task is completed successfully. See Description for the other values of the parameter and their meaning.
<i>res</i>	INTEGER. Informs about the task completion. $res = TR\_SUCCESS$ means the routine completed the task normally.

dtrnlsp_get

Retrieves the number of iterations, stop criterion, initial residual, and final residual.

Syntax

Fortran:

```
res = dtrnlsp_get(handle, iter, st_cr, r1, r2)
```

C:

```
res = dtrnlsp_get(&handle, &iter, &st_cr, &r1, &r2);
```

Description

The routine retrieves the current number of iterations, stop criterion, initial residual, and final residual.

The *st_cr* parameter contains the stop criterion:

st_cr = 1 indicates that the algorithm has exceeded the maximal number of iterations.

st_cr = 2 indicates that $\Delta_k < \text{eps}(1)$

st_cr = 3 indicates that $||F(x)|| < \text{eps}(2)$

st_cr = 4 indicates that $||A(x)_{ij}|| < \text{eps}(3)$

st_cr = 5 indicates that $||s|| < \text{eps}(4)$

st_cr = 6 indicates that $||F(x)|| - ||F(x) - A(x)s|| < \text{eps}(5)$,

where *A* is the Jacobi matrix, Δ_k is the trust region area, *F(x)* is the value of the functional, and *s* is the trial step.

Input Parameters

handle Data object of the `_TRNSP_HANDLE_t` type in C/C++ and `INTEGER*8` in FORTRAN.

Output Parameters

iter INTEGER. Contains the current number of iterations.

st_cr INTEGER. Contains the stop criterion.

See Description for the parameter values and their meanings.

r1 DOUBLE PRECISION. Contains the initial residual, that is, the value of the functional $(||y - f(x)||)$ of the initial *x* set by the user.

r2 DOUBLE PRECISION. Contains the final residual, that is, the value of the functional $(||y - f(x)||)$ of the final *x* resulting from the algorithm operation.

res INTEGER. Informs about the task completion.

res = `TR_SUCCESS` means the routine completed the task normally.

dtrnlsp_delete

Removes the data object required by the TR solver.

Syntax

Fortran:

```
res = dtrnlsp_delete(handle)
```

C:

```
res = dtrnlsp_delete(&handle);
```

Description

The routine removes the data object needed for the RCI TR solver.

Input Parameters

<i>handle</i>	Data object of the <code>_TRNSP_HANDLE_t</code> type in C/C++ and <code>INTEGER*8</code> in FORTRAN.

Output Parameters

<i>res</i>	<code>INTEGER</code> . Informs about the task completion. <i>res</i> = <code>TR_SUCCESS</code> means the routine completed the task normally.

Examples of dtrnlsp Usage

Example 14-1. dtrnlsp Usage in Fortran

```

C** NONLINEAR LEAST SQUARE PROBLEM WITHOUT BOUNDARY CONSTRAINTS

    PROGRAM EXAMPLE_DTRNLSP_POWELL

        IMPLICIT NONE

C** HEADER-FILE WITH DEFINITIONS (CONSTANTS, EXTERNALS)

        INCLUDE 'mkl_rci.fi'

C** USER'S OBJECTIVE FUNCTION

        EXTERNAL          EXTENDET_POWELL

C** N - NUMBER OF FUNCTION VARIABLES

        INTEGER           N

        PARAMETER         (N = 40)

C** M - DIMENSION OF FUNCTION VALUE

        INTEGER           M

        PARAMETER         (M = 40)

C** SOLUTION VECTOR. CONTAINS VALUES X FOR F(X)

        DOUBLE PRECISION  X (N)

C** PRECISIONS FOR STOP-CRITERIA (SEE MANUAL FOR MORE DETAILS)

        DOUBLE PRECISION  ESP (6)

C** JACOBI CALCULATION PRECISION

        DOUBLE PRECISION  JAC_EPS

C** REVERSE COMMUNICATION INTERFACE PARAMETER

        INTEGER           RCI_REQUEST

C** FUNCTION (F(X)) VALUE VECTOR

        DOUBLE PRECISION  FVEC (M)

C** JACOBI MATRIX

        DOUBLE PRECISION  FJAC (M, N)

```

```
C** NUMBER OF ITERATIONS
      INTEGRER          ITER
C** NUMBER OF STOP-CRITERION
      INTEGRER          ST_CR
C** CONTROLS OF RCI CYCLE
      INTEGRER          SUCCESSFUL
C** MAXIMUM NUMBER OF ITERATIONS
      INTEGRER          ITER1
C** MAXIMUM NUMBER OF ITERATIONS OF CALCULATION OF TRIAL-STEP
      INTEGRER          ITER2
C** INITIAL STEP BOUND
      DOUBLE PRECISION  RS
C** INITIAL AND FINAL RESIDUALS
      DOUBLE PRECISION  R1, R2
C** TR SOLVER HANDLE
      INTEGRER*8        HANDLE
C** CYCLE'S COUNTERS
      INTEGRER          I, J

C** SET PRECISIONS FOR STOP-CRITERIA
      EPS (1:6) = 1.D-5
C** SET MAXIMUM NUMBER OF ITERATIONS
      ITER1 = 1000
C** SET MAXIMUM NUMBER OF ITERATIONS OF CALCULATION OF TRIAL-STEP
      ITER2 = 100
C** SET INITIAL STEP BOUND
      RS = 100.D0
C** PRECISIONS FOR JACOBI CALCULATION
```

```

        JAC_EPS = 1.D-8
C** SET THE INITIAL GUESS
        DO I = 1, N/4
            X (4*I - 3) = 3.D0
            X (4*I - 2) = -1.D0
            X (4*I - 1) = 0.D0
            X (4*I)      = 1.D0
        ENDDO
C** SET INITIAL VALUES
        DO I = 1, M
            FVEC (I) = 0.D0
            DO J = 1, N
                FJAC (I, J) = 0.D0
            ENDDO
        ENDDO
C** INITIALIZE SOLVER (ALLOCATE MEMORY, SET INITIAL VALUES)
C**  HANDLE      IN/OUT: TR SOLVER HANDLE
C**  N           IN:      NUMBER OF FUNCTION VARIABLES
C**  M           IN:      DIMENSION OF FUNCTION VALUE
C**  X           IN:      SOLUTION VECTOR. CONTAINS VALUES X FOR F(X)
C**  EPS         IN:      PRECISIONS FOR STOP-CRITERIA
C**  ITER1       IN:      MAXIMUM NUMBER OF ITERATIONS
C**  ITER2       IN:      MAXIMUM NUMBER OF ITERATIONS OF CALCULATION OF TRIAL-STEP
C**  RS          IN:      INITIAL STEP BOUND
        IF (DTRNLSP_INIT (HANDLE, N, M, X, EPS, ITER1, ITER2, RS)
+         /= TR_SUCCESS) THEN
C** IF FUNCTION DOES NOT COMPLETE SUCCESSFULLY THEN PRINT ERROR MESSAGE
        PRINT *, '| ERROR IN DTRNLSP_INIT'
```



```
C** AND STOP
        STOP
    ENDIF
C** SET INITIAL RCI CYCLE VARIABLES
        RCI_REQUEST = 0
        SUCCESSFUL = 0
C** RCI CYCLE
        DO WHILE (SUCCESSFUL == 0)
C** CALL TR SOLVER
C**   HANDLE           IN/OUT: TR SOLVER HANDLE
C**   FVEC             IN:      VECTOR
C**   FJAC             IN:      JACOBI MATRIX
C**   RCI_REQUEST      IN/OUT: RETURN NUMBER THAT DENOTES NEXT STEP FOR PERFORMING
        IF (DTRNLSP_SOLVE (HANDLE, FVEC, FJAC, RCI_REQUEST)
            +   /= TR_SUCCESS) THEN
C** IF FUNCTION DOES NOT COMPLETE SUCCESSFULLY THEN PRINT ERROR MESSAGE
        PRINT *, '| ERROR IN DTRNLSP_SOLVE'
C** AND STOP
        STOP
    ENDIF
C**   RCI_REQUEST      IN/OUT: RETURN NUMBER THAT DENOTES NEXT STEP FOR PERFORMING
C** ACCORDING TO RCI_REQUEST VALUE WE DO NEXT STEP
        SELECT CASE (RCI_REQUEST)
        CASE (-1, -2, -3, -4, -5, -6)
C**   STOP RCI CYCLE
        SUCCESSFUL = 1
        CASE (1)
C**   RECALCULATE FUNCTION VALUE
```

```

C**      M                IN:      DIMENSION OF FUNCTION VALUE
C**      N                IN:      NUMBER OF FUNCTION VARIABLES
C**      X                IN:      SOLUTION VECTOR
C**      FVEC             OUT:      FUNCTION VALUE F(X)

          CALL EXTENDET_POWELL (M, N, X, FVEC)

          CASE (2)

C**      COMPUTE JACOBI MATRIX
C**      EXTENDET_POWELL IN:      EXTERNAL OBJECTIVE FUNCTION
C**      N                IN:      NUMBER OF FUNCTION VARIABLES
C**      M                IN:      DIMENSION OF FUNCTION VALUE
C**      FJAC             OUT:      JACOBI MATRIX
C**      X                IN:      SOLUTION VECTOR
C**      JAC_EPS          IN:      JACOBI CALCULATION PRECISION

          IF (DJACOBI (EXTENDET_POWELL, N, M, FJAC, X, JAC_EPS)
+           /= TR_SUCCESS) THEN
C** IF FUNCTION DOES NOT COMPLETE SUCCESSFULLY THEN PRINT ERROR MESSAGE

          PRINT *, '| ERROR IN DTRNLSP_SOLVE'

C** AND STOP

          STOP

          ENDIF

          ENDSELECT

          ENDDO

C** GET SOLUTION STATUSES
C**      HANDLE          IN:      TR SOLVER HANDLE
C**      ITER            OUT:      NUMBER OF ITERATIONS
C**      ST_CR           OUT:      NUMBER OF STOP CRITERION
C**      R1              OUT:      INITIAL RESIDUALS
C**      R2              OUT:      FINAL RESIDUALS

```

```

        IF (DTRNLSP_GET (HANDLE, ITER, ST_CR, R1_R2)
+      /= TR_SUCCESS) THEN
C** IF FUNCTION DOES NOT COMPLETE SUCCESSFULLY THEN PRINT ERROR MESSAGE
        PRINT *, '| ERROR IN DTRNLSP_GET'
C** AND STOP
        STOP
    ENDIF
C** FREE HANDLE MEMORY
        IF (DTRNLSP_DELETE (HANDLE) /= TR_SUCCESS) THEN
C** IF FUNCTION DOES NOT COMPLETE SUCCESSFULLY THEN PRINT ERROR MESSAGE
        PRINT *, '| ERROR IN DTRNLSP_DELETE'
C** AND STOP
        STOP
    ENDIF

C** IF FINAL RESIDUAL IS LESS THAN REQUIRED PRECISION THEN PRINT PASS
        IF (R2 < 1.D-5) THEN
            PRINT *, '|          DTRNLSP POWELL.....PASS'!, R1, R2
C** ELSE PRINT FAILED
        ELSE
            PRINT *, '|          DTRNLSP POWELL.....FAILED'!, R1, R2
        ENDIF

    END PROGRAM EXAMPLE_DTRNLSP_POWELL

C** ROUTINE FOR EXTENDED POWELL FUNCTION CALCULATION
C**  M          IN:      DIMENSION OF FUNCTION VALUE
C**  N          IN:      NUMBER OF FUNCTION VARIABLES

```

```

C**  X          IN:    VECTOR FOR FUNCTION CALCULATION
C**  F          OUT:    FUNCTION VALUE F(X)

SUBROUTINE EXTENDET_POWELL (M, N, X, F)

    IMPLICIT NONE

    INTEGER M, N

    DOUBLE PRECISION X (*), F (*)

    INTEGER I

    DO I = 1, N/4

        F (4*I-3) = X(4*I - 3) + 10.D0 * X(4*I - 2)

        F (4*I-2) = 2.2360679774997896964091736687313D0*(X(4*I-1) - X(4*I))

        F (4*I-1) = (X(4*I-2) - 2.D0*X(4*I-1))**2

        F (4*I)   = 3.1622776601683793319988935444327D0*(X(4*I-3) - X(4*I))**2

    ENDDO

ENDSUBROUTINE EXTENDET_POWELL

```

Example 14-2. dtrnlsp Usage in C

```
#include <stdio.h>
#include <malloc.h>
#include <math.h>

#include "mkl_rci.h"

/* nonlinear least square problem without boundary constraints */
int main ()
{
    /* user's objective function */
    extern void extendet_powell (int *, int *, double*, double*);
    /* n - number of function variables
       m - dimension of function value */
    int          n = 4, m = 4;
    /* precisions for stop-criteria (see manual for more details) */
    double          eps[6];
    /* solution vector. contains values x for f(x) */
    double          *x;
    /* iter1 - maximum number of iterations
       iter2 - maximum number of iterations of calculation of trial-step */
    int          iter1 = 1000, iter2 = 100;
    /* initial step bound */
    double          rs = 0.0;
    /* reverse communication interface parameter */
    int          RCI_Request; // reverse communication interface variable
    /* controls of rci cycle */
    int          successful;
```

```

/* function (f(x)) value vector */
double          *fvec;

/* jacobi matrix */
double          *fjac;

/* number of iterations */
int             iter;

/* number of stop-criterion */
int             st_cr;

/* initial and final residuals */
double          r1, r2;

/* TR solver handle */
_TRNSP_HANDLE_t handle; // TR solver handle

/* cycle's counter */
int i;

/* memory allocation */
x = (double*) malloc (sizeof (double)*n);
fvec = (double*) malloc (sizeof (double)*m);
fjac = (double*) malloc (sizeof (double)*m*n);

/* set precisions for stop-criteria */
for (i = 0; i < 6; i++)
{
    eps [i] = 0.00001;
}

/* set the initial guess */
for (i = 0; i < n/4; i++)
{
    x [4*i]      = 3.0;

```

```

    x [4*i + 1] = -1.0;
    x [4*i + 2] =  0.0;
    x [4*i + 3] =  1.0;
}
/* set the initial values */
for (i = 0; i < m; i++)
    fvec [i] = 0.0;
for (i = 0; i < m*n; i++)
    fjac [i] = 0.0;
/* initialize solver (allocate memory, set initial values)
   handle  in/out: TR solver handle
   n       in:    number of function variables
   m       in:    dimension of function value
   x       in:    solution vector. contains values x for f(x)
   eps     in:    precisions for stop-criteria
   iter1   in:    maximum number of iterations
   iter2   in:    maximum number of iterations of calculation of trial-step
   rs      in:    initial step bound */
if (dtrnlsp_init (&handle, &n, &m, x, eps, &iter1, &iter2, &rs) != TR_SUCCESS)
{
    /* if function does not complete successfully then print error message */
    printf ("| error in dtrnlsp_init\n");
    /* and exit */
    return 0;
}
/* set initial rci cycle variables */
RCI_Request = 0;
successful = 0;

```

```

/* rci cycle */
while (successful == 0)
{
    /* call tr solver
        handle      in/out: tr solver handle
        fvec        in:      vector
        fjac        in:      jacobi matrix
        RCI_request in/out: return number which denotes next step for performing */
    if (dtrnlsp_solve (&handle, fvec, fjac, &RCI_Request) != TR_SUCCESS)
    {
        /* if function does not complete successfully then print error message */
        printf ("| error in dtrnlsp_solve\n");
        /* and exit */
        return 0;
    }
    /* according to rci_request value we do next step */
    if (RCI_Request == -1 ||
        RCI_Request == -2 ||
        RCI_Request == -3 ||
        RCI_Request == -4 ||
        RCI_Request == -5 ||
        RCI_Request == -6)
        /* exit rci cycle */
        successful = 1;
    if (RCI_Request == 1)
    {
        /* recalculate function value
            m      in:      dimension of function value

```

```

        n      in:    number of function variables
        x      in:    solution vector
        fvec   out:    function value f(x) */
    extendet_powell (&m, &n, x, fvec);
}
if (RCI_Request == 2)
{
    /* compute jacobi matrix
        extendet_powell in:    external objective function
        n              in:    number of function variables
        m              in:    dimension of function value
        fjac           out:    jacobi matrix
        x              in:    solution vector
        jac_eps        in:    jacobi calculation precision */
    if (djacobi (extendet_powell, &n, &m, fjac, x, eps) != TR_SUCCESS)
    {
        /* if function does not complete successfully then print error message */
        printf ("| error in djacobi\n");
        /* and exit */
        return 0;
    }
}
}
/* get solution statuses
    handle          in:    TR solver handle
    iter            out:    number of iterations
    st_cr           out:    number of stop criterion
    r1              out:    initial residuals

```

```

        r2                out:    final residuals */
if (dtrnlsp_get (&handle, &iter, &st_cr, &r1, &r2) != TR_SUCCESS)
{
    /* if function does not complete successfully then print error message */
    printf ("| error in dtrnlsp_get\n");
    /* and exit */
    return 0;
}
/* free handle memory */
if (dtrnlsp_delete (&handle) != TR_SUCCESS)
{
    /* if function does not complete successfully then print error message */
    printf ("| error in dtrnlsp_delete\n");
    /* and exit */
    return 0;
}
/* free allocated memory */
free (x);
free (fvec);
free (fjac);
/* if final residual is less than required precision then print pass */
if (r2 < 0.00001)
    printf ("|          dtrnlsp powell.....PASS\n");
/* else print failed */
else
    printf ("|          dtrnlsp powell.....FAILED\n");
return 0;
}

```

```

/* nonlinear system equations without constraints */
/* routine for extendet powell function calculation
m      in:      dimension of function value
n      in:      number of function variables
x      in:      vector for function calculation
f      out:     function value f(x) */
void extendet_powell (int *m, int *n, double *x, double *f)
{
    int i;

    for (i = 0; i < (*n)/4; i++)
    {
        f [4*i] = x [4*i] + 10.0*x [4*i + 1];
        f [4*i + 1] = 2.2360679774997896964091736687313*(x [4*i + 2] - x [4*i + 3]);
        f [4*i + 2] = (x [4*i + 1] - 2.0*x [4*i + 2])*(x [4*i + 1] - 2.0*x [4*i + 2]);
        f [4*i + 3] = 3.1622776601683793319988935444327*(x [4*i] - x [4*i + 3])*(x [4*i] - x [4*i
+ 3]);
    }
    return;
}

```

Nonlinear Least Squares Problem with Linear (Bound) Constraints

The nonlinear least squares problem with linear bound constraints can be described and solved in the same way as the [nonlinear least squares problem without constraints](#) but it has the following constraints:

$$l_i \leq x_i \leq u_i, i = 1, \dots, n, \quad l, u \in R^n.$$

Table 14-2 RCI TR Routines for Problem with Bound Constraints

Routine Name	Operation
<code>dtrnlspbc_init</code>	Initializes the solver.
<code>dtrnlspbc_solve</code>	Solves a nonlinear least squares problem on the basis of RCI using the Trust-Region algorithm.
<code>dtrnlspbc_get</code>	Retrieves the number of iterations, stop criterion, initial residual, and final residual.
<code>dtrnlspbc_delete</code>	Removes data.

`dtrnlspbc_init`

Initializes the solver of nonlinear least squares problem with linear (boundary) constraints.

Syntax

Fortran:

```
res = dtrnlspbc_init(handle, n, m, x, LW, UP, eps, iter1, iter2, rs)
```

C:

```
res = dtrnlspbc_init(&handle, &n, &m, x, LW, UP, eps, &iter1, &iter2, &rs);
```

Description

The routine initializes the solver. After initialization all invocations of the `dtrnlspbc_solve` routine should use the values of the handle returned by `dtrnlspbc_init`.

The *eps* array contains the stopping tests:

```
eps(1):  $\Delta_k < \text{eps}(1)$   
eps(2):  $\|F(x)\| < \text{eps}(2)$   
eps(3):  $\|A(x)_{ij}\| < \text{eps}(3)$   
eps(4):  $\|s\| < \text{eps}(4)$   
eps(5):  $\|F(x)\| - \|F(x) - A(x)s\| < \text{eps}(5)$ 
```

$eps(6)$: the trial step precision. If $eps(6) = 0$, then $eps(6) = 1.d-10$,

where A is the Jacobi matrix, Δ_k is the trust region area, $F(x)$ is the value of the functional, s is the trial step.

Input Parameters

n	INTEGER. Length of x .
m	INTEGER. Length of $F(x)$.
x	DOUBLE PRECISION. Array of size n . Initial guess.
LW	DOUBLE PRECISION. Array of size n . Contains low bounds for x ($lw_i \leq x_i$).
UP	DOUBLE PRECISION. Array of size n . Contains upper bounds for x ($up_i \leq x_i$).
eps	DOUBLE PRECISION. Array of size 6; contains stopping tests. See the values in Description.
$iter1$	INTEGER. Specifies the maximum number of iterations.
$iter2$	INTEGER. Specifies the maximum number of iterations of trial-step calculation.
rs	DOUBLE PRECISION. Positive input variable used in determining the initial step bound. In most cases the factor should lie within the interval (0.1, 100.0). The generally recommended value is 100.

Output Parameters

$handle$	Data object of the <code>_TRNSPBC_HANDLE_t</code> type in C/C++ and <code>INTEGER*8</code> in FORTRAN.
res	INTEGER. Informs about the task completion. $res = TR\_SUCCESS$ indicates that the routine has completed the task normally. $res = TR\_INVALID\_OPTION$ indicates an error in the input parameters. $res = TR\_OUT\_OF\_MEMORY$ indicates a memory error.

dtrnlspbc_solve

Solves a nonlinear least squares problem with linear (bound) constraints using the Trust-Region algorithm.

Syntax

Fortran:

```
res = dtrnlspbc_solve(handle, fvec, fjac, RCI_Request)
```

C:

```
res = dtrnlspbc_solve(&handle, fvec, fjac, &RCI_Request);
```

Description

The `dtrnlspbc_solve` routine, based on Reverse Communication Interface (RCI), uses the Trust-Region algorithm to solve nonlinear least squares problems with linear (bound) constraints. The problem is stated as follows:

$$\min_{x \in \mathbb{R}^n} \frac{1}{2} F(x)^T F(x) = \frac{1}{2} \sum_{i=1}^m f_i(x)^2, \text{ where } l_i \leq x_i \leq u_i \text{ } i = 1, \dots, n,$$

where $m \geq n$, $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$ and $f_i(x)$ is the i -th component function of $F(x)$.

The `RCI_Request` parameter informs about the task completion and may have the following values:

`RCI_Request = 2` that indicates the requirement to calculate the Jacobian matrix and put the result into `fjac`.

`RCI_Request = 1` that indicates the requirement to recalculate the function at vector `x` and put the result into `fvec`.

`RCI_Request = 0` that indicates successful completion of the task

`RCI_Request = -1` indicating that the algorithm has exceeded the maximal number of iterations.

`RCI_Request = -2` indicating that $\Delta_k < \text{eps}(1)$

`RCI_Request = -3` indicating that $\|F(x)\| < \text{eps}(2)$

$RCI\_Request = -4$ indicating that $||A(x)_{ij}|| < eps(3)$

$RCI\_Request = -5$ indicating that $||s|| < eps(4)$

$RCI\_Request = -6$ indicating that $||F(x)|| - ||F(x) - A(x)s|| < eps(5)$,

where A is the Jacobi matrix, Δ_k is the trust region area, $F(x)$ is the value of the functional, s is the trial step.

Input Parameters

handle Data object of the `_TRNSPBC_HANDLE_t` type in C/C++ and `INTEGER*8` in FORTRAN.

fvec DOUBLE PRECISION. Array of size m . Contains the function values at X , where $fvec(i) = (y_i - f_i(x))$.

fjac DOUBLE PRECISION. Array of size m by n . Contains the Jacobi matrix of the function.

Output Parameters

fvec DOUBLE PRECISION. Array of size m . Contains the updated function values at X

RCI_Request INTEGER. Informs about the task completion. When equal to 0, the task is completed successfully.
See Description for the other values of the parameter and their meaning.

res INTEGER. Informs about the task completion.
 $res = TR\_SUCCESS$ means the routine completed the task normally.

dtrnlspsc_get

Retrieves the number of iterations, stop criterion, initial residual, and final residual.

Syntax

Fortran:

```
res = dtrnlspsc_get(handle, iter, st_cr, r1, r2)
```

C:

```
res = dtrnlspsc_get(&handle, &iter, &st_cr, &r1, &r2);
```

Description

The routine retrieves the current number of iterations, stop criterion, initial residual, and final residual.

The *st_cr* parameter contains the stop criterion:

st_cr = 1 indicates that the algorithm has exceeded the maximal number of iterations.

st_cr = 2 indicates that $\Delta_k < \text{eps}(1)$

st_cr = 3 indicates that $\|F(x)\| < \text{eps}(2)$

st_cr = 4 indicates that $\|A(x)_{ij}\| < \text{eps}(3)$

st_cr = 5 indicates that $\|s\| < \text{eps}(4)$

st_cr = 6 indicates that $\|F(x)\| - \|F(x) - A(x)s\| < \text{eps}(5)$,

where *A* is the Jacobi matrix, Δ_k is the trust region area, *F*(*x*) is the value of the functional, *s* is the trial step.

Input Parameters

handle Data object of the `_TRNSPBC_HANDLE_t` type in C/C++ and `INTEGER*8` in FORTRAN.

Output Parameters

<i>iter</i>	INTEGER. Contains the current number of iterations.
<i>st_cr</i>	INTEGER. Contains the stop criterion. See Description for the parameter values and their meanings.
<i>r1</i>	DOUBLE PRECISION. Contains the initial residual, that is, the value of the function $\ y - f(x)\ $ of the initial <i>x</i> set by the user.
<i>r2</i>	DOUBLE PRECISION. Contains the final residual, that is, the value of the function $\ y - f(x)\ $ of the final <i>x</i> resulting from the algorithm operation.
<i>res</i>	INTEGER. Informs about the task completion. <i>res</i> = <code>TR_SUCCESS</code> means the routine completed the task normally.

dtrnlspbc_delete

Removes the data object required by the TR solver.

Syntax

Fortran:

```
res = dtrnlspbc_delete(handle)
```

C:

```
res = dtrnlspbc_delete(&handle);
```

Description

The routine removes the data object needed for the RCI TR solver.

Input Parameters

handle Data object of the `_TRNSPBC_HANDLE_t` type in C/C++ and `INTEGER*8` in FORTRAN.

Output Parameters

res INTEGER. Informs about the task completion.
res = `TR_SUCCESS` means the routine completed the task normally.

Examples of dtrnlspbc Usage

Example 14-3. dtrnlspbc Usage in Fortran

```
C** NONLINEAR LEAST SQUARE PROBLEM WITH BOUNDARY CONSTRAINTS

      PROGRAM EXAMPLE_DTRNLSPBC_POWELL

      IMPLICIT NONE

C** HEADER-FILE WITH DEFINITIONS (CONSTANTS, EXTERNALS)

      INCLUDE 'mkl_rci.f'

C** USER'S OBJECTIVE FUNCTION

      EXTERNAL          EXTENDET_POWELL

C** N - NUMBER OF FUNCTION VARIABLES

      INTEGER           N

      PARAMETER         (N = 40)

C** M - DIMENSION OF FUNCTION VALUE

      INTEGER           M

      PARAMETER         (M = 40)

C** SOLUTION VECTOR. CONTAINS VALUES X FOR F(X)

      DOUBLE PRECISION  X (N)

C** PRECISIONS FOR STOP-CRITERIA (SEE MANUAL FOR MORE DETAILS)

      DOUBLE PRECISION  ESP (6)

C** JACOBI CALCULATION PRECISION

      DOUBLE PRECISION  JAC_EPS

C** LOWER AND UPPER BOUNDS

      DOUBLE PRECISION  LW (N), UP (N)

C** REVERSE COMMUNICATION INTERFACE PARAMETER

      INTEGER           RCI_REQUEST

C** FUNCTION (F(X)) VALUE VECTOR

      DOUBLE PRECISION  FVEC (M)
```

```
C** JACOBI MATRIX
      DOUBLE PRECISION    FJAC (M, N)
C** NUMBER OF ITERATIONS
      INTEGRER            ITER
C** NUMBER OF STOP-CRITERION
      INTEGRER            ST_CR
C** CONTROLS OF RCI CYCLE
      INTEGRER            SUCCESSFUL
C** MAXIMUM NUMBER OF ITERATIONS
      INTEGRER            ITER1
C** MAXIMUM NUMBER OF ITERATIONS OF CALCULATION OF TRIAL-STEP
      INTEGRER            ITER2
C** INITIAL STEP BOUND
      DOUBLE PRECISION    RS
C** INITIAL AND FINAL RESIDUALS
      DOUBLE PRECISION    R1, R2
C** TR SOLVER HANDLE
      INTEGRER*8          HANDLE
C** CYCLE'S COUNTERS
      INTEGRER            I, J
C** SET PRECISIONS FOR STOP-CRITERIA
      EPS (1:6) = 1.D-5
C** SET MAXIMUM NUMBER OF ITERATIONS
      ITER1 = 1000
C** SET MAXIMUM NUMBER OF ITERATIONS OF CALCULATION OF TRIAL-STEP
      ITER2 = 100
C** SET INITIAL STEP BOUND
      RS = 100.D0
```

C** PRECISIONS FOR JACOBI CALCULATION

JAC_EPS = 1.D-8

C** SET THE INITIAL GUESS

DO I = 1, N/4

X (4*I - 3) = 3.D0

X (4*I - 2) = -1.D0

X (4*I - 1) = 0.D0

X (4*I) = 1.D0

ENDDO

C** SET LOWER AND UPPER BOUNDS

DO I = 1, N/4

LW(4*I-3) = 0.1D0

LW(4*I-2) = -20.D0

LW(4*I-1) = -1.D0

LW(4*I) = -1.D0

UP(4*I-3) = 100.D0

UP(4*I-2) = 20.D0

UP(4*I-1) = 1.D0

UP(4*I) = 50.D0

ENDDO

C** SET INITIAL VALUES

DO I = 1, M

FVEC (I) = 0.D0

DO J = 1, N

FJAC (I, J) = 0.D0

ENDDO

ENDDO

```

C** INITIALIZE SOLVER (ALLOCATE MEMORY, SET INITIAL VALUES)

C**  HANDLE          IN/OUT: TR SOLVER HANDLE

C**  N               IN:    NUMBER OF FUNCTION VARIABLES

C**  M               IN:    DIMENSION OF FUNCTION VALUE

C**  X               IN:    SOLUTION VECTOR. CONTAINS VALUES X FOR F(X)

C**  LW              IN:    LOWER BOUND

C**  UP              IN:    UPPER BOUND

C**  EPS             IN:    PRECISIONS FOR STOP-CRITERIA

C**  ITER1           IN:    MAXIMUM NUMBER OF ITERATIONS

C**  ITER2           IN:    MAXIMUM NUMBER OF ITERATIONS OF CALCULATION OF TRIAL-STEP

C**  RS              IN:    INITIAL STEP BOUND

      IF (DTRNLSPBC_INIT (HANDLE, N, M, X, LW, UP, EPS, ITER1, ITER2
+    , RS) /= TR_SUCCESS) THEN

C** IF FUNCTION DOES NOT COMPLETE SUCCESSFULLY THEN PRINT ERROR MESSAGE

      PRINT *, '| ERROR IN DTRNLSPBC_INIT'

C** AND STOP

      STOP

      ENDIF

C** SET INITIAL RCI CYCLE VARIABLES

      RCI_REQUEST = 0

      SUCCESSFUL = 0

C** RCI CYCLE

      DO WHILE (SUCCESSFUL == 0)

C** CALL TR SOLVER

C**  HANDLE          IN/OUT: TR SOLVER HANDLE

C**  FVEC            IN:    VECTOR

C**  FJAC            IN:    JACOBI MATRIX

C**  RCI_REQUEST      IN/OUT: RETURN NUMBER THAT DENOTES NEXT STEP FOR PERFORMING

```

```

        IF (DTRNLSPBC_SOLVE (HANDLE, FVEC, FJAC, RCI_REQUEST)
+      /= TR_SUCCESS) THEN
C** IF FUNCTION DOES NOT COMPLETE SUCCESSFULLY THEN PRINT ERROR MESSAGE
        PRINT *, '| ERROR IN DTRNLSPBC_SOLVE'
C** AND STOP
        STOP
    ENDIF
C**  RCI_REQUEST      IN/OUT: RETURN NUMBER THAT DENOTES NEXT STEP FOR PERFORMING
C** ACCORDING TO RCI_REQUEST VALUE WE DO NEXT STEP
        SELECT CASE (RCI_REQUEST)
        CASE (-1, -2, -3, -4, -5, -6)
C**  STOP RCI CYCLE
            SUCCESSFUL = 1
            CASE (1)
C**  RECALCULATE FUNCTION VALUE
C**      M              IN:      DIMENSION OF FUNCTION VALUE
C**      N              IN:      NUMBER OF FUNCTION VARIABLES
C**      X              IN:      SOLUTION VECTOR
C**      FVEC           OUT:      FUNCTION VALUE F(X)
            CALL EXTENDET_POWELL (M, N, X, FVEC)
            CASE (2)
C**  COMPUTE JACOBI MATRIX
C**      EXTENDET_POWELL IN:      EXTERNAL OBJECTIVE FUNCTION
C**      N              IN:      NUMBER OF FUNCTION VARIABLES
C**      M              IN:      DIMENSION OF FUNCTION VALUE
C**      FJAC           OUT:      JACOBI MATRIX
C**      X              IN:      SOLUTION VECTOR
C**      JAC_EPS        IN:      JACOBI CALCULATION PRECISION

```

```

                IF (DJACOBI (EXTENDET_POWELL, N, M, FJAC, X, JAC_EPS)
+
                /= TR_SUCCESS) THEN
C** IF FUNCTION DOES NOT COMPLETE SUCCESSFULLY THEN PRINT ERROR MESSAGE
                PRINT *, '| ERROR IN DTRNLSPBC_SOLVE'
C** AND STOP

                STOP

            ENDIF

        ENDSELECT

    ENDDO

C** GET SOLUTION STATUSES
C**  HANDLE      IN:  TR SOLVER HANDLE
C**  ITER        OUT:  NUMBER OF ITERATIONS
C**  ST_CR       OUT:  NUMBER OF STOP CRITERION
C**  R1          OUT:  INITIAL RESIDUALS
C**  R2          OUT:  FINAL RESIDUALS

                IF (DTRNLSPBC_GET (HANDLE, ITER, ST_CR, R1_R2)
+
                /= TR_SUCCESS) THEN
C** IF FUNCTION DOES NOT COMPLETE SUCCESSFULLY THEN PRINT ERROR MESSAGE
                PRINT *, '| ERROR IN DTRNLSPBC_GET'
C** AND STOP

                STOP

            ENDIF

C** FREE HANDLE MEMORY

                IF (DTRNLSPBC_DELETE (HANDLE) /= TR_SUCCESS) THEN
C** IF FUNCTION DOES NOT COMPLETE SUCCESSFULLY THEN PRINT ERROR MESSAGE
                PRINT *, '| ERROR IN DTRNLSPBC_DELETE'
C** AND STOP

                STOP

```

```

ENDIF

C** IF FINAL RESIDUAL IS LESS THAN REQUIRED PRECISION THEN PRINT PASS
      IF (R2 < 1.D-1) THEN
        PRINT *, '|', DTRNLSPBC POWELL.....PASS'!, R1, R2
C** ELSE PRINT FAILED
      ELSE
        PRINT *, '|', DTRNLSPBC POWELL.....FAILED'!, R1, R2
      ENDIF

END PROGRAM EXAMPLE_DTRNLSPBC_POWELL

C** ROUTINE FOR EXTENDET POWELL FUNCTION CALCULATION
C**  M      IN:      DIMENSION OF FUNCTION VALUE
C**  N      IN:      NUMBER OF FUNCTION VARIABLES
C**  X      IN:      VECTOR FOR FUNCTION CALCULATION
C**  F      OUT:     FUNCTION VALUE F(X)
SUBROUTINE EXTENDET_POWELL (M, N, X, F)
  IMPLICIT NONE
  INTEGER M, N
  DOUBLE PRECISION X (*), F (*)
  INTEGER I

  DO I = 1, N/4
    F (4*I-3) = X(4*I - 3) + 10.D0 * X(4*I - 2)
    F (4*I-2) = 2.2360679774997896964091736687313D0*(X(4*I-1) - X(4*I))
    F (4*I-1) = (X(4*I-2) - 2.D0*X(4*I-1))**2
    F (4*I)   = 3.1622776601683793319988935444327D0*(X(4*I-3) - X(4*I))**2
  
```

ENDDO

ENDSUBROUTINE EXTENDET_POWELL

Example 14-4. dtrnlspsc Usage in C

```
#include <stdio.h>

#include <malloc.h>

#include <math.h>

#include "mkl_rci.h"

/* nonlinear least square problem with boundary constraints */
int main ()
{
    /* user's objective function */
    extern void extendet_powell (int *, int *, double*, double*);
    /* n - number of function variables
       m - dimension of function value */
    int          n = 4, m = 4;
    /* precisions for stop-criteria (see manual for more details) */
    double          eps[6];
    /* solution vector. contains values x for f(x) */
    double          *x;
    /* iter1 - maximum number of iterations
       iter2 - maximum number of iterations of calculation of trial-step */
    int          iter1 = 1000, iter2 = 100;
    /* initial step bound */
    double          rs = 0.0;
    /* reverse communication interface parameter */
    int          RCI_Request;
```

```
/* controls of rci cycle */
int          successful;
/* function (f(x)) value vector */
double       *fvec;
/* jacobi matrix */
double       *fjac;
/* lower and upper bounds */
double       *LW, *UP;
/* number of iterations */
int          iter;
/* number of stop-criterion */
int          st_cr;
/* initial and final residuals */
double       r1, r2;
/* TR solver handle */
_TRNSPBC_HANDLE_t handle;
/* cycle's counter */
int i;

/* memory allocation */
x = (double*) malloc (sizeof (double)*n);
fvec = (double*) malloc (sizeof (double)*m);
fjac = (double*) malloc (sizeof (double)*m*n);
LW = (double*) malloc (sizeof (double)*n);
UP = (double*) malloc (sizeof (double)*n);
/* set precisions for stop-criteria */
for (i = 0; i < 6; i++)
{
```

```

        eps [i] = 0.00001;
    }
    /* set the initial guess */
    for (i = 0; i < n/4; i++)
    {
        x [4*i]      = 3.0;
        x [4*i + 1] = -1.0;
        x [4*i + 2] = 0.0;
        x [4*i + 3] = 1.0;
    }
    /* set the initial values */
    for (i = 0; i < m; i++)
        fvec [i] = 0.0;
    for (i = 0; i < m*n; i++)
        fjac [i] = 0.0;
    /* set bounds */
    for (i = 0; i < n/4; i++)
    {
        LW [4*i]      = 0.1;
        LW [4*i + 1] = -20.0;
        LW [4*i + 2] = -1.0;
        LW [4*i + 3] = -1.0;
        UP [4*i]      = 100.0;
        UP [4*i + 1] = 20.0;
        UP [4*i + 2] = 1.0;
        UP [4*i + 3] = 50.0;
    }
    /* initialize solver (allocate memory, set initial values)

```

```
    handle  in/out: TR solver handle
    n       in:     number of function variables
    m       in:     dimension of function value
    x       in:     solution vector. contains values x for f(x)
    LW      in:     lower bound
    UP      in:     upper bound
    eps     in:     precisions for stop-criteria
    iter1   in:     maximum number of iterations
    iter2   in:     maximum number of iterations of calculation of trial-step
    rs      in:     initial step bound */
if (dtrnlspbc_init (&handle, &n, &m, x, LW, UP, eps, &iter1, &iter2, &rs) != TR_SUCCESS)
{
    /* if function does not complete successfully then print error message */
    printf ("| error in dtrnlspbc_init\n");
    /* and exit */
    return 0;
}
/* set initial rci cycle variables */
RCI_Request = 0;
successful = 0;
/* rci cycle */
while (successful == 0)
{
    /* call tr solver
    handle      in/out: tr solver handle
    fvec        in:     vector
    fjac        in:     jacobi matrix
    RCI_request in/out: return number which denotes next step for performing */
```

```

if (dtrnlsbbc_solve (&handle, fvec, fjac, &RCI_Request) != TR_SUCCESS)
{
    /* if function does not complete successfully then print error message */
    printf ("| error in dtrnlsbbc_solve\n");
    /* and exit */
    return 0;
}

/* according to rci_request value we do next step */
if (RCI_Request == -1 ||
    RCI_Request == -2 ||
    RCI_Request == -3 ||
    RCI_Request == -4 ||
    RCI_Request == -5 ||
    RCI_Request == -6)
    /* exit rci cycle */
    successful = 1;
if (RCI_Request == 1)
{
    /* recalculate function value
       m      in:    dimension of function value
       n      in:    number of function variables
       x      in:    solution vector
       fvec   out:   function value f(x) */
    extendet_powell (&m, &n, x, fvec);
}
if (RCI_Request == 2)
{
    /* compute jacobi matrix

```

```

        extendet_powell in:    external objective function
        n              in:    number of function variables
        m              in:    dimension of function value
        fjac           out:    jacobi matrix
        x              in:    solution vector
        jac_eps        in:    jacobi calculation precision */
if (djacobi (extendet_powell, &n, &m, fjac, x, eps) != TR_SUCCESS)
{
    /* if function does not complete successfully then print error message */
    printf ("| error in djacobi\n");
    /* and exit */
    return 0;
}
}

/* get solution statuses
    handle          in:    TR solver handle
    iter            out:    number of iterations
    st_cr           out:    number of stop criterion
    r1              out:    initial residuals
    r2              out:    final residuals */
if (dtrnlspbc_get (&handle, &iter, &st_cr, &r1, &r2) != TR_SUCCESS)
{
    /* if function does not complete successfully then print error message */
    printf ("| error in dtrnlspbc_get\n");
    /* and exit */
    return 0;
}

```

```

/* free handle memory */
if (dtrnlspbc_delete (&handle) != TR_SUCCESS)
{
    /* if function does not complete successfully then print error message */
    printf ("| error in dtrnlspbc_delete\n");
    /* and exit */
    return 0;
}

/* free allocated memory */
free (x);
free (fvec);
free (fjac);
free (LW);
free (UP);

/* if final residual is less than required precision then print pass */
if (r2 < 0.1)
    printf ("|          dtrnlspbc powell.....PASS\n");
/* else print failed */
else
    printf ("|          dtrnlspbc powell.....FAILED\n");
return 0;
}

/* nonlinear system equations with constraints */
/* routine for extendet powell function calculation
m    in:    dimension of function value
n    in:    number of function variables
x    in:    vector for function calculation

```



```
f      out:      function value f(x) */
void extendet_powell (int *m, int *n, double *x, double *f)
{
    int i;
    for (i = 0; i < (*n)/4; i++)
    {
        f [4*i] = x [4*i] + 10.0*x [4*i + 1];
        f [4*i + 1] = 2.2360679774997896964091736687313*(x [4*i + 2] - x [4*i + 3]);
        f [4*i + 2] = (x [4*i + 1] - 2.0*x [4*i + 2])*(x [4*i + 1] - 2.0*x [4*i + 2]);
        f [4*i + 3] = 3.1622776601683793319988935444327*(x [4*i] - x [4*i + 3])*(x [4*i] - x [4*i
+ 3]);
    }
    return;
}
```

Jacobi Matrix Calculation Routines

This section describes routines that compute the Jacobi matrix using central differences. Jacobi matrix calculation is required to solve nonlinear least squares problem and systems of nonlinear equations (with or without linear bound constraints). Routines for calculation of the Jacobi matrix have "Black-Box" interfaces, where users pass the objective function via parameters. Note that in this case the user's objective function must have fixed interface.

Table 14-3 Jacobi Matrix Calculation Routines

Routine Name	Operation
djacobi_init	Initializes the solver.
djacobi_solve	Computes the Jacobi matrix of the function on the basis of RCI using the central difference.
djacobi_delete	Removes data.
djacobi	Computes the Jacobi matrix of the fcn function using the central difference.

djacobi_init

Initializes the solver for Jacobian calculations.

Syntax

Fortran:

```
res = djacobi_init(handle, n, m, x, a, esp)
```

C:

```
res = djacobi_init(&handle, &n, &m, x, a, &eps);
```

Description

The routine initializes the solver.

Input Parameters

<i>n</i>	INTEGER. Length of <i>x</i> .
<i>m</i>	INTEGER. Length of <i>F</i> .
<i>x</i>	DOUBLE PRECISION. Array of size <i>n</i> . Vector, at which the function is evaluated.
<i>eps</i>	DOUBLE PRECISION. Precision of the Jacobi matrix calculation.

Output Parameters

<i>handle</i>	Data object of the <code>_JACOBI_MATRIX_HANDLE_t</code> type in C/C++ and <code>INTEGER*8</code> in FORTRAN.
<i>a</i>	DOUBLE PRECISION. Array of size (m, n) . Contains the Jacobi matrix of the function.
<i>res</i>	INTEGER. Informs about the task completion. <i>res</i> = <code>TR_SUCCESS</code> indicates that the routine completed the task normally. <i>res</i> = <code>TR_INVALID_OPTION</code> indicates an error in the input parameters. <i>res</i> = <code>TR_OUT_OF_MEMORY</code> indicates a memory error.

djacobi_solve

Computes the Jacobi matrix of the function on the basis of RCI using the central difference.

Syntax

Fortran:

```
res = djacobi_solve(handle, f1, f2, RCI_Request)
```

C:

```
res = djacobi_solve(&handle, f1, f2, &RCI_Request);
```

Description

The `djacobi_solve` routine computes the Jacobi matrix of the function on the basis of RCI using the central difference.

Input Parameters

handle Data object of the `_JACOBI_MATRIX_HANDLE_t` type in C/C++ and `INTEGER*8` in FORTRAN.

Output Parameters

f1 DOUBLE PRECISION. Array of size *m*. Contains the updated function values at $X + \epsilon$.

f2 DOUBLE PRECISION. Array of size *m*. Contains the updated function values at $X - \epsilon$.

RCI_Request INTEGER. Informs about the task completion. When equal to 0, the task is completed successfully.
RCI_Request = 1 indicates the user should calculate the Jacobian matrix and put the result into *f1*.
RCI_Request = 2 indicates the user should calculate the Jacobian matrix and put the result into *f2*.

res INTEGER. Informs about the task completion.
res = `TR_SUCCESS` indicates that the routine has completed the task normally.
res = `TR_INVALID_OPTION` indicates an error in the input parameters.

djacobi_delete

Removes the data object required by the Jacobian calculation.

Syntax

Fortran:

```
res = djacobi_delete(handle)
```

C:

```
res = djacobi_delete(&handle);
```

Description

The routine removes the data object needed for the Jacobi matrix RCI solver.

Input Parameters

<i>handle</i>	Data object of the <code>_JACOBIMATRIX_HANDLE_t</code> type in C/C++ and <code>INTEGER*8</code> in FORTRAN.

Output Parameters

<i>res</i>	<code>INTEGER</code> . Informs about the task completion. <i>res</i> = <code>TR_SUCCESS</code> means the routine completed the task normally.

djacobi

Computes the Jacobi matrix of the user's objective function using the cenral difference.

Syntax

Fortran:

```
res = djacobi(fcn, n, m, fjac, x, jac_eps)
```

C:

```
res = djacobi(fcn, &n, &m, fjac, x, &jac_eps);
```

Description

The routine computes the Jacobi matrix for function *fcn* using the central difference. This routine has "Black-Box" interface, where the user inputs the objective function via parameters. Note that in this case the user's objective function must have a fixed interface.

Input Parameters

<i>fcn</i>	User-supplied subroutine to evaluate the function that defines the least squares problem. Call <i>fcn</i> (<i>m</i> , <i>n</i> , <i>x</i> , <i>f</i>) with the following parameters: <i>m</i> - INTEGER. Input parameter. Length of <i>f</i> <i>n</i> - INTEGER. Input parameter. Length of <i>x</i> . <i>x</i> - DOUBLE PRECISION. Input parameter. Array of size <i>n</i> . Vector, at which the function is evaluated. The <i>fcn</i> function should not change this parameter. <i>f</i> - DOUBLE PRECISION. Output parameter. Array of size <i>m</i> ; contains the function values at <i>x</i> . Declare <i>fcn</i> as EXTERNAL in the calling program.
<i>n</i>	INTEGER. Length of <i>x</i> .
<i>m</i>	INTEGER. Length of <i>F</i> .
<i>x</i>	DOUBLE PRECISION. Array of size <i>n</i> . Vector at which the function is evaluated.
<i>eps</i>	DOUBLE PRECISION. Precision of the Jacobi matrix calculation.

Output Parameters

<i>a</i>	DOUBLE PRECISION. Array of size (<i>m</i> , <i>n</i>). Contains the Jacobi matrix of the function.
<i>res</i>	INTEGER. Informs about the task completion. <i>res</i> = TR_SUCCESS indicates that the routine has completed the task normally. <i>res</i> = TR_INVALID_OPTION indicates an error in the input parameters. <i>res</i> = TR_OUT_OF_MEMORY indicates a memory error.

Examples of djacobi_solve Usage

Example 14-5. djacobi_solve Usage in Fortran

```

PROGRAM JACOBI_MATRIX

    IMPLICIT NONE

    INCLUDE '../include/mkl_opt_tr.f'

    EXTERNAL          EXTENDET_POWELL

C** N - NUMBER OF FUNCTION VARIABLES
C** M - DIMENSION OF FUNCTION VALUE
    INTEGER N, M

    PARAMETER (N = 4, M = 4)

C** JACOBI MATRIX
C** SOLUTION VECTOR. CONTAINS VALUES X FOR F(X)
C** FUNCTION (F(X)) VALUE VECTOR
C** TEMPORARY ARRAYS F1 & F2 WHICH CONTAINS F1 = F(X+EPS) | F2 = F(X-EPS)
    DOUBLE PRECISION A (M,N), X(N), F1(N), F2(N)
    DOUBLE PRECISION EPS

C** PRECISIONS FOR JACOBI_MATRIX CALCULATION
    PARAMETER (EPS = 1.D-6)

C** JACOBI-MATRIX SOLVER HANDLE
    INTEGER*8 HANDLE

C** CONTROLS OF RCI CYCLE
    INTEGER SUCCESSFUL, RCI_REQUEST

C** SET THE X VALUES
    X = 10.d0

```

```
C** INITIALIZE SOLVER (ALLOCATE MEMORY, SET INITIAL VALUES)
      IF (DJACOBI_INIT (HANDLE, N, M, X, EPS) /= TR_SUCCESS) THEN
C** IF FUNCTION DOES NOT COMPLETE SUCCESSFULLY THEN PRINT ERROR MESSAGE
      PRINT *, '#ERROR IN DJACOBI_INIT'
      STOP
    ENDIF
C** SET INITIAL RCI CYCLE VARIABLES
      RCI_REQUEST = 0
      SUCCESSFUL = 0
C** RCI CYCLE
      DO WHILE (SUCCESSFUL == 0)
C** CALL SOLVER
      IF (DJACOBI_SOLVE (HANDLE, F1, F2, RCI_REQUEST) /= TR_SUCCESS) THEN
C** IF FUNCTION DOES NOT COMPLETE SUCCESSFULLY THEN PRINT ERROR MESSAGE
      PRINT *, '#ERROR IN DJACOBI_SOLVE'
      STOP
    ENDIF
      IF      RCI_REQUEST == 1) THEN
C**  CALCULATE FUNCTION VALUE F1 = F(X+EPS)
      CALL EXTENDET_POWELL (M, N, X, F1)
      ELSEIF (RCI_REQUEST == 2) THEN
C**  CALCULATE FUNCTION VALUE F1 = F(X-EPS)
      CALL EXTENDET_POWELL (M, N, X, F2)
      ELSEIF (RCI_REQUEST == 0) THEN
C**  EXIT RCI CYCLE
      SUCCESSFUL = 1
    ENDIF
  ENDDO
```

```

C** FREE HANDLE MEMORY

      IF (DJACOBI_DELETE (HANDLE) /= TR_SUCCESS) THEN
C** IF FUNCTION DOES NOT COMPLETE SUCCESSFULLY THEN PRINT ERROR MESSAGE
      PRINT *, '#ERROR IN DJACOBI_DELETE'
      STOP
    ENDIF

  ENDPROGRAM JACOBI_MATRIX


C** ROUTINE FOR EXTENDET POWELL FUNCTION CALCULATION
C**  M          IN:      DIMENSION OF FUNCTION VALUE
C**  N          IN:      NUMBER OF FUNCTION VARIABLES
C**  X          IN:      VECTOR FOR FUNCTION CALCULATION
C**  F          OUT:     FUNCTION VALUE F(X)

SUBROUTINE EXTENDET_POWELL (M, N, X, F)
  IMPLICIT NONE
  INTEGER M, N
  DOUBLE PRECISION X (*), F (*)
  INTEGER I

  DO I = 1, N/4
    F (4*I-3) = X(4*I - 3) + 10.D0 * X(4*I - 2)
    F (4*I-2) = DSQRT(5.D0) * (X(4*I-1) - X(4*I))
    F (4*I-1) = (X(4*I-2) - 2.D0*X(4*I-1))**2
    F (4*I)   = DSQRT(10.D0)*(X(4*I-3) - X(4*I))**2
  ENDDO

ENDSUBROUTINE EXTENDET_POWELL

```

Example 14-6. djacobi_solve Usage in C

```
#include "../include/mkl_opt_tr.h"

#include <stdlib.h>
#include <stdio.h>

int main ()
{
    /* user's objective function */
    extern void extendet_powell (int*, int*, double*, double*);

    /* n - number of function variables
       m - dimension of function value */
    int    n = 4, m = 4;
    /* jacobi matrix */
    solution vector. contains values x for f(x)
    temporary arrays f1 & f2 that contain f1 = f(x+eps) | f2 = f(x-eps) */
    double *a, *x, *f1, *f2;
    /* precisions for jacobi_matrix calculation */
    double eps = 0.000001;
    /* jacobi-matrix solver handle */
    _JACOBIMATRIX_HANDLE_t handle;
    /* controls of rci cycle */
    int successful, rci_request, i;

    a = (double*) malloc (sizeof (double) * n*m);
    x = (double*) malloc (sizeof (double) * n);
```

```

    f1 = (double*) malloc (sizeof (double) * n);
    f2 = (double*) malloc (sizeof (double) * n);

/* set the x values */
    for (i = 0; i < n; i++) x[i] = 10.0;

/* initialize solver (allocate memory, set initial values) */
    if (djacobi_init (&handle, &n, &m, x, a, &eps) != TR_SUCCESS){
/* if function does not complete successfully then print error message */
        printf ("\n#ERROR IN DJACOBI_INIT\n");
        return 0;
    }

/* set initial rci cycle variables */
    rci_request = 0;
    successful = 0;

/* rci cycle */
    while (successful == 0) {
/* call solver */
        if (djacobi_solve (&handle, f1, f2, &rci_request) != TR_SUCCESS){
/* if function does not complete successfully then print error message */
            printf ("\n#ERROR IN DJACOBI_SOLVE\n");
            return 0;
        }

        if (rci_request == 1)
/* calculate function value f1 = f(x+eps) */
            extendet_powell (&m, &n, x, f1);
        else if (rci_request == 2)
/* calculate function value f1 = f(x-eps) */
            extendet_powell (&m, &n, x, f2);
        else if (rci_request == 0)

```

```
/* exit rci cycle */
    successful = 1;

}

/* free handle memory */
    if (djacobi_delete (&handle) != TR_SUCCESS) {
/* if function does not complete successfully then print error message */
        printf ("\n#ERROR IN DJACOBI_DELETE\n");
        return 0;
    }
    return 0;
}

/* routine for extendet powell function calculation
m      in:      dimension of function value
n      in:      number of function variables
x      in:      vector for function calculation
f      out:     function value f(x) */
void extendet_powell (int *m, int *n, double *x, double *f)
{
    int i;

    for (i = 0; i < (*n)/4; i++)
    {
        f [4*i] = x [4*i] + 10.0*x [4*i + 1];
        f [4*i + 1] = 2.2360679774*(x [4*i + 2] - x [4*i + 3]);
        f [4*i + 2] = (x [4*i + 1] - 2.0*x [4*i + 2])*(x [4*i + 1] - 2.0*x [4*i + 2]);
        f [4*i + 3] = 3.1622776601*(x [4*i] - x [4*i + 3])*(x [4*i] - x [4*i + 3]);
    }
    return;
}
```

}

Examples of djacobi Usage

Example 14-7. djacobi Usage in Fortran

```

C**  COMPUTE JACOBI MATRIX

C**      EXTENDET_POWELL IN:      EXTERNAL OBJECTIVE FUNCTION
C**      N                IN:      NUMBER OF FUNCTION VARIABLES
C**      M                IN:      DIMENSION OF FUNCTION VALUE
C**      FJAC             OUT:     JACOBI MATRIX
C**      X                IN:      SOLUTION VECTOR
C**      JAC_EPS          IN:      JACOBI CALCULATION PRECISION
      IF (DJACOBI (EXTENDET_POWELL, N, M, FJAC, X, JAC_EPS)
+          /= TR_SUCCESS) THEN
C** IF FUNCTION DOES NOT COMPLETE SUCCESSFULLY THEN PRINT ERROR MESSAGE
      PRINT *, '| ERROR IN DJACOBI'
      ENDIF

.....

C** ROUTINE FOR EXTENDET POWELL FUNCTION CALCULATION
C**      M                IN:      DIMENSION OF FUNCTION VALUE
C**      N                IN:      NUMBER OF FUNCTION VARIABLES
C**      X                IN:      VECTOR FOR FUNCTION CALCULATION
C**      F                OUT:     FUNCTION VALUE F(X)
      SUBROUTINE EXTENDET_POWELL (M, N, X, F)
      IMPLICIT NONE
      INTEGER M, N
      DOUBLE PRECISION X (*), F (*)
      INTEGER I

```

```

DO I = 1, N/4
    F (4*I-3) = X(4*I - 3) + 10.D0 * X(4*I - 2)
    F (4*I-2) = DSQRT(5.D0) * (X(4*I-1) - X(4*I))
    F (4*I-1) = (X(4*I-2) - 2.D0*X(4*I-1))**2
    F (4*I)   = DSQRT(10.D0)*(X(4*I-3) - X(4*I))**2
ENDDO

ENDSUBROUTINE EXTENDET_POWELL

```

Example 14-8. djacobi Usage in C

```

/* compute jacobi matrix
    extendet_powell in:    external objective function
    n                in:    number of function variables
    m                in:    dimension of function value
    fjac             out:   jacobi matrix
    x                in:    solution vector
    jac_eps          in:    jacobi calculation precision */

    if (djacobi (extendet_powell, &n, &m, fjac, x, &jac_eps) != TR_SUCCESS){
/* if function does not complete successfully then print error message */
        printf ("\n#ERROR IN DJACOBI\n");
        return 0;
    }

/* .....*/

/* routine for extendet powell function calculation
    m    in:    dimension of function value
    n    in:    number of function variables
    x    in:    vector for function calculation
    f    out:    function value f(x) */
void extendet_powell (int *m, int *n, double *x, double *f)
{
    int i;

    for (i = 0; i < (*n)/4; i++)
    {

```

```
f [4*i] = x [4*i] + 10.0*x [4*i + 1];  
f [4*i + 1] = 2.2360679774*(x [4*i + 2] - x [4*i + 3]);  
f [4*i + 2] = (x [4*i + 1] - 2.0*x [4*i + 2])*(x [4*i + 1] - 2.0*x [4*i + 2]);  
f [4*i + 3] = 3.1622776601*(x [4*i] - x [4*i + 3])*(x [4*i] - x [4*i + 3]);  
}  
return;  
}
```


Support Functions

Intel® MKL support functions are used to:

- retrieve information about the current Intel MKL version
- additionally control the number of threads
- handle errors
- test characters and character strings for equality
- measure user time for a process and elapsed CPU time
- set and measure CPU frequency
- free memory allocated by Intel MKL memory management software

Functions described below are subdivided according to their purpose into the following groups:

[Version Information Functions](#)

[Threading Control Functions](#)

[Error Handling Functions](#)

[Equality Test Functions](#)

[Timing Functions](#)

[Memory Functions](#)

Table 15-1 contains the list of support functions common for Intel MKL.

Table 15-1 Intel MKL Support Functions

Function Name	Operation
Version Information Functions	
<code>MKLGetVersion</code>	Returns information about the active library version.
<code>MKLGetVersionString</code>	Returns information about the library version string.
Threading Control Functions	
<code>mkl_set_num_threads</code>	Suggests the number of threads to use.
<code>mkl_domain_set_num_threads</code>	Suggests the number of threads for a particular function domain.
<code>mkl_set_dynamic</code>	Enables Intel MKL to dynamically change the number of threads.

Function Name	Operation
<code>mkl_get_max_threads</code>	Inquires about the number of threads targeted for parallelism.
<code>mkl_domain_get_max_threads</code>	Inquires about the number of threads targeted for parallelism in different domains.
<code>mkl_get_dynamic</code>	Returns the current value of the <code>MKL_DYNAMIC</code> variable.
Error Handling Functions	
<code>xerbla</code>	Handles error conditions for the BLAS, LAPACK, VML routines.
<code>pxerbla</code>	Handles error conditions for the ScaLAPACK routines.
Equality Test Functions	
<code>lsame</code>	Tests two characters for equality regardless of the case.
<code>lsamen</code>	Tests two character strings for equality regardless of the case.
Timing Functions	
<code>second/dsecnd</code>	Returns user time for a process.
<code>getcpuclocks</code>	Returns full precision elapsed CPU clocks.
<code>getcpufrequency</code>	Returns CPU frequency value in GHz.
<code>setcpufrequency</code>	Sets CPU frequency value in GHz.
Memory Functions	
<code>MKL_FreeBuffers</code>	Frees memory buffers.
<code>MKL_MemStat</code>	Reports an amount of memory utilized by MKL Memory Manager.
<code>MKL_malloc</code>	Allocates the aligned memory buffer.
<code>MKL_free</code>	Frees the aligned memory buffer allocated by <code>MKL_malloc</code> .

Version Information Functions

Intel® MKL provides two methods for extracting information about the library version number:

- extracting a version string using the `MKLGetString` function
- using the `MKLGetVersion` function to obtain an `MKLVersion` structure that contains the version information

A makefile is also provided to automatically build the examples and output summary files containing the version information for the current library.

MKLGetVersion

Returns information about the active library C version.

Syntax

```
void MKLGetVersion( MKLVersion* pVersion );
```

Output Parameters

`pVersion` Pointer to the `MKLVersion` structure.

Description

The `MKLGetVersion` function collects information about the active C version of the Intel MKL software and returns this information in a structure of `MKLVersion` type by the `pVersion` address. The `MKLVersion` structure type is defined in the `mkl_types.h` file. The following fields of the `MKLVersion` structure are available:

<code>MajorVersion</code>	is the major number of the current library version.
<code>MinorVersion</code>	is the minor number of the current library version.
<code>BuildNumber</code>	is the update number of the current library version.
<code>ProductStatus</code>	is the status of the current library version. Possible variants could be "Beta", "Product".
<code>Build</code>	is the string that contains the build date and the internal build number.

Processor is the processor optimization that is targeted for the specific processor. It is not the definition of the processor installed in the system, rather the MKL library detection that is optimal for the processor installed in the system.

MKLGetVersion Usage

```
#include <stdio.h>
#include <stdlib.h>
#include "mkl_blas.h"
#include "mkl_types.h"

int main(void)
{
    MKLVersion Version;
    MKLGetVersion(&Version);

    printf("Major version:          %d\n", Version.MajorVersion);
    printf("Minor version:            %d\n", Version.MinorVersion);
    printf("Update number:              %d\n", Version.BuildNumber);
    printf("Product status:             %s\n", Version.ProductStatus);
    printf("Build:                      %s\n", Version.Build);
    printf("Processor optimization:    %s\n", Version.Processor);
    printf("=====\n");
    printf("\n");
    return 0;
}
```

Output:

Major Version	9
Minor Version	0

Build number	0
Product status	Product
Build	061909.09
Processor optimization	Intel® Xeon® Processor with Intel® 64 architecture

MKLGetVersionString

Gets the library version string.

Syntax

Fortran:

```
call MKLGetVersionString( buf )
```

C:

```
MKLGetVersionString( char* buf, int len );
```

Output Parameters

Name	Type	Description
<i>buf</i>	FORTTRAN: CHARACTER*198 C: char*	Source string
<i>len</i>	FORTTRAN: INTEGER C: int	Length of the source string

Description

This function returns a string that contains the library version information.
See example below:

Examples

Fortran:

```
program getversionstring

character*198 buf

call mklgetversionstring(buf)
write(*,'(a)') buf

end
```

C:

```
#include <stdio.h>
#include "mkl_blas.h"

int main(void)
{
    int len=198;
    char buf[198];
    MKLGetVersionString(buf, len);
    printf("%s\n",buf);
    printf("\n");
    return 0;
}
```

Threading Control Functions

Intel® MKL provides optional threading control via service functions that take precedence over setting new environment variables with the same purpose (see *Intel® MKL User's Guide* for details).

These functions enable you to specify the number of threads for Intel MKL independently of the OpenMP* settings and prevailing over OpenMP* equivalents. Although Intel MKL may actually use the number of threads that differs from the one suggested, the controls will also enable you to instruct the library to try using the suggested number in the event of undetectable threading behavior in the application calling the library.

See the following examples of Fortran and C usage:

Fortran Usage

```
call mkl_set_num_threads( foo )
ret = mkl_domain_set_num_threads( num, MKL_BLAS )
call mkl_set_dynamic ( 1 )
num = mkl_get_max_threads()
num = mkl_domain_get_max_threads( MKL_BLAS );
ret = mkl_get_dynamic()
```

C Usage

```
#include "mkl.h" // Mandatory to make these definitions work!
mkl_set_num_threads(num); // Equivalent to omp_set_num_threads( int )
return_code = mkl_domain_set_num_threads( num, MKL_FFT );
mkl_set_dynamic( 1 );
num = mkl_get_max_threads();
num = mkl_domain_get_max_threads( MKL_FFT );
return_code = mkl_get_dynamic();
```



NOTE. Always remember to add `#include "mkl.h"` to use the C usage syntax.

mkl_set_num_threads

Suggests the number of threads to use.

Syntax

Fortran:

```
call mkl_set_num_threads( number )
```

C:

```
void mkl_set_num_threads( int number );
```

Input Parameters

Name	Type	Description
<i>number</i>	FORTTRAN: INTEGER C: int	Number of threads suggested by user

Description

This function allows the user to request independently of OpenMP* how many threads MKL should use. This is just a hint, and it is not guaranteed that this number of threads will be used. The user must enter a positive integer. This routine takes precedence over the `MKL_NUM_THREADS` environment variable .

See *Intel MKL User's Guide* for implementation details.

mkl_domain_set_num_threads

Suggests the number of threads for a particular function domain.

Syntax

Fortran:

```
call integer mkl_domain_set_num_threads( num, mask )
```

C:

```
int mkl_domain_set_num_threads( int num, int mask );
```


Input Parameters

Name	Type	Description
<i>num</i>	FORTRAN: INTEGER C: int	Number of threads suggested by user
<i>mask</i>	FORTRAN: INTEGER C: int	Name of the targeted domain

Description

This function allows the user to request different domains of Intel MKL to use different numbers of threads. The currently supported domains are:

- MKL_BLAS - BLAS domain
- MKL_FFT - FFT domain (excluding cluster FFT)
- MKL_VML - vector math library
- MKL_ALL - another way to do what `mkl_set_num_threads` does

This is just a hint, and it is not guaranteed that this number of threads will be used. The user must enter a valid domain and a positive integer for the number of threads. This routine has precedence over the `MKL_DOMAIN_NUM_THREADS` environment variable.

See *Intel MKL User’s Guide* for implementation details.

Return Values

1 (true)	Indicates no error, execution is successful.
0 (false)	Indicates failure, possibly because the inputs were invalid.

mkl_set_dynamic

Enables Intel MKL to dynamically change the number of threads.

Syntax

Fortran:

```
call mkl_set_dynamic( boolean_var )
```

C:

```
void mkl_set_dynamic( int  boolean_var );
```

Input Parameters

Name	Type	Description
<i>boolean_var</i>	FORTRAN: INTEGER C: int	The parameter that determines whether dynamic adjustment of the number of threads is enabled or disabled.

Description

This function indicates whether or not Intel MKL can dynamically change the number of threads. The default for this is `true`, regardless of how the `OMP_DYNAMIC` variable is set. This will also hold precedent over the `OMP_DYNAMIC` variable.

A value of `false` does not guarantee that the user's requested number of threads will be used. But it means that MKL will attempt to use that value. This routine takes precedence over the environment variable `MKL_DYNAMIC`.

Note that if MKL is called from within a parallel region, MKL may not thread unless `MKL_DYNAMIC` is set to `false`, either with the environment variable or by this routine call.

See *Intel MKL User's Guide* for implementation details.

mkl_get_max_threads

Inquires about the number of threads targeted for parallelism.

Syntax

Fortran:

```
num = mkl_get_max_threads()
```

C:

```
int mkl_get_max_threads();
```

Description

This function allows the user to inquire independently of OpenMP* how many threads Intel MKL is targeting for parallelism. This is just a hint, and it is not guaranteed that this number of threads will be used.

See *Intel MKL User's Guide* for implementation details.

Return Values

The output is `INTEGER` equal to the number of threads.

`mkl_domain_get_max_threads`

Inquires about the number of threads targeted for parallelism in different domains.

Syntax

Fortran:

```
call integer mkl_domain_get_max_threads( mask )
```

C:

```
int mkl_domain_get_max_threads( int mask );
```

Input Parameters

Name	Type	Description
<i>mask</i>	FORTTRAN: <code>INTEGER</code> C: <code>int</code>	The name of the targeted domain

Description

This function allows the user of different domains of Intel MKL to inquire what number of threads is being used as a hint. The inquiry does not imply that this is the actual number of threads used. The number may vary depending on the value of the `MKL_DYNAMIC` variable and/or problem size, system resources, etc. But the function returns the value that MKL is targeting for the given domain.

The currently supported domains are:

- MKL_BLAS - BLAS domain
- MKL_FFT - FFT domain (excluding cluster FFT)
- MKL_VML - vector math library
- MKL_ALL - another way to do what `mkl_get_max_threads` does.

The user must enter a valid domain.

See *Intel MKL User's Guide* for implementation details.

Return Values

Returns the hint at the number of threads for the given domain.

mkl_get_dynamic

Returns current value of MKL_DYNAMIC variable.

Syntax

Fortran:

```
ret = mkl_get_dynamic()
```

C:

```
int mkl_get_dynamic();
```

Description

This function returns the current value of the `MKL_DYNAMIC` variable. This variable can be changed by manipulating the `MKL_DYNAMIC` environment variable before the MKL run is launched, or by calling `mkl_set_dynamic()`. Doing the latter has precedence over the former.

The function returns a value of 0 or 1. 1 indicates that `MKL_DYNAMIC` is true. 0 indicates that `MKL_DYNAMIC` is false. This variable indicates whether or not Intel MKL can dynamically change the number of threads. A value of `false` does not guarantee that the user's requested number of threads will be used. But it means that Intel MKL will attempt to use that value. This routine takes precedence over the environment variable `MKL_DYNAMIC`.

Note that if MKL is called from within a parallel region, MKL may not thread unless `MKL_DYNAMIC` is set to `false`, either with the environment variable or by this routine call.

See *Intel MKL User's Guide* for implementation details.

Return Values

1	Indicates MKL_DYNAMIC is true.
0	Indicates MKL_DYNAMIC is false.

Error Handling Functions

xerbla

Error handling routine called by BLAS, LAPACK, VML routines.

Syntax

```
call xerbla( sname, info )
```

Input Parameters

<i>sname</i>	CHARACTER*6. The name of the routine which called xerbla.
<i>info</i>	INTEGER. The position of the invalid parameter in the parameter list of the calling routine.

Description

The routine `xerbla` is an error handler for the BLAS, LAPACK, and VML routines. It is called by a BLAS, LAPACK, or VML routine if an input parameter has an invalid value.

A message is printed and execution stops.

Installers may consider modifying the `stop` statement to call system-specific exception-handling facilities.

pxerbla

Error handling routine called by ScaLAPACK routines.

Syntax

```
call pxerbla(ictxt, sname, info)
```

Input Parameters

<i>ictxt</i>	(global) INTEGER The BLACS context handle, indicating the global context of the operation. The context itself is global.
<i>sname</i>	(global) CHARACTER*6 The name of the routine which called <code>pxerbla</code> .
<i>info</i>	(global) INTEGER. The position of the invalid parameter in the parameter list of the calling routine.

Description

This routine is an error handler for the *ScaLAPACK* routines. It is called by a *ScaLAPACK* routine if an input parameter has an invalid value. A message is printed. Program execution is not terminated. For the *ScaLAPACK* driver and computational routines, a `RETURN` statement is issued following the call to `pxerbla`.

Control returns to the higher-level calling routine, and it is left to the user to determine how the program should proceed. However, in the specialized low-level *ScaLAPACK* routines (auxiliary routines that are Level 2 equivalents of computational routines), the call to `pxerbla()` is immediately followed by a call to `BLACS_ABORT()` to terminate program execution since recovery from an error at this level in the computation is not possible.

It is always good practice to check for a nonzero value of *info* on return from a *ScaLAPACK* routine. Installers may consider modifying this routine in order to call system-specific exception-handling facilities.

Equality Test Functions

lsame

Tests two characters for equality regardless of the case.

Syntax

```
val = lsame( ca, cb )
```

Input Parameters

ca, cb CHARACTER*1. Specify the single characters to be compared.

Output Parameters

val LOGICAL. Result of the comparison.

Description

This logical function returns `.TRUE.` if *ca* is the same letter as *cb* regardless of the case.

lsamen

Tests two character strings for equality regardless of the case.

Syntax

```
val = lsamen( n, ca, cb )
```

Input Parameters

n INTEGER. The number of characters in *ca* and *cb* to be compared.

ca, cb CHARACTER*(*). Specify two character strings of length at least *n* to be compared. Only the first *n* characters of each string will be accessed.

Output Parameters

val LOGICAL. Result of the comparison.

Description

This logical function tests if the first *n* letters of the string *ca* are the same as the first *n* letters of *cb*, regardless of case. The function `lsamen` returns `.TRUE.` if *ca* and *cb* are equivalent except for the case and `.FALSE.` otherwise. `lsamen` also returns `.FALSE.` if `len(ca)` or `len(cb)` is less than *n*.

Timing Functions

second/dsecnd

Returns elapsed CPU time in seconds.

Syntax

```
val = second()
```

```
val = dsecnd()
```

Output Parameters

<i>val</i>	REAL for <code>second</code> DOUBLE PRECISION for <code>dsecnd</code> Elapsed CPU time in seconds.

Description

The `second/dsecnd` functions return the elapsed CPU time in seconds. These versions get the time from the elapsed CPU clocks divided by CPU frequency. The difference is that `dsecnd` returns the result with double precision.

The functions should be applied in pairs: the first time, before a routine to be measured, and the second time - after the measurement. The difference between the returned values is the time spent in the routine. The usage of `second` is discouraged for measuring short time intervals because the single precision format is not capable of holding sufficient timer precision.

getcpuclocks

Returns full precision elapsed CPU clocks.

Syntax

```
getcpuclocks( clocks )
```

Output Parameters

<i>clocks</i>	INTEGER*8. Elapsed CPU clocks.

Description

The `getcpuclocks` subroutine returns the elapsed CPU clocks.

This may be useful when timing short intervals with high resolution. The `getcpuclocks` subroutine is also applied in pairs like `second/dsecnd`. Note that out-of-order code execution on IA-32 or Intel® 64 architecture processors may disturb the exact elapsed CPU clocks value a little bit, which may be important while measuring extremely short time intervals.

getcpufrequency

Returns CPU frequency value in GHz.

Syntax

```
val = getcpufrequency()
```

Output Parameters

val DOUBLE PRECISION. CPU frequency value in GHz.

Description

The function `getcpufrequency` returns the CPU frequency in GHz. This value is used by `second/dsecnd` functions while converting CPU clocks into seconds.

Obtaining a frequency may take some time when `second/dsecnd/getcpufrequency` is called for the first time. To avoid it, call `setcpufrequency` before setting the exact CPU frequency if it is known in advance.

setcpufrequency

Sets CPU frequency value in GHz.

Syntax

```
setcpufrequency( freq )
```

Output Parameters

freq DOUBLE PRECISION. CPU frequency value in GHz.

Description

The `setcpufrequency` function sets the CPU frequency in GHz, used then by `second` / `dsecond` functions while converting CPU clocks into seconds. Setting the exact CPU frequency is useful to bypass obtaining a frequency by `getcpufrequency`.

Initially, CPU frequency value is unset. The CPU frequency value can be set only by `setcpufrequency` call, or during the first call of `getcpufrequency`, if `setcpufrequency` has not been called before. Calling `setcpufrequency` with a special parameter `freq = -1.0` forces the CPU frequency value be unset.

Note that the CPU frequency of your machine is not actually changed when the `setcpufrequency` subroutine is called.

Memory Functions

This section describes the Intel MKL memory support functions. See the *Intel® MKL User's Guide* for details of the Intel MKL memory management.

MKL_FreeBuffers

Frees memory buffers.

Syntax

```
void MKL_FreeBuffers(void)
```

Description

The `MKL_FreeBuffers` function frees the memory allocated by the Intel MKL Memory Manager. The Memory Manager allocates new buffers if no free buffers are currently available. Call `MKL_FreeBuffers()` to free all memory buffers and to avoid memory leaking on completion of work with the Intel MKL functions, that is, after the last call of an Intel MKL function from your application.

See *Intel® MKL User's Guide* for details.

MKL_FreeBuffers Usage with FFT Functions

```

-----
DFTI_DESCRIPTOR *hand1, *hand2;
void MKL_FreeBuffers(void);
. . . . .
/* Using MKL FFT */
Status = DftiCreateDescriptor(&hand1, DFTI_SINGLE, DFTI_COMPLEX, dim, m1);
Status = DftiCommitDescriptor(hand1);
Status = DftiComputeForward(hand1, s_array1);
. . . . .
Status = DftiCreateDescriptor(&hand2, DFTI_SINGLE, DFTI_COMPLEX, dim, m2);
Status = DftiCommitDescriptor(hand2);
. . . . .
Status = DftiFreeDescriptor(&hand1);
/* Do not call MKL_FreeBuffers() here as the hand2 descriptor will be destroyed! */
. . . . .
Status = DftiComputeBackward(hand2, s_array2));
Status = DftiFreeDescriptor(&hand2);
/* Here user finishes the MKL FFT usage */
/* Memory leak will be triggered by any memory control tool */
/* Use MKL_FreeBuffers() to avoid memory leaking */
MKL_FreeBuffers();
-----

```

If the memory space is sufficient, use `MKL_FreeBuffers` after the last call of the MKL functions. Otherwise, a drop in performance can occur due to reallocation of buffers for the subsequent MKL functions.



WARNING. For FFT calls, do not use `MKL_FreeBuffers` between `DftiCreateDescriptor(hand)` and `DftiFreeDescriptor(&hand)`.

MKL_MemStat

Reports amount of memory utilized by MKL Memory Manager.

Syntax

Fortran:

```
AllocatedBytes = MKL_MEMSTAT( AllocatedBuffers )
```

C:

```
AllocatedBytes = MKL_MemStat ( int* AllocatedBuffers );
```

Output Parameters

Name	Type	Description
<i>AllocatedBytes</i>	FORTTRAN: INTEGER*8 C: MKL_INT64	Amount of allocated bytes
<i>AllocatedBuffers</i>	FORTTRAN: INTEGER*4, C: int*	Number of allocated buffers

Description

The function returns the amount of the allocated memory in the *AllocatedBuffers* buffers. If there are no allocated buffers at the moment, the function returns 0. Call the `MKL_MemStat()` function to check the Intel MKL memory status.

Note that after calling `MKL_FreeBuffers()` there should not be any allocated buffers.

See [Example “MKL_malloc\(\), MKL_free\(\), MKL_MemStat\(\) Usage”](#).

MKL_malloc

Allocates the aligned memory buffer.

Syntax

Fortran:

```
ptr = MKL_MALLOC ( size, alignment )
```

C:

```
void* MKL_malloc ( size_t size , int alignment );
```

Input Parameters

Name	Type	Description
<i>size</i>	FORTTRAN: INTEGER C: size_t	Size of the buffer to be allocated
<i>alignment</i>	FORTTRAN: INTEGER C: int	Alignment of the allocated buffer

Output Parameters

Name	Type	Description
<i>ptr</i>	FORTTRAN: POINTER C: void*	Pointer to the allocated buffer

Description

The function allocates a *size*-bytes buffer, aligned on the *alignment* boundary, and returns a pointer to this buffer.

The function returns `NULL` if *size* < 1. If alignment is not power of 2, the alignment 32 is used.

See [Example “MKL_malloc\(\), MKL_free\(\), MKL_MemStat\(\) Usage”](#).

MKL_free

Frees the aligned memory buffer allocated by MKL_malloc.

Syntax

Fortran:

```
CALL MKL_FREE ( ptr )
```

C:

```
void* MKL_free ( void* ptr );
```

Input Parameters

Name	Type	Description
<i>ptr</i>	FORTTRAN: POINTER C: void*	Pointer to the buffer to be freed

Description

The function frees the buffer pointed by *ptr* and allocated by `MKL_malloc()`.

See [Example “MKL_malloc\(\), MKL_free\(\), MKL_MemStat\(\) Usage”](#).

Examples of MKL_malloc(), MKL_free(), MKL_MemStat() Usage

Usage Example in Fortran

```
PROGRAM FOO

REAL*8      A,B,C
POINTER      (A_PTR,A(1)), (B_PTR,B(1)), (C_PTR,C(1))
INTEGER      N, I
REAL*8      ALPHA, BETA
INTEGER      ALLOCATED_BYTES
INTEGER      ALLOCATED_BUFFERS
INTEGER      MKL_MEMSTAT, MKL_MALLOC
EXTERNAL     MKL_MALLOC, MKL_FREE, MKL_MEMSTAT

ALPHA = 1.1; BETA = -1.2
N = 1000
A_PTR = MKL_MALLOC(8*N*N,128)
B_PTR = MKL_MALLOC(8*N*N,128)
C_PTR = MKL_MALLOC(8*N*N,128)
DO I=1,N*N
    A(1) = I
    B(1) = -I
    C(1) = 0.0
END DO

CALL DGEMM('N','N',N,N,N,ALPHA,A,N,B,N,BETA,C,N);

ALLOCATED_BYTES = MKL_MEMSTAT(ALLOCATED_BUFFERS)
```

```
PRINT *, 'DGEMM uses ', ALLOCATED_BYTES, ' bytes in ',  
$  ALLOCATED_BUFFERS, ' buffers '  
  
CALL MKL_FREEBUFFERS  
  
ALLOCATED_BYTES = MKL_MEMSTAT(ALLOCATED_BUFFERS)  
IF (ALLOCATED_BYTES > 0) THEN  
    PRINT *, 'MKL MEMORY LEAK!'  
    PRINT *, 'AFTER MKL_FREEBUFFERS there are ',  
$    ALLOCATED_BYTES, ' bytes in ',  
$    ALLOCATED_BUFFERS, ' buffers '  
END IF  
  
CALL MKL_FREE(A_PTR)  
CALL MKL_FREE(B_PTR)  
CALL MKL_FREE(C_PTR)  
  
STOP  
END
```

Usage Example in C

```
#include <stdio.h>

#include <mkl.h>

int main(void) {
    double *a, *b, *c;

    int n, i;

    double alpha, beta;

    MKL_INT64 AllocatedBytes;
    int N_AllocatedBuffers;

    alpha = 1.1; beta = -1.2;
    n = 1000;
    a = (double*)MKL_malloc(n*n*sizeof(double),128);
    b = (double*)MKL_malloc(n*n*sizeof(double),128);
    c = (double*)MKL_malloc(n*n*sizeof(double),128);
    for (i=0;i<(NN*NN);i++) {
        a[i] = (double)(i+1);
        b[i] = (double)(-i-1);
        c[i] = 0.0;
    }

    dgemm('N','N', &n, &n, &n, &alpha, a, &n, b, &n, &beta, c, &n);

    AllocatedBytes = MKL_MemStat(&N_AllocatedBuffers);
    printf('\nDGEMM uses %ld bytes in %d buffers', (long)AllocatedBytes, N_AllocatedBuffers);

    MKL_FreeBuffers();
}
```

```
AllocatedBytes = MKL_MemStat(&N_AllocatedBuffers);  
if (AllocatedBytes > 0) {  
    printf("\nMKL memory leak!");  
    printf("\nAfter MKL_FreeBuffers there are %ld bytes in %d buffers",  
        (long)AllocatedBytes, N_AllocatedBuffers);  
}  
  
MKL_free(a);  
MKL_free(b);  
MKL_free(c);  
  
return 0;  
}
```

BLACS Routines

This chapter describes the Intel® Math Kernel Library implementation of routines from the BLACS (Basic Linear Algebra Communication Subprograms) package that are used to support a linear algebra oriented message passing interface that may be implemented efficiently and uniformly across a large range of distributed memory platforms.

The BLACS routines make linear algebra applications both easier to program and more portable. For this purpose, they are used in Intel MKL intended for the Linux* and Windows* OSs as the communication layer of ScaLAPACK and Cluster FFT.

On computers, a linear algebra matrix is represented by a two dimensional array (2D array), and therefore the BLACS operate on 2D arrays. See description of the basic [matrix shapes](#) in a special section.

The BLACS routines implemented in Intel MKL are of four categories:

- Combines
- Point to Point Communication
- Broadcast
- Support.

The [Combines](#) take data distributed over processes and combine the data to produce a result. The [Point to Point](#) routines are intended for point-to-point communication and [Broadcast](#) routines send data possessed by one process to all processes within a scope.

The [Support routines](#) perform distinct tasks that can be used for initialization, destruction, information, and miscellaneous tasks.

Matrix Shapes

The BLACS routines recognize the two most common classes of matrices for dense linear algebra. The first of these classes consists of general rectangular matrices, which in machine storage are 2D arrays consisting of m rows and n columns, with a leading dimension, lda , that determines the distance between successive columns in memory.

The *general rectangular* matrices take the following parameters as input when determining what array to operate on:

m	(input) INTEGER. The number of matrix rows to be operated on.
n	(input) INTEGER. The number of matrix columns to be operated on.
a	(input/output) TYPE (depends on routine), array of dimension (lda, n) . A pointer to the beginning of the (sub)array to be sent.

lda (input) INTEGER. The distance between two elements in matrix row.

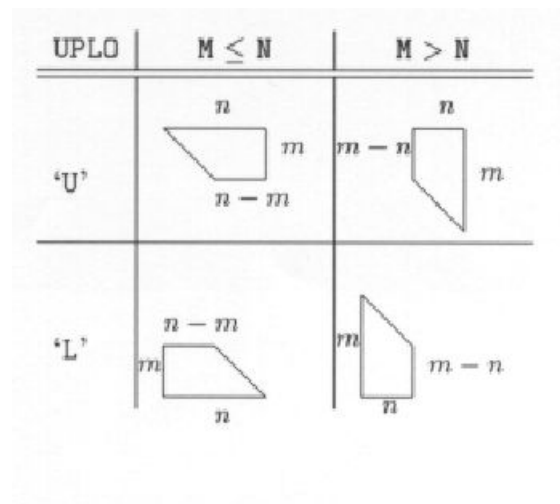
The second class of matrices recognized by the BLACS are *trapezoidal* matrices (triangular matrices are a sub-class of trapezoidal). Trapezoidal arrays are defined by m , n , and lda , as above, but they have two additional parameters as well. These parameters are:

uplo (input) CHARACTER*1 . Indicates whether the matrix is upper or lower trapezoidal, as discussed below.

diag (input) CHARACTER*1 . Indicates whether the diagonal of the matrix is unit diagonal (will not be operated on) or otherwise (will be operated on).

The shape of the trapezoidal arrays is determined by these parameters as follows:

Figure 16-1 Trapezoidal Arrays Shapes



The packing of arrays, if required, so that they may be sent efficiently is hidden, allowing the user to concentrate on the logical matrix, rather than on how the data is organized in the system memory.

BLACS Combine Operations

This section describes BLACS routines that combine the data to produce a result.

In a combine operation, each participating process contributes data that is combined with other processes' data to produce a result. This result can be given to a particular process (called the *destination* process), or to all participating processes. If the result is given to only one process, the operation is referred to as a *leave-on-one* combine, and if the result is given to all participating processes the operation is referenced as a *leave-on-all* combine.

At present, three kinds of combines are supported. They are:

- element-wise summation
- element-wise absolute value maximization
- element-wise absolute value minimization

of general rectangular arrays.

Note that a combine operation combines data between processes. By definition, a combine performed across a scope of only one process does not change the input data. This is why the operations (`max/min/sum`) are specified as *element-wise*. Element-wise indicates that each element of the input array will be combined with the corresponding element from all other processes' arrays to produce the result. Thus, a 4 x 2 array of inputs produces a 4 x 2 answer array.

When the `max/min` comparison is being performed, absolute value is used. For example, -5 and 5 are equivalent. However, the returned value is unchanged; that is, it is not the absolute value, but is a signed value instead. Therefore, if you performed a BLACS absolute value maximum combine on the numbers -5, 3, 1, 8 the result would be -8.

The initial symbol ? in the routine names below masks the data type:

i	integer
s	single precision real
d	double precision real
c	single precision complex
z	double precision complex.

Table 16-1 BLACS Combines

Routine name	Results of operation
?gamx2d	Entries of result matrix will have the value of the greatest absolute value found in that position.
?gamn2d	Entries of result matrix will have the value of the smallest absolute value found in that position.
?gsum2d	Entries of result matrix will have the summation of that position.

?gamx2d

Performs element-wise absolute value maximization.

Syntax

```
call igamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest
)
call sgamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest
)
call dgamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest
)
call cgamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest
)
call zgamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest
)
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>scope</i>	CHARACTER*1. Indicates what scope the combine should proceed on. Limited to ROW, COLUMN, or ALL.
<i>top</i>	CHARACTER*1. Communication pattern to use during the combine operation.
<i>m</i>	INTEGER. The number of matrix rows to be combined.
<i>n</i>	INTEGER. The number of matrix columns to be combined.
<i>a</i>	TYPE array (lda, n). Matrix to be compared with to produce the maximum.
<i>lda</i>	INTEGER. The leading dimension of the matrix A, that is, the distance between two successive elements in a matrix row.
<i>rcflag</i>	INTEGER. If <i>rcflag</i> = -1, the arrays <i>ra</i> and <i>ca</i> are not referenced and need not exist. Otherwise, <i>rcflag</i> indicates the leading dimension of these arrays, and so must be $\geq m$.

<i>rdest</i>	INTEGER. The process row coordinate of the process that should receive the result. If <i>rdest</i> or <i>cdest</i> = -1, all processes within the indicated scope receive the answer.
<i>cdest</i>	INTEGER. The process column coordinate of the process that should receive the result. If <i>rdest</i> or <i>cdest</i> = -1, all processes within the indicated scope receive the answer.

Output Parameters

<i>a</i>	TYPE array (<i>lda</i>, <i>n</i>). Contains the result if this process is selected to receive the answer, or intermediate results if the process is not selected to receive the result.
<i>ra</i>	INTEGER array (<i>rcflag</i>, <i>n</i>). If <i>rcflag</i> = -1, this array will not be referenced, and need not exist. Otherwise, it is an integer array (of size at least <i>rcflag</i> × <i>n</i>) indicating the row index of the process that provided the maximum. If the calling process is not selected to receive the result, this array will contain intermediate (useless) results.
<i>ca</i>	INTEGER array (<i>rcflag</i>, <i>n</i>). If <i>rcflag</i> = -1, this array will not be referenced, and need not exist. Otherwise, it is an integer array (of size at least <i>rcflag</i> × <i>n</i>) indicating the row index of the process that provided the maximum. If the calling process is not selected to receive the result, this array will contain intermediate (useless) results.

Description

This routine performs element-wise absolute value maximization, that is, each element of matrix *A* is compared with the corresponding element of the other process's matrices. Note that the value of *A* is returned, but the absolute value is used to determine the maximum (the 1-norm is used for complex numbers). Combines may be globally-blocking, so they must be programmed as if no process returns until all have called the routine.

?gamn2d

Performs element-wise absolute value minimization.

Syntax

```
call igamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest
)
call sgamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest
)
call dgamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest
)
call cgamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest
)
call zgamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest
)
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>scope</i>	CHARACTER*1. Indicates what scope the combine should proceed on. Limited to ROW, COLUMN, or ALL.
<i>top</i>	CHARACTER*1. Communication pattern to use during the combine operation.
<i>m</i>	INTEGER. The number of matrix rows to be combined.
<i>n</i>	INTEGER. The number of matrix columns to be combined.
<i>a</i>	TYPE array (<i>lda</i> , <i>n</i>). Matrix to be compared with to produce the minimum.
<i>lda</i>	INTEGER. The leading dimension of the matrix <i>A</i> , that is, the distance between two successive elements in a matrix row.
<i>rcflag</i>	INTEGER. If <i>rcflag</i> = -1, the arrays <i>ra</i> and <i>ca</i> are not referenced and need not exist. Otherwise, <i>rcflag</i> indicates the leading dimension of these arrays, and so must be $\geq m$.

<i>rdest</i>	INTEGER. The process row coordinate of the process that should receive the result. If <i>rdest</i> or <i>cdest</i> = -1, all processes within the indicated scope receive the answer.
<i>cdest</i>	INTEGER. The process column coordinate of the process that should receive the result. If <i>rdest</i> or <i>cdest</i> = -1, all processes within the indicated scope receive the answer.

Output Parameters

<i>a</i>	TYPE array (<i>lda</i>, <i>n</i>). Contains the result if this process is selected to receive the answer, or intermediate results if the process is not selected to receive the result.
<i>ra</i>	INTEGER array (<i>rcflag</i>, <i>n</i>). If <i>rcflag</i> = -1, this array will not be referenced, and need not exist. Otherwise, it is an integer array (of size at least <i>rcflag</i> × <i>n</i>) indicating the row index of the process that provided the minimum. If the calling process is not selected to receive the result, this array will contain intermediate (useless) results.
<i>ca</i>	INTEGER array (<i>rcflag</i>, <i>n</i>). If <i>rcflag</i> = -1, this array will not be referenced, and need not exist. Otherwise, it is an integer array (of size at least <i>rcflag</i> × <i>n</i>) indicating the row index of the process that provided the minimum. If the calling process is not selected to receive the result, this array will contain intermediate (useless) results.

Description

This routine performs element-wise absolute value minimization, that is, each element of matrix *A* is compared with the corresponding element of the other process's matrices. Note that the value of *A* is returned, but the absolute value is used to determine the minimum (the 1-norm is used for complex numbers). Combines may be globally-blocking, so they must be programmed as if no process returns until all have called the routine.

?gsum2d

Performs element-wise summation.

Syntax

```
call igsum2d(icontxt, scope, top, m, n, a, lda, rdest, cdest)
call sgsum2d(icontxt, scope, top, m, n, a, lda, rdest, cdest)
call dgsum2d(icontxt, scope, top, m, n, a, lda, rdest, cdest)
call cgsum2d(icontxt, scope, top, m, n, a, lda, rdest, cdest)
call zgsum2d(icontxt, scope, top, m, n, a, lda, rdest, cdest)
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>scope</i>	CHARACTER*1. Indicates what scope the combine should proceed on. Limited to ROW, COLUMN, or ALL.
<i>top</i>	CHARACTER*1. Communication pattern to use during the combine operation.
<i>m</i>	INTEGER. The number of matrix rows to be combined.
<i>n</i>	INTEGER. The number of matrix columns to be combined.
<i>a</i>	TYPE array (<i>lda</i> , <i>n</i>). Matrix to be added to produce the sum.
<i>lda</i>	INTEGER. The leading dimension of the matrix <i>A</i> , that is, the distance between two successive elements in a matrix row.
<i>rdest</i>	INTEGER. The process row coordinate of the process that should receive the result. If <i>rdest</i> or <i>cdest</i> = -1, all processes within the indicated scope receive the answer.
<i>cdest</i>	INTEGER. The process column coordinate of the process that should receive the result. If <i>rdest</i> or <i>cdest</i> = -1, all processes within the indicated scope receive the answer.

Output Parameters

a TYPE array (*lda*, *n*). Contains the result if this process is selected to receive the answer, or intermediate results if the process is not selected to receive the result.

Description

This routine performs element-wise summation, that is, each element of matrix *A* is summed with the corresponding element of the other process's matrices. Combines may be globally-blocking, so they must be programmed as if no process returns until all have called the routine.

BLACS Point To Point Communication

This section describes BLACS routines for point to point communication.

Point to point communication requires two complementary operations. The *send* operation produces a message that is then consumed by the *receive* operation. These operations have various resources associated with them. The main such resource is the buffer that holds the data to be sent or serves as the area where the incoming data is to be received. The level of *blocking* indicates what correlation the return from a send/receive operation has with the availability of these resources and with the status of message.

Non-blocking

The return from the *send* or *receive* operations does not imply that the resources may be reused, that the message has been sent/received or that the complementary operation has been called. Return means only that the send/receive has been started, and will be completed at some later date. Polling is required to determine when the operation has finished.

In non-blocking message passing, the concept of *communication/computation overlap* (abbreviated C/C overlap) is important. If a system possesses C/C overlap, independent computation can occur at the same time as communication. That means a nonblocking operation can be posted, and unrelated work can be done while the message is sent/received in parallel. If C/C overlap is not present, after returning from the routine call, computation will be interrupted at some later date when the message is actually sent or received.

Locally-blocking

Return from the *send* or *receive* operations indicates that the resources may be reused. However, since this only depends on local information, it is unknown whether the complementary operation has been called. There are no locally-blocking receives: the send must be completed before the receive buffer is available for re-use.

If a receive has not been posted at the time a locally-blocking send is issued, buffering will be required to avoid losing the message. Buffering can be done on the sending process, the receiving process, or not done at all, losing the message.

Globally-blocking

Return from a globally-blocking procedure indicates that the operation resources may be reused, and that complement of the operation has at least been posted. Since the receive has been posted, there is no buffering required for globally-blocking sends: the message is always sent directly into the user's receive buffer.

Almost all processors support non-blocking communication, as well as some other level of blocking sends. What level of blocking the send possesses varies between platforms. For instance, the Intel® processors support locally-blocking sends, with buffering done on the receiving process. This is a very important distinction, because codes written assuming locally-blocking sends will hang on platforms with globally-blocking sends. Below is a simple example of how this can occur:

```
IAM = MY_PROCESS_ID()
IF (IAM .EQ. 0) THEN
    SEND TO PROCESS 1
    RECV FROM PROCESS 1
ELSE IF (IAM .EQ. 1) THEN
    SEND TO PROCESS 0
    RECV FROM PROCESS 0
END IF
```

If the send is globally-blocking, process 0 enters the send, and waits for process 1 to start its receive before continuing. In the meantime, process 1 starts to send to 0, and waits for 0 to receive before continuing. Both processes are now waiting on each other, and the program will never continue.

The solution for this case is obvious. One of the processes simply reverses the order of its communication calls and the hang is avoided. However, when the communication is not just between two processes, but rather involves a hierarchy of processes, determining how to avoid this kind of difficulty can become problematic.

For this reason, it was decided the BLACS would support locally-blocking sends. On systems natively supporting globally-blocking sends, non-blocking sends coupled with buffering is used to simulate locally-blocking sends. The BLACS support globally-blocking receives.

In addition, the BLACS specify that point to point messages between two given processes will be strictly ordered. If process 0 sends three messages (label them *A*, *B*, and *C*) to process 1, process 1 must receive *A* before it can receive *B*, and message *C* can be received only after both *A* and *B*. The main reason for this restriction is that it allows for the computation of message identifiers.

Note, however, that messages from different processes are not ordered. If processes 0, . . . , 3 send messages *A*, . . . , *D* to process 4, process 4 may receive these messages in any order that is convenient.

Convention

The convention used in the communication routine names follows the template `?xxyy2d`, where the letter in the `?` position indicates the data type being sent, `xx` is replaced to indicate the shape of the matrix, and the `yy` positions are used to indicate the type of communication to perform:

i	integer
s	single precision real
d	double precision real
c	single precision complex
z	double precision complex
ge	The data to be communicated is stored in a general rectangular matrix.
tr	The data to be communicated is stored in a trapezoidal matrix.
sd	Send. One process sends to another.
rv	Receive. One process receives from another.

Table 16-2 BLACS Point To Point Communication

Routine name	Operation performed
?gesd2d ?trsd2d	Take the indicated matrix and send it to the destination process.
?gerv2d ?trrv2d	Receive a message from the process into the matrix.

As a simple example, the pseudo code given above is rewritten below in terms of the BLACS. It is further specified that the data being exchanged is the double precision vector x , which is 5 elements long.

```
CALL GRIDINFO(NPROW, NPCOL, MYPROW, MYPCOL)
```

```
IF (MYPROW.EQ.0 .AND. MYPCOL.EQ.0) THEN
    CALL DGESD2D(5, 1, X, 5, 1, 0)
    CALL DGERV2D(5, 1, X, 5, 1, 0)
ELSE IF (MYPROW.EQ.1 .AND. MYPCOL.EQ.0) THEN
    CALL DGESD2D(5, 1, X, 5, 0, 0)
    CALL DGERV2D(5, 1, X, 5, 0, 0)
END IF
```

?gesd2d

Takes a general rectangular matrix and sends it to the destination process.

Syntax

```
call igesd2d( icontxt, m, n, a, lda, rdest, cdest )
call sgesd2d( icontxt, m, n, a, lda, rdest, cdest )
call dgesd2d( icontxt, m, n, a, lda, rdest, cdest )
call cgesd2d( icontxt, m, n, a, lda, rdest, cdest )
call zgesd2d( icontxt, m, n, a, lda, rdest, cdest )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>m, n, a, lda</i>	Describe the matrix to be sent. See Matrix Shapes for details.
<i>rdest</i>	INTEGER. The process row coordinate of the process to send the message to.
<i>cdest</i>	INTEGER. The process column coordinate of the process to send the message to.

Description

This routine takes the indicated general rectangular matrix and sends it to the destination process located at {RDEST, CDEST} in the process grid. Return from the routine indicates that the buffer (the matrix *A*) may be reused. The routine is locally-blocking, that is, it will return even if the corresponding receive is not posted.

?trsd2d

Takes a trapezoidal matrix and sends it to the destination process.

Syntax

```
call itrdsd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
call strsd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
call dtrsd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
call ctrsd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
call ztrsd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>uplo, diag, m,</i> <i>n, a, lda</i>	Describe the matrix to be sent. See Matrix Shapes for details.

<i>rdest</i>	INTEGER. The process row coordinate of the process to send the message to.
<i>cdest</i>	INTEGER. The process column coordinate of the process to send the message to.

Description

This routine takes the indicated trapezoidal matrix and sends it to the destination process located at {RDEST, CDEST} in the process grid. Return from the routine indicates that the buffer (the matrix *A*) may be reused. The routine is locally-blocking, that is, it will return even if the corresponding receive is not posted.

?gerv2d

Receives a message from the process into the general rectangular matrix.

Syntax

```
call igerv2d( icontxt, m, n, a, lda, rsrc, csrc )
call sgerv2d( icontxt, m, n, a, lda, rsrc, csrc )
call dgerv2d( icontxt, m, n, a, lda, rsrc, csrc )
call cgerv2d( icontxt, m, n, a, lda, rsrc, csrc )
call zgerv2d( icontxt, m, n, a, lda, rsrc, csrc )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>m, n, lda</i>	Describe the matrix to be sent. See Matrix Shapes for details.
<i>rsrc</i>	INTEGER. The process row coordinate of the source of the message.
<i>csrc</i>	INTEGER. The process column coordinate of the source of the message.

Output Parameters

a An array of dimension (lda, n) to receive the incoming message into.

Description

This routine receives a message from process {RSRC, CSRC} into the general rectangular matrix *A*. This routine is globally-blocking, that is, return from the routine indicates that the message has been received into *A*.

?trrv2d

Receives a message from the process into the trapezoidal matrix.

Syntax

```
call itrsv2d( icontxt, uplo, diag, m, n, a, lda, rsrc, csrc )
call strsv2d( icontxt, uplo, diag, m, n, a, lda, rsrc, csrc )
call dtrsv2d( icontxt, uplo, diag, m, n, a, lda, rsrc, csrc )
call ctrsv2d( icontxt, uplo, diag, m, n, a, lda, rsrc, csrc )
call ztrsv2d( icontxt, uplo, diag, m, n, a, lda, rsrc, csrc )
```

Input Parameters

icontxt INTEGER. Integer handle that indicates the context.

uplo, diag, m, n, lda Describe the matrix to be sent. See [Matrix Shapes](#) for details.

rsrc INTEGER.
The process row coordinate of the source of the message.

csrc INTEGER.
The process column coordinate of the source of the message.

Output Parameters

a An array of dimension (lda, n) to receive the incoming message into.

Description

This routine receives a message from process {RSRC, CSRC} into the trapezoidal matrix A . This routine is globally-blocking, that is, return from the routine indicates that the message has been received into A .

BLACS Broadcast Routines

This section describes BLACS broadcast routines.

A broadcast sends data possessed by one process to all processes within a scope. Broadcast, much like point to point communication, has two complementary operations. The process that owns the data to be broadcast issues a *broadcast/send*. All processes within the same scope must then issue the complementary *broadcast/receive*.

The BLACS define that both broadcast/send and broadcast/receive are *globally-blocking*. Broadcasts/receives cannot be locally-blocking since they must post a receive. Note that receives cannot be locally-blocking. When a given process can leave, a broadcast/receive operation is topology dependent, so, to avoid a hang as topology is varied, the broadcast/receive must be treated as if no process can leave until all processes have called the operation.

Broadcast/sends could be defined to be *locally-blocking*. Since no information is being received, as long as locally-blocking point to point sends are used, the broadcast/send will be locally blocking. However, defining one process within a scope to be locally-blocking while all other processes are globally-blocking adds little to the programmability of the code. On the other hand, leaving the option open to have globally-blocking broadcast/sends may allow for optimization on some platforms.

The fact that broadcasts are defined as globally-blocking has several important implications. The first is that scoped operations (broadcasts or combines) must be strictly ordered, that is, all processes within a scope must agree on the order of calls to separate scoped operations. This constraint falls in line with that already in place for the computation of message IDs, and is present in point to point communication as well.

A less obvious result is that scoped operations with `SCOPE = 'ALL'` must be ordered with respect to any other scoped operation. This means that if there are two broadcasts to be done, one along a column, and one involving the entire process grid, all processes within the process column issuing the column broadcast must agree on which broadcast will be performed first.

The convention used in the communication routine names follows the template `?xxyy2d`, where the letter in the `?` position indicates the data type being sent, `xx` is replaced to indicate the shape of the matrix, and the `yy` positions are used to indicate the type of communication to perform:

`i` integer

s	single precision real
d	double precision real
c	single precision complex
z	double precision complex
ge	The data to be communicated is stored in a general rectangular matrix.
tr	The data to be communicated is stored in a trapezoidal matrix.
bs	Broadcast/send. A process begins the broadcast of data within a scope.
br	Broadcast/receive A process receives and participates in the broadcast of data within a scope.

Table 16-3 BLACS Broadcast Routines

Routine name	Operation performed
?gebs2d	Start a broadcast along a scope.
?trbs2d	
?gebr2d	Receive and participate in a broadcast along a scope.
?trbr2d	

?gebs2d

Starts a broadcast along a scope for a general rectangular matrix.

Syntax

```
call igebs2d( icontxt, scope, top, m, n, a, lda )
call sgebs2d( icontxt, scope, top, m, n, a, lda )
call dgebs2d( icontxt, scope, top, m, n, a, lda )
call cgebs2d( icontxt, scope, top, m, n, a, lda )
call zgebs2d( icontxt, scope, top, m, n, a, lda )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>scope</i>	CHARACTER*1. Indicates what scope the broadcast should proceed on. Limited to 'Row', 'Column', or 'All'.

<i>top</i>	CHARACTER*1. Indicates the communication pattern to use for the broadcast.
<i>m, n, a, lda</i>	Describe the matrix to be sent. See Matrix Shapes for details.

Description

This routine starts a broadcast along a scope. All other processes within the scope must call broadcast/receive for the broadcast to proceed. At the end of a broadcast, all processes within the scope will possess the data in the general rectangular matrix *A*.

Broadcasts may be globally-blocking. This means no process is guaranteed to return from a broadcast until all processes in the scope have called the appropriate routine (broadcast/send or broadcast/receive).

?trbs2d

Starts a broadcast along a scope for a trapezoidal matrix.

Syntax

```
call itrbs2d(icontxt, scope, top, uplo, diag, m, n, a, lda )
call strbs2d(icontxt, scope, top, uplo, diag, m, n, a, lda )
call dtrbs2d(icontxt, scope, top, uplo, diag, m, n, a, lda )
call ctrbs2d(icontxt, scope, top, uplo, diag, m, n, a, lda )
call ztrbs2d(icontxt, scope, top, uplo, diag, m, n, a, lda )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>scope</i>	CHARACTER*1. Indicates what scope the broadcast should proceed on. Limited to 'Row', 'Column', or 'All'.
<i>top</i>	CHARACTER*1. Indicates the communication pattern to use for the broadcast.
<i>uplo, diag, m,</i> <i>n, a, lda</i>	Describe the matrix to be sent. See Matrix Shapes for details.

Description

This routine starts a broadcast along a scope. All other processes within the scope must call broadcast/receive for the broadcast to proceed. At the end of a broadcast, all processes within the scope will possess the data in the trapezoidal matrix A .

Broadcasts may be globally-blocking. This means no process is guaranteed to return from a broadcast until all processes in the scope have called the appropriate routine (broadcast/send or broadcast/receive).

?gebr2d

Receives and participates in a broadcast along a scope for a general rectangular matrix.

Syntax

```
call igebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )
call sgebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )
call dgebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )
call cgebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )
call zgebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>scope</i>	CHARACTER*1. Indicates what scope the broadcast should proceed on. Limited to 'Row', 'Column', or 'All'.
<i>top</i>	CHARACTER*1. Indicates the communication pattern to use for the broadcast.
<i>m, n, lda</i>	Describe the matrix to be sent. See Matrix Shapes for details.
<i>rsrc</i>	INTEGER. The process row coordinate of the process that called broadcast/send.
<i>csrc</i>	INTEGER. The process column coordinate of the process that called broadcast/send.

Output Parameters

a An array of dimension (lda, n) to receive the incoming message into.

Description

This routine receives and participates in a broadcast along a scope. At the end of a broadcast, all processes within the scope will possess the data in the general rectangular matrix *A*. Broadcasts may be globally-blocking. This means no process is guaranteed to return from a broadcast until all processes in the scope have called the appropriate routine (broadcast/send or broadcast/receive).

?trbr2d

Receives and participates in a broadcast along a scope for a trapezoidal matrix.

Syntax

```
call itrbr2d( icontxt, scope, top, uplo, diag, m, n, a, lda, rsrc, csrc )
call strbr2d( icontxt, scope, top, uplo, diag, m, n, a, lda, rsrc, csrc )
call dtrbr2d( icontxt, scope, top, uplo, diag, m, n, a, lda, rsrc, csrc )
call ctrbr2d( icontxt, scope, top, uplo, diag, m, n, a, lda, rsrc, csrc )
call ztrbr2d( icontxt, scope, top, uplo, diag, m, n, a, lda, rsrc, csrc )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>scope</i>	CHARACTER*1. Indicates what scope the broadcast should proceed on. Limited to 'Row', 'Column', or 'All'.
<i>top</i>	CHARACTER*1. Indicates the communication pattern to use for the broadcast.
<i>uplo, diag, m, n, lda</i>	Describe the matrix to be sent. See Matrix Shapes for details.
<i>rsrc</i>	INTEGER. The process row coordinate of the process that called broadcast/send.

csrc INTEGER.
The process column coordinate of the process that called broadcast/send.

Output Parameters

a An array of dimension (lda,n) to receive the incoming message into.

Description

This routine receives and participates in a broadcast along a scope. At the end of a broadcast, all processes within the scope will possess the data in the trapezoidal matrix *A*. Broadcasts may be globally-blocking. This means no process is guaranteed to return from a broadcast until all processes in the scope have called the appropriate routine (broadcast/send or broadcast/receive).

BLACS Support Routines

The support routines perform distinct tasks that can be used for:

- Initialization
- Destruction
- Information Purposes
- Miscellaneous Tasks.

Initialization Routines

This section describes BLACS routines that deal with grid/context creation, and processing before the grid/context has been defined.

Table 16-4 BLACS Intialization Routines

Routine name	Operation performed
<i>blacs_pinfo</i>	Returns the number of processes available for use.
<i>blacs_setup</i>	Allocates virtual machine and spawns processes.
<i>blacs_get</i>	Gets values that BLACS use for internal defaults.
<i>blacs_set</i>	Sets values that BLACS use for internal defaults.

Routine name	Operation performed
<code>blacs_gridinit</code>	Assigns available processes into BLACS process grid.
<code>blacs_gridmap</code>	Maps available processes into BLACS process grid.

blacs_pinfo

Returns the number of processes available for use.

Syntax

```
call blacs_pinfo( mypnum, nprocs )
```

Output Parameters

mypnum INTEGER. An integer between 0 and (*nprocs* - 1) that uniquely identifies each process.

nprocs INTEGER. The number of processes available for BLACS use.

Description

This routine is used when some initial system information is required before the BLACS are set up. On all platforms except PVM, *nprocs* is the actual number of processes available for use, that is, *nprows* * *npcols* ≤ *nprocs*. In PVM, the virtual machine may not have been set up before this call, and therefore no parallel machine exists. In this case, *nprocs* is returned as less than one. If a process has been spawned via the keyboard, it receives *mypnum* of 0, and all other processes get *mypnum* of -1. As a result, the user can distinguish between processes. Only after the virtual machine has been set up via a call to `BLACS_SETUP`, this routine returns the correct values for *mypnum* and *nprocs*.

blacs_setup

Allocates virtual machine and spawns processes.

Syntax

```
call blacs_setup( mypnum, nprocs )
```


Input Parameters

nprocs INTEGER. On the process spawned from the keyboard rather than from `pvmspawn`, this parameter indicates the number of processes to create when building the virtual machine.

Output Parameters

mypnum INTEGER. An integer between 0 and (*nprocs* - 1) that uniquely identifies each process.

nprocs INTEGER. For all processes other than spawned from the keyboard, this parameter means the number of processes available for BLACS use.

Description

This routine only accomplishes meaningful work in the PVM BLACS. On all other platforms, it is functionally equivalent to `blacs_pinfo`. The BLACS assume a static system, that is, the given number of processes does not change. PVM supplies a dynamic system, allowing processes to be added to the system on the fly.

`blacs_setup` is used to allocate the virtual machine and spawn off processes. It reads in a file called `blacs_setup.dat`, in which the first line must be the name of your executable. The second line is optional, but if it exists, it should be a PVM spawn flag. Legal values at this time are 0 (`PvmTaskDefault`), 4 (`PvmTaskDebug`), 8 (`PvmTaskTrace`), and 12 (`PvmTaskDebug + PvmTaskTrace`). The primary reason for this line is to allow the user to easily turn on and off PVM debugging. Additional lines, if any, specify what machines should be added to the current configuration before spawning *nprocs*-1 processes to the machines in a round robin fashion.

nprocs is input on the process which has no PVM parent (that is, *mypnum*=0), and both parameters are output for all processes. So, on PVM systems, the call to `blacs_pinfo` informs you that the virtual machine has not been set up, and a call to `blacs_setup` then sets up the machine and returns the real values for *mypnum* and *nprocs*.

Note that if the file `blacs_setup.dat` does not exist, the BLACS prompt the user for the executable name, and processes are spawned to the current PVM configuration.

blacs_get

Gets values that BLACS use for internal defaults.

Syntax

```
call blacs_get( icoctxt, what, val )
```

Input Parameters

<i>icoctxt</i>	INTEGER. On values of <i>what</i> that are tied to a particular context, this parameter is the integer handle indicating the context. Otherwise, ignored.
<i>what</i>	INTEGER. Indicates what BLACS internal(s) should be returned in <i>val</i>. Present options are: • <i>what</i> = 0 : Handle indicating default system context • <i>what</i> = 1 : The BLACS message ID range • <i>what</i> = 2 : The BLACS debug level the library was compiled with • <i>what</i> = 10 : Handle indicating the system context used to define the BLACS context whose handle is <i>icoctxt</i> • <i>what</i> = 11 : Number of rings multiring topology is presently using • <i>what</i> = 12 : Number of branches general tree topology is presently using.

Output Parameters

<i>val</i>	INTEGER. The value of the BLACS internal.

Description

This routine gets the values that the BLACS are using for internal defaults. Some values are tied to a BLACS context, and some are more general. The most common use is in retrieving a default system context for input into [blacs_gridinit](#) or [blacs_gridmap](#).

Some systems, such as MPI*, supply their own version of context. For those users who mix system code with BLACS code, a BLACS context should be formed in reference to a system context. Thus, the grid creation routines take a system context as input. If you wish to have

strictly portable code, you may use `blacs_get` to retrieve a default system context that will include all available processes. This value is not tied to a BLACS context, so the parameter `icontxt` is unused.

`blacs_get` returns information on three quantities that are tied to an individual BLACS context, which is passed in as `icontxt`. The information that may be retrieved is:

- The handle of the system context upon which this BLACS context was defined
- The number of rings for `TOP = 'M'` (multiring broadcast)
- The number of branches for `TOP = 'T'` (general tree broadcast/general tree gather).

blacs_set

Sets values that BLACS use for internal defaults.

Syntax

```
call blacs_set( icontxt, what, val )
```

Input Parameters

<i>icontxt</i>	INTEGER. For values of <i>what</i> that are tied to a particular context, this parameter is the integer handle indicating the context. Otherwise, ignored.
<i>what</i>	INTEGER. Indicates what BLACS internal(s) should be set. Present values are: • 1 = The BLACS message ID range • 11 = Number of rings for multiring topology to use • 12 = Number of branches for general tree topology to use.
<i>val</i>	INTEGER. Array of dimension (*). Indicates the value(s) the internals should be set to. The specific meanings depend on <i>what</i> values, as discussed in Description below.

Description

This routine sets the BLACS internal defaults depending on *what* values:

what = 1 Setting the BLACS message ID range.

If you wish to mix the BLACS with other message-passing packages, restrict the BLACS to a certain message ID range not to be used by the non-BLACS routines. The message ID range must be set before the first call to `blacs_gridinit` or `blacs_gridmap`. Subsequent calls will have no effect. Because the message ID range is not tied to a particular context, the parameter *icontxt* is ignored, and *val* is defined as:

VAL (input) INTEGER array of dimension (2)

VAL(1) : The smallest message ID (also called message type or message tag) the BLACS should use.

VAL(2) : The largest message ID (also called message type or message tag) the BLACS should use.

what = 11

Set number of rings for TOP = 'M' (multiring broadcast). This quantity is tied to a context, so *icontxt* is used, and *val* is defined as:

VAL (input) INTEGER array of dimension (1)

VAL(1) : The number of rings for multiring topology to use.

what = 12

Set number of rings for TOP = 'T' (general tree broadcast/general tree gather). This quantity is tied to a context, so *icontxt* is used, and *val* is defined as:

VAL (input) INTEGER array of dimension (1)

VAL(1) : The number of branches for general tree topology to use.

blacs_gridinit

Assigns available processes into BLACS process grid.

Syntax

call `blacs_gridinit( icontxt, order, nprow, npcol )`

Input Parameters

icontxt

INTEGER. Integer handle indicating the system context to be used in creating the BLACS context. Call `blacs_get` to obtain a default system context.

order

CHARACTER*1. Indicates how to map processes to BLACS grid. Options are:

- 'R' : Use row-major natural ordering

- 'C' : Use column-major natural ordering
- ELSE : Use row-major natural ordering

nprow INTEGER. Indicates how many process rows the process grid should contain.

npcol INTEGER. Indicates how many process columns the process grid should contain.

Output Parameters

icontxt INTEGER. Integer handle to the created BLACS context.

Description

All BLACS codes must call this routine, or its sister routine `blacs_gridmap`. These routines take the available processes, and assign, or map, them into a BLACS process grid. In other words, they establish how the BLACS coordinate system maps into the native machine process numbering system. Each BLACS grid is contained in a context, so that it does not interfere with distributed operations that occur within other grids/contexts. These grid creation routines may be called repeatedly to define additional contexts/grids.

The creation of a grid requires input from all processes that are defined to be in this grid. Processes belonging to more than one grid have to agree on which grid formation will be serviced first, much like the globally blocking sum or broadcast.

These grid creation routines set up various internals for the BLACS, and one of them must be called before any calls are made to the non-initialization BLACS.

Note that these routines map already existing processes to a grid: the processes are not created dynamically. On most parallel machines, the processes are actual processors (hardware), and they are "created" when you run your executable. When using the PVM BLACS, if the virtual machine has not been set up yet, the routine `blacs_setup` should be used to create the virtual machine.

This routine creates a simple `nprow x npcol` process grid. This process grid uses the first `nprow * npcol` processes, and assigns them to the grid in a row- or column-major natural ordering. If these process-to-grid mappings are unacceptable, call `blacs_gridmap`.

blacs_gridmap

Maps available processes into BLACS process grid.

Syntax

```
call blacs_gridmap( ictxt, usermap, ldumap, nprow, npcol )
```

Input Parameters

<i>ictxt</i>	INTEGER. Integer handle indicating the system context to be used in creating the BLACS context. Call <code>blacs_get</code> to obtain a default system context.
<i>usermap</i>	INTEGER. Array, dimension (<i>ldumap</i> , <i>npcol</i>), indicating the process-to-grid mapping.
<i>ldumap</i>	INTEGER. Leading dimension of the 2D array <i>usermap</i> . $ldumap \geq nprow$.
<i>nprow</i>	INTEGER. Indicates how many process rows the process grid should contain.
<i>npcol</i>	INTEGER. Indicates how many process columns the process grid should contain.

Output Parameters

<i>ictxt</i>	INTEGER. Integer handle to the created BLACS context.

Description

All BLACS codes must call this routine, or its sister routine `blacs_gridinit`. These routines take the available processes, and assign, or map, them into a BLACS process grid. In other words, they establish how the BLACS coordinate system maps into the native machine process numbering system. Each BLACS grid is contained in a context, so that it does not interfere with distributed operations that occur within other grids/contexts. These grid creation routines may be called repeatedly to define additional contexts/grids.

The creation of a grid requires input from all processes that are defined to be in this grid. Processes belonging to more than one grid have to agree on which grid formation will be serviced first, much like the globally blocking sum or broadcast.

These grid creation routines set up various internals for the BLACS, and one of them must be called before any calls are made to the non-initialization BLACS.

Note that these routines map already existing processes to a grid: the processes are not created dynamically. On most parallel machines, the processes are actual processors (hardware), and they are "created" when you run your executable. When using the PVM BLACS, if the virtual machine has not been set up yet, the routine `blacs_setup` should be used to create the virtual machine.

This routine allows the user to map processes to the process grid in an arbitrary manner.

`usermap(i,j)` holds the process number of the process to be placed in $\{i, j\}$ of the process grid. On most distributed systems, this process number is a machine defined number between $0 \dots nprow-1$. For PVM, these node numbers are the PVM TIDS (Task IDs). The `blacs_gridmap` routine is intended for an experienced user. The `blacs_gridinit` routine is much simpler.

`blacs_gridinit` simply performs a `gridmap` where the first $nprow * npcol$ processes are mapped into the current grid in a row-major natural ordering. If you are an experienced user, `blacs_gridmap` allows you to take advantage of your system's actual layout. That is, you can map nodes that are physically connected to be neighbors in the BLACS grid, etc. The `blacs_gridmap` routine also opens the way for *multigridding*: you can separate your nodes into arbitrary grids, join them together at some later date, and then re-split them into new grids. `blacs_gridmap` also provides the ability to make arbitrary grids or subgrids (for example, a "nearest neighbor" grid), which can greatly facilitate operations among processes that do not fall on a row or column of the main process grid.

Destruction Routines

This section describes BLACS routines that destroy grids, abort processes, and free resources.

Table 16-5 BLACS Destruction Routines

Routine name	Operation performed
<code>blacs_freebuff</code>	Frees BLACS buffer.
<code>blacs_gridexit</code>	Frees a BLACS context.
<code>blacs_abort</code>	Aborts all processes.
<code>blacs_exit</code>	Frees all BLACS contexts and releases all allocated memory.

blacs_freebuff

Frees BLACS buffer.

Syntax

```
call blacs_freebuff( ictxt, wait )
```

Input Parameters

<i>ictxt</i>	INTEGER. Integer handle that indicates the BLACS context.
<i>wait</i>	INTEGER. Parameter indicating whether to wait for non-blocking operations or not. If equals 0, the operations should not be waited for; free only unused buffers. Otherwise, wait in order to free all buffers.

Description

This routine releases the BLACS buffer.

The BLACS have at least one internal buffer that is used for packing messages. The number of internal buffers depends on what platform you are running the BLACS on. On systems where memory is tight, keeping this buffer or buffers may become expensive. Call `freebuff` to release the buffer. However, the next call of a communication routine that requires packing reallocates the buffer.

The *wait* parameter determines whether the BLACS should wait for any non-blocking operations to be completed or not. If *wait* = 0, the BLACS free any buffers that can be freed without waiting. If *wait* is not 0, the BLACS free all internal buffers, even if non-blocking operations must be completed first.

blacs_gridexit

Frees a BLACS context.

Syntax

```
call blacs_gridexit( ictxt )
```

Input Parameters

<i>ictxt</i>	INTEGER. Integer handle that indicates the BLACS context to be freed.

Description

This routine frees a BLACS context.

Release the resources when contexts are no longer needed. After freeing a context, the context no longer exists, and its handle may be re-used if new contexts are defined.

blacs_abort

Aborts all processes.

Syntax

```
call blacs_abort( icontxt, errornum )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the BLACS context to be aborted.
<i>errornum</i>	INTEGER. User-defined integer error number.

Description

This routine aborts all the BLACS processes, not only those confined to a particular context.

Use `blacs_abort` to abort all the processes in case of a serious error. Note that both parameters are input, but the routine uses them only in printing out the error message. The context handle passed in is not required to be a valid context handle.

blacs_exit

Frees all BLACS contexts and releases all allocated memory.

Syntax

```
call blacs_exit( continue )
```

Input Parameters

continue INTEGER. Flag indicating whether message passing continues after the BLACS are done. If *continue* is non-zero, the user is assumed to continue using the machine after completing the BLACS. Otherwise, no message passing is assumed after calling this routine.

Description

This routine frees all BLACS contexts and releases all allocated memory.

This routine should be called when a process has finished all use of the BLACS. The *continue* parameter indicates whether the user will be using the underlying communication platform after the BLACS are finished. This information is most important for the PVM BLACS. If *continue* is set to 0, then `pvm_exit` is called; otherwise, it is not called. Setting *continue* not equal to 0 indicates that explicit PVM `send/recvs` will be called after the BLACS routines are used. Make sure your code calls `pvm_exit`. PVM users should either call `blacs_exit` or explicitly call `pvm_exit` to avoid PVM problems.

Informational Routines

This section describes BLACS routines that return information involving the process grid.

Table 16-6 BLACS Informational Routines

Routine name	Operation performed
<code>blacs_gridinfo</code>	Returns information on the current grid.
<code>blacs_pnum</code>	Returns the system process number of the process in the process grid.
<code>blacs_pcoord</code>	Returns the row and column coordinates in the process grid.

`blacs_gridinfo`

Returns information on the current grid.

Syntax

```
call blacs_gridinfo( ictxt, nprow, npcol, myprow, mypcol )
```

Input Parameters

icontxt INTEGER. Integer handle that indicates the context.

Output Parameters

nprow INTEGER. Number of process rows in the current process grid.

npcol INTEGER. Number of process columns in the current process grid.

myproW INTEGER. Row coordinate of the calling process in the process grid.

mypcol INTEGER. Column coordinate of the calling process in the process grid.

Description

This routine returns information on the current grid. If the context handle does not point at a valid context, all quantities are returned as -1.

blacs_pnum

Returns the system process number of the process in the process grid.

Syntax

call blacs_pnum(*icontxt*, *prow*, *pcol*)

Input Parameters

icontxt INTEGER. Integer handle that indicates the context.

prow INTEGER. Row coordinate of the process the system process number of which is to be determined.

pcol INTEGER. Column coordinate of the process the system process number of which is to be determined.

Description

This function returns the system process number of the process at {*PROW*, *PCOL*} in the process grid.

blacs_pcoord

Returns the row and column coordinates in the process grid.

Syntax

```
call blacs_pcoord( ictxt, pnum, prow, pcol )
```

Input Parameters

<i>ictxt</i>	INTEGER. Integer handle that indicates the context.
<i>pnum</i>	INTEGER. Process number the coordinates of which are to be determined. This parameter stand for the process number of the underlying machine, that is, it is a <code>tid</code> for PVM.

Output Parameters

<i>prow</i>	INTEGER. Row coordinates of the <i>pnum</i> process in the BLACS grid.
<i>pcol</i>	INTEGER. Column coordinates of the <i>pnum</i> process in the BLACS grid.

Description

Given the system process number, this function returns the row and column coordinates in the BLACS process grid.

Miscellaneous Routines

This section describes `blacs_barrier` routine.

Table 16-7 BLACS Informational Routines

Routine name	Operation performed
<code>blacs_barrier</code>	Holds up execution of all processes within the indicated scope until they have all called the routine.

blacs_barrier

Holds up execution of all processes within the indicated scope.

Syntax

```
call blacs_barrier( icontxt, scope )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>scope</i>	CHARACTER*1. Parameter that indicates whether a process row (<i>scope</i> ='R'), column ('C'), or entire grid ('A') will participate in the barrier.

Description

This routine holds up execution of all processes within the indicated scope until they have all called the routine.

Examples of BLACS Routines Usage

Example 16-1. BLACS Usage. Hello World

The following routine takes the available processes, forms them into a process grid, and then has each process check in with the process at {0,0} in the process grid.

```
PROGRAM HELLO
*
*  -- BLACS example code --
*
*  Written by Clint Whaley 7/26/94
*
*  Performs a simple check-in type hello world
*
*  ..
*
*  .. External Functions ..
*
*  INTEGER BLACS_PNUM
*
*  EXTERNAL BLACS_PNUM
*
*  ..
*
*  .. Variable Declaration ..
*
*  INTEGER CONXTXT, IAM, NPROCS, NPROW, NPCOL, MYPROW, MYPCOL
*
*  INTEGER ICALLER, I, J, HISROW, HISCOL
*
*
*  Determine my process number and the number of processes in
*  machine
*
*
*  CALL BLACS_PINFO(IAM, NPROCS)
*
*
*  If in PVM, create virtual machine if it doesn't exist
*
*
*  IF (NPROCS .LT. 1) THEN
*    IF (IAM .EQ. 0) THEN
*      WRITE(*, 1000)
```

```

        READ(*, 2000) NPROCS
    END IF

    CALL BLACS_SETUP(IAM, NPROCS)
END IF

*
*   Set up process grid that is as close to square as possible
*
    NPROW = INT( SQRT( REAL(NPROCS) ) )
    NPCOL = NPROCS / NPROW

*
*   Get default system context, and define grid
*
    CALL BLACS_GET(0, 0, CONTXT)
    CALL BLACS_GRIDINIT(CONTXT, 'Row', NPROW, NPCOL)
    CALL BLACS_GRIDINFO(CONTXT, NPROW, NPCOL, MYPROW, MYPCOL)

*
*   If I'm not in grid, go to end of program
*
    IF ( (MYPROW.GE.NPROW) .OR. (MYPCOL.GE.NPCOL) ) GOTO 30

*
*   Get my process ID from my grid coordinates
*
    ICALLER = BLACS_PNUM(CONTXT, MYPROW, MYPCOL)

*
*   If I am process {0,0}, receive check-in messages from
*   all nodes
*
    IF ( (MYPROW.EQ.0) .AND. (MYPCOL.EQ.0) ) THEN

```

```

WRITE(*,*) ' '
DO 20 I = 0, NPROW-1
    DO 10 J = 0, NPCOL-1

        IF ( (I.NE.0) .OR. (J.NE.0) ) THEN
            CALL IGERV2D(CONTXT, 1, 1, ICALLER, 1, I, J)
        END IF

*
*       Make sure ICALLER is where we think in process grid
*

        CALL BLACS_PCOORD(CONTXT, ICALLER, HISROW, HISCOL)
        IF ( (HISROW.NE.I) .OR. (HISCOL.NE.J) ) THEN
            WRITE(*,*) 'Grid error! Halting . . .'
            STOP
        END IF
        WRITE(*, 3000) I, J, ICALLER

10      CONTINUE
20      CONTINUE
        WRITE(*,*) ' '
        WRITE(*,*) 'All processes checked in. Run finished.'

*
*       All processes but {0,0} send process ID as a check-in
*

        ELSE
            CALL IGESD2D(CONTXT, 1, 1, ICALLER, 1, 0, 0)

```

```
        END IF

30      CONTINUE

        CALL BLACS_EXIT(0)
1000   FORMAT('How many processes in machine?')
2000   FORMAT(I)
3000   FORMAT('Process {',i2,',',',i2,','} (node number =',I,
$         ') has checked in.')
```

STOP

END

Example 16-2. BLACS Usage. PROCMAP

This routine maps processes to a grid using `blacs_gridmap`.

```

SUBROUTINE PROCMAP(CONTEXT, MAPPING, BEGPROC, NPROW, NPCOL, IMAP)

*
*   -- BLACS example code --
*   Written by Clint Whaley 7/26/94
*   ..
*   .. Scalar Arguments ..
      INTEGER CONTEXT, MAPPING, BEGPROC, NPROW, NPCOL
*   ..
*   .. Array Arguments ..
      INTEGER IMAP(NPROW, *)
*   ..
*
* Purpose
* =====
*
* PROCMAP maps NPROW*NPCOL processes starting from process BEGPROC to
* the grid in a variety of ways depending on the parameter MAPPING.
*
* Arguments
* =====
*
* CONTEXT      (output) INTEGER
*
*              This integer is used by the BLACS to indicate a context.
*
*              A context is a universe where messages exist and do not
*
*              interact with other context's messages.  The context
*
*              includes the definition of a grid, and each process's

```

* coordinates in it.

*

* MAPPING (input) INTEGER

* Way to map processes to grid. Choices are:

* 1 : row-major natural ordering

* 2 : column-major natural ordering

*

* BEGPROC (input) INTEGER

* The process number (between 0 and NPROCS-1) to use as

* {0,0}. From this process, processes will be assigned

* to the grid as indicated by MAPPING.

*

* NPROW (input) INTEGER

* The number of process rows the created grid

* should have.

*

* NPCOL (input) INTEGER

* The number of process columns the created grid

* should have.

*

* IMAP (workspace) INTEGER array of dimension (NPROW, NPCOL)

* Workspace, where the array which maps the

* processes to the grid will be stored for the

* call to GRIDMAP.

```

*
*
* =====
*
*
* ..
*
* .. External Functions ..
*
*   INTEGER  BLACS_PNUM
*
*   EXTERNAL BLACS_PNUM
*
* ..
*
* .. External Subroutines ..
*
*   EXTERNAL BLACS_PINFO, BLACS_GRIDINIT, BLACS_GRIDMAP
*
* ..
*
* .. Local Scalars ..
*
*   INTEGER TMPCONXT, NPROCS, I, J, K
*
* ..
*
* .. Executable Statements ..
*
*
* See how many processes there are in the system
*
*
*   CALL BLACS_PINFO( I, NPROCS )
*
*   IF (NPROCS-BEGPROC .LT. NPROW*NPCOL) THEN
*
*     WRITE(*,*) 'Not enough processes for grid'
*
*     STOP
*
*   END IF
*
*
* Temporarily map all processes into 1 x NPROCS grid
*
*
*   CALL BLACS_GET( 0, 0, TMPCONXT )
*
*   CALL BLACS_GRIDINIT( TMPCONXT, 'Row', 1, NPROCS )

```

```
K = BEGPROC

*
*   If we want a row-major natural ordering
*
*
IF (MAPPING .EQ. 1) THEN
  DO I = 1, NPROW
    DO J = 1, NPCOL
      IMAP(I, J) = BLACS_PNUM(TMPCONXT, 0, K)
      K = K + 1W
    END DO
  END DO
*
*   If we want a column-major natural ordering
*
*
ELSE IF (MAPPING .EQ. 2) THEN
  DO J = 1, NPCOL
    DO I = 1, NPROW
      IMAP(I, J) = BLACS_PNUM(TMPCONXT, 0, K)
      K = K + 1
    END DO
  END DO
ELSE
  WRITE(*,*) 'Unknown mapping.'
  STOP
END IF
*
*   Free temporary context
```

*

```
CALL BLACS_GRIDEXIT(TMPCONXT)
```

*

*

Apply the new mapping to form desired context

*

```
CALL BLACS_GET( 0, 0, CONTEXT )
```

```
CALL BLACS_GRIDMAP( CONTEXT, IMAP, NPROW, NPROW, NPCOL )
```

```
RETURN
```

```
END
```

Example 16-3. BLACS Usage. PARALLEL DOT PRODUCT

This routine does a bone-headed parallel double precision dot product of two vectors. Arguments are input on process {0,0}, and output everywhere else.

```

      DOUBLE PRECISION FUNCTION PDDOT( CONTEXT, N, X, Y )
*
*   -- BLACS example code --
*   Written by Clint Whaley 7/26/94
*   ..
*   .. Scalar Arguments ..
      INTEGER CONTEXT, N
*   ..
*   .. Array Arguments ..
      DOUBLE PRECISION X(*), Y(*)
*   ..
*
* Purpose
* =====
* PDDOT is a restricted parallel version of the BLAS routine
* DDOT.  It assumes that the increment on both vectors is one,
* and that process {0,0} starts out owning the vectors and
*
* has N.  It returns the dot product of the two N-length vectors
* X and Y, that is, PDDOT = X' Y.
*
* Arguments
* =====
*
* CONTEXT      (input) INTEGER
*
*              This integer is used by the BLACS to indicate a context.

```

```

*           A context is a universe where messages exist and do not
*           interact with other context's messages. The context
*
*           includes the definition of a grid, and each process's
*
*           coordinates in it.
*
*
* N         (input/output) INTEGER
*           The length of the vectors X and Y. Input
*           for {0,0}, output for everyone else.
*
*
* X         (input/output) DOUBLE PRECISION array of dimension (N)
*           The vector X of  $PDDOT = X' Y$ . Input for {0,0},
*           output for everyone else.
*
*
* Y         (input/output) DOUBLE PRECISION array of dimension (N)
*           The vector Y of  $PDDOT = X' Y$ . Input for {0,0},
*           output for everyone else.
*

```

```

*      =====
*
*      ..
*      .. External Functions ..
      DOUBLE PRECISION DDOT
      EXTERNAL DDOT
*      ..
*      .. External Subroutines ..
      EXTERNAL BLACS_GRIDINFO, DGEBS2D, DGEBR2D, DGSUM2D
*      ..
*      .. Local Scalars ..
      INTEGER IAM, NPROCS, NPROW, NPCOL, MYPROW, MYPCOL, I, LN

      DOUBLE PRECISION LDDOT
*      ..
*      .. Executable Statements ..
*
*      Find out what grid has been set up, and pretend it is 1-D
*
      CALL BLACS_GRIDINFO( CONTXT, NPROW, NPCOL, MYPROW, MYPCOL )
      IAM = MYPROW*NPCOL + MYPCOL
      NPROCS = NPROW * NPCOL
*
*      Temporarily map all processes into 1 x NPROCS grid
*
      CALL BLACS_GET( 0, 0, TMPCONTXT )
      CALL BLACS_GRIDINIT( TMPCONTXT, 'Row', 1, NPROCS )
      K = BEGPROC

```

```

*
*   Do bone-headed thing, and just send entire X and Y to
*
*   everyone
*
*
  IF ( (MYPROW.EQ.0) .AND. (MYPCOL.EQ.0) ) THEN
    CALL IGEBS2D(CONTXT, 'All', 'i-ring', 1, 1, N, 1 )
    CALL DGEBS2D(CONTXT, 'All', 'i-ring', N, 1, X, N )
    CALL DGEBS2D(CONTXT, 'All', 'i-ring', N, 1, Y, N )
  ELSE
    CALL IGEBR2D(CONTXT, 'All', 'i-ring', 1, 1, N, 1, 0, 0 )
    CALL DGEBR2D(CONTXT, 'All', 'i-ring', N, 1, X, N, 0, 0 )
    CALL DGEBR2D(CONTXT, 'All', 'i-ring', N, 1, Y, N, 0, 0 )
  ENDIF
*
*   Find out the number of local rows to multiply (LN), and
*
*   where in vectors to start (I)
*
*
  LN = N / NPROCS
  I = 1 + IAM * LN
*
*   Last process does any extra rows
*
*
  IF (IAM .EQ. NPROCS-1) LN = LN + MOD(N, NPROCS)
*
*   Figure dot product of my piece of X and Y

```

```
*  
      LDDOT = DDOT( LN, X(I), 1, Y(I), 1 )  
*  
*      Add local dot products to get global dot product;  
*  
*      give all procs the answer  
*  
      CALL DGSUM2D( CONTXT, 'All', '1-tree', 1, 1, LDDOT, 1, -1, 0 )  
  
      PDDOT = LDDOT  
  
      RETURN  
      END
```

Example 16-4 BLACS Usage. PARALLEL MATRIX INFINITY NORM

This routine does a parallel infinity norm on a distributed double precision matrix. Unlike the PDDOT example, this routine assumes the matrix has already been distributed.

```

      DOUBLE PRECISION FUNCTION PDINFNRM(CONTXT, LM, LN, A, LDA, WORK)

*
*   -- BLACS example code --
*   Written by Clint Whaley.
*   ..
*   .. Scalar Arguments ..
      INTEGER CONTEXT, LM, LN, LDA

*   ..
*   .. Array Arguments ..
      DOUBLE PRECISION A(LDA, *), WORK(*)

*   ..
*
* Purpose
* =====
* Compute the infinity norm of a distributed matrix, where
* the matrix is spread across a 2D process grid. The result is
* left on all processes.
*
* Arguments
* =====
*
* CONTEXT      (input) INTEGER
*
*              This integer is used by the BLACS to indicate a context.
*
*              A context is a universe where messages exist and do not

```

* interact with other context's messages. The context

* includes the definition of a grid, and each process's

* coordinates in it.

*

* LM (input) INTEGER

* Number of rows of the global matrix owned by this

* process.

*

* LN (input) INTEGER

* Number of columns of the global matrix owned by this

* process.

*

* A (input) DOUBLE PRECISION, dimension (LDA,N)

* The matrix whose norm you wish to compute.

*

* LDA (input) INTEGER

* Leading Dimension of A.

*

* WORK (temporary) DOUBLE PRECISION array, dimension (LM)

* Temporary work space used for summing rows.

*

```

*      .. External Subroutines ..
      EXTERNAL BLACS_GRIDINFO, DGEBS2D, DGEBR2D, DGSUM2D, DGAMX2D
*
*      ..
*
*      .. External Functions ..
      INTEGER IDAMAX
      DOUBLE PRECISION DASUM
*
*
*      .. Local Scalars ..
      INTEGER NPROW, NPCOL, MYROW, MYCOL,  I, J
*
      DOUBLE PRECISION MAX
*
*
*      .. Executable Statements ..
*
*      Get process grid information
*
      CALL BLACS_GRIDINFO( CONTXT, NPROW, NPCOL, MYPROW, MYPCOL )
*
*      Add all local rows together
*
      DO 20 I = 1, LM
          WORK(I) = DASUM(LN, A(I,1), LDA)
20    CONTINUE
*
*      Find sum of global matrix rows and store on column 0 of
*
*      process grid

```

```
*
      CALL DGSUM2D(CONTXT, 'Row', '1-tree', LM, 1, WORK, LM, MYROW, 0)

*
*   Find maximum sum of rows for supnorm
*
*
      IF (MYCOL .EQ. 0) THEN
        MAX = WORK(IDAMAX(LM,WORK,1))
        IF (LM .LT. 1) MAX = 0.0D0

        CALL DGAMX2D(CONTXT, 'Col', 'h', 1, 1, MAX, 1, I, I, -1, -1, 0)
      END IF

*
*   Process column 0 has answer; send answer to all nodes
*
*
      IF (MYCOL .EQ. 0) THEN
        CALL DGEBS2D(CONTXT, 'Row', ' ', 1, 1, MAX, 1)
      ELSE

        CALL DGEBR2D(CONTXT, 'Row', ' ', 1, 1, MAX, 1, 0, 0)
      END IF

*
      PDINFNRM = MAX

*
      RETURN

*
*   End of PDINFNRM
*
*
```

END

Linear Solvers Basics

Many applications in science and engineering require the solution of a system of linear equations. This problem is usually expressed mathematically by the matrix-vector equation, $Ax = b$, where A is an $m \times n$ matrix, x is the n element column vector and b is the m element column vector. The matrix A is usually referred to as the coefficient matrix, and the vectors x and b are referred to as the solution vector and the right-hand side, respectively.

Basic concepts related to solving linear systems with sparse matrices are described in section [Sparse Linear Systems](#) that follows.

If the coefficients in matrix A and right-hand sides in vector b are not defined exactly but rather belong to known intervals, the system is called an *interval linear system*. Some basic definitions and concepts used in solving interval linear systems are described in section [Interval Linear Systems](#).

Sparse Linear Systems

In many real-life applications, most of the elements in A are zero. Such a matrix is referred to as sparse. Conversely, matrices with very few zero elements are called dense. For sparse matrices, computing the solution to the equation $Ax = b$ can be made much more efficient with respect to both storage and computation time, if the sparsity of the matrix can be exploited. The more an algorithm can exploit the sparsity without sacrificing the correctness, the better the algorithm.

Generally speaking, computer software that finds solutions to systems of linear equations is called a solver. A solver designed to work specifically on sparse systems of equations is called a sparse solver. Solvers are usually classified into two groups - direct and iterative.

Iterative Solvers start with an initial approximation to a solution and attempt to estimate the difference between the approximation and the true result. Based on the difference, an iterative solver calculates a new approximation that is closer to the true result than the initial approximation. This process is repeated until the difference between the approximation and the true result is sufficiently small. The main drawback to iterative solvers is that the rate of convergence depends greatly on the values in the matrix A . Consequently, it is not possible to predict how long it will take for an iterative solver to produce a solution. In fact, for ill-conditioned matrices, the iterative process will not converge to a solution at all. However, for well-conditioned matrices it is possible for iterative solvers to converge to a solution very quickly. Consequently for the right applications, iterative solvers can be very efficient.

Direct Solvers, on the other hand, often factor the matrix A into the product of two triangular matrices and then perform a forward and backward triangular solve.

This approach makes the time required to solve a systems of linear equations relatively predictable, based on the size of the matrix. In fact, for sparse matrices, the solution time can be predicted based on the number of non-zero elements in the array A .

Matrix Fundamentals

A matrix is a rectangular array of either real or complex numbers. A matrix is denoted by a capital letter; its elements are denoted by the same lower case letter with row/column subscripts. Thus, the value of the element in row i and column j in matrix A is denoted by $a(i, j)$. For example, a 3 by 4 matrix A , is written as follows:

$$A = \begin{bmatrix} a(1, 1) & a(1, 2) & a(1, 3) & a(1, 4) \\ a(2, 1) & a(2, 2) & a(2, 3) & a(2, 4) \\ a(3, 1) & a(3, 2) & a(3, 3) & a(3, 4) \end{bmatrix}$$

Note that with the above notation, we assume the standard Fortran programming language convention of starting array indices at 1 rather than the C programming language convention of starting them at 0.

A matrix in which all of the elements are real numbers is called a real matrix. A matrix that contains at least one complex number is called a complex matrix. A real or complex matrix A with the property that $a(i, j) = a(j, i)$, is called a symmetric matrix. A complex matrix A with the property that $a(i, j) = \text{conj}(a(j, i))$, is called a Hermitian matrix. Note that programs that manipulate symmetric and Hermitian matrices need only store half of the matrix values, since the values of the non-stored elements can be quickly reconstructed from the stored values.

A matrix that has the same number of rows as it has columns is referred to as a square matrix. The elements in a square matrix that have same row index and column index are called the diagonal elements of the matrix, or simply the diagonal of the matrix.

The transpose of a matrix A is the matrix obtained by “flipping” the elements of the array about its diagonal. That is, we exchange the elements $a(i, j)$ and $a(j, i)$. For a complex matrix, if we both flip the elements about the diagonal and then take the complex conjugate of the element, the resulting matrix is called the Hermitian transpose or conjugate transpose of the original matrix. The transpose and Hermitian transpose of a matrix A are denoted by A^T and A^H respectively.

A column vector, or simply a vector, is a $n \times 1$ matrix, and a row vector is a $1 \times n$ matrix. A real or complex matrix A is said to be positive definite if the vector-matrix product $x^T A x$ is greater than zero for all non-zero vectors x . A matrix that is not positive definite is referred to as indefinite.

An upper (or lower) triangular matrix, is a square matrix in which all elements below (or above) the diagonal are zero. A unit triangular matrix is an upper or lower triangular matrix with all 1's along the diagonal.

A matrix P is called a permutation matrix if, for any matrix A , the result of the matrix product PA is identical to A except for interchanging the rows of A . For a square matrix, it can be shown that if PA is a permutation of the rows of A , then AP^T is the same permutation of the columns of A . Additionally, it can be shown that the inverse of P is P^T .

In order to save space, a permutation matrix is usually stored as a linear array, called a permutation vector, rather than as an array. Specifically, if the permutation matrix maps the i -th row of a matrix to the j -th row, then the i -th element of the permutation vector is j .

A matrix with non-zero elements only on the diagonal is called a diagonal matrix. As is the case with a permutation matrix, it is usually stored as a vector of values, rather than as a matrix.

Direct Method

For solvers that use the direct method, the basic technique employed in finding the solution of the system $Ax = b$ is to first factor A into triangular matrices. That is, find a lower triangular matrix L and an upper triangular matrix U , such that $A = LU$. Having obtained such a factorization (usually referred to as an LU decomposition or LU factorization), the solution to the original problem can be rewritten as follows.

$$\begin{aligned} Ax &= b \\ \Rightarrow LUx &= b \\ \Rightarrow (Ux) &= b \end{aligned}$$

This leads to the following two-step process for finding the solution to the original system of equations:

1. Solve the systems of equations $Ly = b$.
2. Solve the system $Ux = y$.

Solving the systems $Ly = b$ and $Ux = y$ is referred to as a forward solve and a backward solve, respectively.

If a symmetric matrix A is also positive definite, it can be shown that A can be factored as LL^T where L is a lower triangular matrix. Similarly, a Hermitian matrix, A , that is positive definite can be factored as $A = LL^H$. For both symmetric and Hermitian matrices, a factorization of this form is called a Cholesky factorization.

In a Cholesky factorization, the matrix U in an LU decomposition is either L^T or L^H . Consequently, a solver can increase its efficiency by only storing L , and one-half of A , and not computing U . Therefore, users who can express their application as the solution of a system of positive definite equations will gain a significant performance improvement over using a general representation.

For matrices that are symmetric (or Hermitian) but not positive definite, there are still some significant efficiencies to be had. It can be shown that if A is symmetric but not positive definite, then A can be factored as $A = LDL^T$, where D is a diagonal matrix and L is a lower unit triangular matrix. Similarly, if A is Hermitian, it can be factored as $A = LDL^H$. In either case, we again only need to store L , D , and half of A and we need not compute U . However, the backward solve phases must be amended to solving $L^T x = D^{-1} y$ rather than $L^T x = y$.

Fill-In and Reordering of Sparse Matrices

Two important concepts associated with the solution of sparse systems of equations are fill-in and reordering. The following example illustrates these concepts.

Consider the system of linear equation $Ax = b$, where A is the symmetric positive definite sparse matrix defined by the following:

$$A = \begin{bmatrix} 9 & \frac{3}{2} & 6 & \frac{3}{4} & 3 \\ \frac{3}{2} & 1 & * & * & * \\ 2 & 2 & * & * & * \\ 6 & * & 12 & * & * \\ \frac{3}{4} & * & \frac{5}{8} & * & * \\ 3 & * & * & * & 16 \end{bmatrix} b = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

A star (*) is used to represent zeros and to emphasize the sparsity of A . The Cholesky factorization of A is: $A = LL^T$, where L is the following:

$$L = \begin{bmatrix} 3 & * & * & * & * \\ \frac{1}{2} & \frac{1}{2} & * & * & * \\ 2 & -2 & 2 & * & * \\ \frac{1}{4} & \frac{1}{-4} & \frac{1}{-2} & \frac{1}{2} & * \\ 1 & -1 & -2 & -3 & 1 \end{bmatrix}$$

Notice that even though the matrix A is relatively sparse, the lower triangular matrix L has no zeros below the diagonal. If we computed L and then used it for the forward and backward solve phase, we would do as much computation as if A had been dense.

The situation of L having non-zeros in places where A has zeros is referred to as fill-in. Computationally, it would be more efficient if a solver could exploit the non-zero structure of A in such a way as to reduce the fill-in when computing L . By doing this, the solver would only need to compute the non-zero entries in L . Toward this end, consider permuting the rows and columns of A . As described in [Matrix Fundamentals](#) section, the permutations of the rows of A can be represented as a permutation matrix, P . The result of permuting the rows is the product of P and A . Suppose, in the above example, we swap the first and fifth row of A , then swap the first and fifth columns of A , and call the resulting matrix B . Mathematically, we can express the process of permuting the rows and columns of A to get B as $B = PAP^T$. After permuting the rows and columns of A , we see that B is given by the following:

$$B = \begin{bmatrix} 16 & * & * & * & 3 \\ * & \frac{1}{2} & * & * & \frac{3}{2} \\ * & * & 12 & * & 6 \\ * & * & * & \frac{5}{8} & \frac{3}{4} \\ 3 & \frac{3}{2} & 6 & \frac{3}{4} & 9 \end{bmatrix}$$

Since B is obtained from A by simply switching rows and columns, the numbers of non-zero entries in A and B are the same. However, when we find the Cholesky factorization, $B = LL^T$, we see the following:

$$L = \begin{bmatrix} 4 & * & * & * & * \\ * & \frac{1}{\sqrt{2}} & * & * & * \\ * & * & 2(\sqrt{3}) & * & * \\ * & * & * & \frac{\sqrt{10}}{4} & * \\ \frac{3}{4} & \frac{3}{\sqrt{2}} & \sqrt{3} & \frac{3}{\sqrt{10}} & \frac{\sqrt{3}}{4} \end{bmatrix}$$

The fill-in associated with B is much smaller than the fill-in associated with A . Consequently, the storage and computation time needed to factor B is much smaller than to factor A . Based on this, we see that an efficient sparse solver needs to find permutation P of the matrix A , which minimizes the fill-in for factoring $B = PAP^T$, and then use the factorization of B to solve the original system of equations.

Although the above example is based on a symmetric positive definite matrix and a Cholesky decomposition, the same approach works for a general LU decomposition. Specifically, let P be a permutation matrix, $B = PAP^T$ and suppose that B can be factored as $B = LU$. Then

$$Ax = b$$

$$\Rightarrow PA(P^{-1}P)x = Pb$$

$$\Rightarrow PA(P^TP)x = Pb$$

$$\Rightarrow (PAP^T)(Px) = Pb$$

$$\Rightarrow B(Px) = Pb$$

$$\Rightarrow LU(Px) = Pb$$

It follows that if we obtain an LU factorization for B , we can solve the original system of equations by a three step process:

- 1.** Solve $Ly = Pb$.
- 2.** Solve $Uz = y$.
- 3.** Set $x = P^Tz$.

If we apply this three-step process to the current example, we first need to perform the forward solve of the systems of equation $Ly = Pb$:

$$L = \begin{bmatrix} 4 & * & * & * & * \\ * & \frac{1}{\sqrt{2}} & * & * & * \\ * & * & 2(\sqrt{3}) & * & * \\ * & * & * & \frac{\sqrt{10}}{4} & * \\ \frac{3}{4} & \frac{3}{\sqrt{2}} & \sqrt{3} & \frac{3}{\sqrt{10}} & \frac{\sqrt{3}}{4} \end{bmatrix} * \begin{bmatrix} y1 \\ y2 \\ y3 \\ y4 \\ y5 \end{bmatrix} = \begin{bmatrix} 5 \\ 2 \\ 3 \\ 4 \\ 1 \end{bmatrix}$$

This gives:

$$Y^T = \frac{5}{4}, 2\sqrt{2}, \frac{\sqrt{3}}{2}, \frac{16}{\sqrt{10}}, \frac{-979\sqrt{3}}{12}.$$

The second step is to perform the backward solve, $Uz = y$. Or, in this case, since we are using a Cholesky factorization, $L^T z = y$.

$$\begin{bmatrix}
 4 & * & * & * & \frac{3}{4} \\
 * & \frac{1}{\sqrt{2}} & * & * & \frac{3}{\sqrt{2}} \\
 * & * & 2(\sqrt{3}) & * & \sqrt{3} \\
 * & * & * & \frac{\sqrt{10}}{4} & \frac{3}{\sqrt{10}} \\
 \frac{3}{4} & \frac{3}{\sqrt{2}} & \sqrt{3} & \frac{3}{\sqrt{10}} & \frac{\sqrt{3}}{4}
 \end{bmatrix} * \begin{bmatrix} z1 \\ z2 \\ z3 \\ z4 \\ z5 \end{bmatrix} = \begin{bmatrix} \frac{5}{4} \\ 2(\sqrt{2}) \\ \frac{\sqrt{3}}{2} \\ \frac{16}{\sqrt{10}} \\ \frac{-979\sqrt{3}}{12} \end{bmatrix}$$

This gives

$$z = \frac{123}{2}, 983, \frac{1961}{12}, 398, \frac{-979}{3}.$$

The third and final step is to set $x = P^T z$. This gives

$$X^T = \frac{-979}{3}, 983, \frac{1961}{12}, 398, \frac{123}{2}.$$

Sparse Matrix Storage Formats

As discussed above, it is more efficient to store only the non-zero elements of a sparse matrix. There are a number of common storage formats used for sparse matrices, but most of them employ the same basic technique. That is, store all non-zero elements of the matrix into a linear array and provide auxiliary arrays to describe the locations of the non-zero elements in the original matrix.

Storage Formats for the Direct Sparse Solvers

The storing the non-zero elements of a sparse matrix into a linear array is done by walking down each column (column-major format) or across each row (row-major format) in order, and writing the non-zero elements to a linear array in the order they appear in the walk.

For symmetric matrices, it is necessary to store only the upper triangular half of the matrix (upper triangular format) or the lower triangular half of the matrix (lower triangular format).

The Intel MKL direct sparse solvers use a row-major upper triangular storage format: the matrix is compressed row-by-row and for symmetric matrices only non-zero elements in the upper triangular half of the matrix are stored.

The Intel MKL sparse matrix storage format for direct sparse solvers is specified by three arrays: *values*, *columns*, and *rowIndex*. The following table describes the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix.

<i>values</i>	A real or complex array that contains the non-zero elements of a sparse matrix. The non-zero elements are mapped into the <i>values</i> array using the row-major upper triangular storage mapping described above.
<i>columns</i>	Element <i>I</i> of the integer array <i>columns</i> is the number of the column that contains the <i>I</i> -th element in the <i>values</i> array.
<i>rowIndex</i>	Element <i>j</i> of the integer array <i>rowIndex</i> gives the index of the element in the <i>values</i> array that is first non-zero element in a row <i>j</i> .

The length of the *values* and *columns* arrays is equal to the number of non-zero elements in the matrix.

As the *rowIndex* array gives the location of the first non-zero element within a row, and the non-zero elements are stored consecutively, the number of non-zero elements in the *I*-th row is equal to the difference of *rowIndex*(*I*) and *rowIndex*(*I*+1).

To have this relationship hold for the last row of the matrix, an additional entry (dummy entry) is added to the end of *rowIndex*. Its value is equal to the number of non-zero elements plus one. This makes the total length of the *rowIndex* array one larger than the number of rows in the matrix.



NOTE. The Intel MKL sparse storage scheme for the direct sparse solvers uses the Fortran programming language convention of starting array indices at 1.

Consider the symmetric matrix *A*:

$$A = \begin{bmatrix} 1 & -1 & * & -3 & * \\ -1 & 5 & * & * & * \\ * & * & 4 & 6 & 4 \\ -3 & * & 6 & 7 & * \\ * & * & 4 & * & -5 \end{bmatrix}$$

Only elements from the upper triangle are stored. The actual arrays for the matrix *A* are as follows:

Table A-1 Storage Arrays for a Symmetric Matrix

<i>values</i>	=	(1	-1	-3	5	4	6	4	7	-5)	
<i>columns</i>	=	(1	2	4	2	3	4	5	4	5)	
<i>rowIndex</i>	=	(1	4	5	8	9	10)				

For a non-symmetric or non-Hermitian matrix, all non-zero elements need to be stored. Consider the non-symmetric matrix *B*:

$$B = \begin{bmatrix} 1 & -1 & * & -3 & * \\ -2 & 5 & * & * & * \\ * & * & 4 & 6 & 4 \\ -4 & * & 2 & 7 & * \\ * & 8 & * & * & -5 \end{bmatrix}$$

The matrix *B* has 13 non-zero elements, and all of them are stored as follows:

Table A-2 Storage Arrays for a Non-Symmetric Matrix

<i>values</i>	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
<i>columns</i>	=	(1	2	4	1	2	3	4	5	1	3	4	2	5)
<i>rowIndex</i>	=	(1	4	6	9	12	14)							

Direct sparse solvers can also solve symmetrically structured systems of equations. A symmetrically structured system of equations is one where the pattern of non-zero elements is symmetric. That is, a matrix has a symmetric structure if *a*(*j*,*i*) is not zero if and only if *a*(*j*,*i*) is not zero. From the point of view of the solver software, a "non-zero" element of a matrix

is any element stored in the *values* array, even if its value is equal to 0. In that sense, any non-symmetric matrix can be turned into a symmetrically structured matrix by carefully adding zeros to the *values* array. For example, the above matrix *B* can be turned into a symmetrically structured matrix by adding two non-zero entries:

$$B = \begin{bmatrix} 1 & -1 & * & -3 & * \\ -2 & 5 & * & * & 0 \\ * & * & 4 & 6 & 4 \\ -4 & * & 2 & 7 & * \\ * & 8 & 0 & * & -5 \end{bmatrix}$$

The matrix *B* can be considered to be symmetrically structured with 15 non-zero elements and represented as:

Table A-3 Storage Arrays for a Symmetrically Structured Matrix

<i>values</i>	=	(1	-1	-3	-2	5	0	4	6	4	-4	2	7	8	0	-5)
<i>columns</i>	=	(1	2	4	1	2	5	3	4	5	1	3	4	2	3	5)
<i>rowIndex</i>	=	(1	4	7	10	13	16)									

Storage Format Restrictions

- The storage format for the sparse solver must conform to two important restrictions:
- the non-zero values in a given row must be placed into the *values* array in the order in which they occur in the row (from left to right);
 - no diagonal element can be omitted from the *values* array for any symmetric or structurally symmetric matrix.

The second restriction implies that if symmetric or structurally symmetric matrices have zero diagonal elements, then they must be explicitly represented in the *values* array.

Sparse Matrix Storage Formats for Sparse BLAS Levels 2 and Level 3

This section describes in detail the sparse matrix storage formats supported in the current version of the Intel MKL Sparse BLAS Level 2 and Level 3.

CSR Format

The Intel MKL compressed sparse row (CSR) format is specified by four arrays: the *values*, *columns*, *pointerB*, and *pointerE*. The following table describes the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix *A*.

<i>values</i>	A real or complex array that contains the non-zero elements of <i>A</i> . Values of the non-zero elements of <i>A</i> are mapped into the <i>values</i> array using the row-major storage mapping described above.
<i>columns</i>	Element <i>I</i> of the integer array <i>columns</i> is the number of the column in <i>A</i> that contains the <i>I</i> -th value in the <i>values</i> array.
<i>pointerB</i>	Element <i>j</i> of this integer array gives the index of the element in the <i>values</i> array that is first non-zero element in a row <i>j</i> of <i>A</i> . Note that this index is equal to <i>pointerB(j) - pointerB(1)+1</i> .
<i>pointerE</i>	An integer array that contains row indices, such that <i>pointerE(j)-pointerB(1)</i> is the index of the element in the <i>values</i> array that is last non-zero element in a row <i>j</i> of <i>A</i> .

The length of the *values* and *columns* arrays is equal to the number of non-zero elements in *A*. The length of the *pointerB* and *pointerE* arrays is equal to the number of rows in *A*.



NOTE. Note that the Intel MKL Sparse BLAS routines support the CSR format both with one-based indexing and zero-based indexing.

The matrix *B*

$$B = \begin{bmatrix} 1 & -1 & * & -3 & * \\ -2 & 5 & * & * & * \\ * & * & 4 & 6 & 4 \\ -4 & * & 2 & 7 & * \\ * & 8 & * & * & -5 \end{bmatrix}$$

can be represented in the CSR format as:

Table A-4 Storage Arrays for a Matrix in CSR Format

one-based indexing														
values	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
columns	=	(1	2	4	1	2	3	4	5	1	3	4	2	5)
pointerB	=	(1	4	6	9	12)								
pointerE	=	(4	6	9	12	14)								
zero-based indexing														
values	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
columns	=	(0	1	3	0	1	2	3	4	0	2	3	1	4)
pointerB	=	(0	3	5	8	11)								
pointerE	=	(3	5	8	11	13)								

This storage format is used in the NIST Sparse BLAS library [[Rem05](#)].

Note that the storage format accepted for the direct sparse solvers and described above (see [Storage Formats for the Direct Sparse Solvers](#)) is a variation of the CSR format. It also is used in the Intel MKL Sparse BLAS Level 2 both with one-based indexing and zero-based indexing. The above matrix *B* can be represented in this format (referred to as the 3-array variation of the CSR format) as:

Table A-5 Storage Arrays for a Matrix in CSR Format (3-Array Variation)

one-based indexing														
values	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
columns	=	(1	2	4	1	2	3	4	5	1	3	4	2	5)
rowIndex	=	(1	4	6	9	12	14)							
zero-based indexing														
values	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
columns	=	(0	1	3	0	1	2	3	4	0	2	3	1	4)
rowIndex	=	(0	3	5	8	11	13)							

The 3-array variation of the CSR format has a restriction: all non-zero elements are stored continuously, that is the set of non-zero elements in the row *J* goes just after the set of non-zero elements in the row *J-1*.

There are no such restrictions in the general (NIST) CSR format. This may be useful, for example, if there is a need to operate with different submatrices of the matrix at the same time. In this case, it is enough to define the arrays *pointerB* and *pointerE* for each needed submatrix so that all these arrays are pointers to the same array *values*.

Comparing the array *rowIndex* from the Table A-2 with the arrays *pointerB* and *pointerE* from the Table A-4 it is easy to see that

pointerB(I) = rowIndex(I) for I=1, ..5;

pointerE(I) = rowIndex(I+1) for I=1, ..5.

This enables calling a routine that has *values*, *columns*, *pointerB* and *pointerE* as input parameters for a sparse matrix stored in the format accepted for the direct sparse solvers. For example, a routine with the interface:

```
Subroutine name_routine(... , values, columns, pointerB, pointerE, ...)
```

can be called with parameters *values*, *columns*, *rowIndex* as follows:

```
call name_routine(... , values, columns, rowIndex, rowindex(2), ...).
```

CSC Format

The compressed sparse column format (CSC) is similar to the CSR format, but the columns are used instead the rows. In other words, the CSC format is identical to the CSR format for the transposed matrix. The CSR format is specified by four arrays: *values*, *columns*, *pointerB*, and *pointerE*. The following table describes the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix *A*.

<i>values</i>	A real or complex array that contains the non-zero elements of <i>A</i> . Values of the non-zero elements of <i>A</i> are mapped into the <i>values</i> array using the column-major storage mapping.
<i>rows</i>	Element <i>I</i> of the integer array <i>rows</i> is the number of the row in <i>A</i> that contains the <i>I</i> -th value in the <i>values</i> array.
<i>pointerB</i>	Element <i>j</i> of this integer array gives the index of the element in the <i>values</i> array that is first non-zero element in a column <i>j</i> of <i>A</i> . Note that this index is equal to <i>pointerB(j) - pointerB(1)+1</i> .
<i>pointerE</i>	An integer array that contains column indices, such that <i>pointerE(j)-pointerB(1)</i> is the index of the element in the <i>values</i> array that is last non-zero element in a column <i>j</i> of <i>A</i> .

The length of the *values* and *columns* arrays is equal to the number of non-zero elements in *A*. The length of the *pointerB* and *pointerE* arrays is equal to the number of columns in *A*.



NOTE. Note that the Intel MKL Sparse BLAS routines support the CSC format both with one-based indexing and zero-based indexing.

The above matrix *B* can be represented in the CSC format as:

Table A-6 Storage Arrays for a Matrix in CSC Format

one-based indexing														
values	=	(1	-2	-4	-1	5	8	4	2	-3	6	7	4	-5)
rows	=	(1	2	4	1	2	5	3	4	1	3	4	2	5)
pointerB	=	(1	4	7	9	12)								
pointerE	=	(4	7	9	12	14)								
zero-based indexing														
values	=	(1	-2	-4	-1	5	8	4	2	-3	6	7	4	-5)
rows	=	(0	1	3	0	1	4	2	3	0	2	3	1	4)
pointerB	=	(0	3	6	8	11)								
pointerE	=	(3	6	8	11	13)								

Coordinate Format

The coordinate format is the most flexible and simplest format for the sparse matrix representation. Only non-zero elements are stored, and the coordinates of each non-zero element are given explicitly. Many commercial libraries support the matrix-vector multiplication for the sparse matrices in the coordinate format.

The Intel MKL coordinate format is specified by three arrays: *values*, *rows*, and *column*, and a parameter *nnz* which is number of non-zero elements in *A*. All three arrays have dimension *nnz*. The following table describes the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix *A*.

<i>values</i>	A real or complex array that contains the non-zero elements of <i>A</i> in any order.
<i>rows</i>	Element <i>I</i> of the integer array <i>rows</i> is the number of the row in <i>A</i> that contains the <i>I</i> -th value in the <i>values</i> array.
<i>columns</i>	Element <i>I</i> of the integer array <i>columns</i> is the number of the column in <i>A</i> that contains the <i>I</i> -th value in the <i>values</i> array .



NOTE. Note that the Intel MKL Sparse BLAS routines support the coordinate format both with one-based indexing and zero-based indexing.

For example, the sparse matrix *c*

$$c = \begin{bmatrix} 1 & -1 & -3 & 0 & 0 \\ -2 & 5 & 0 & 0 & 0 \\ 0 & 0 & 4 & 6 & 4 \\ -4 & 0 & 2 & 7 & 0 \\ 0 & 8 & 0 & 0 & -5 \end{bmatrix}$$

can be represented in the coordinate format as follows:

Table A-7 Storage Arrays for an Example Matrix in case of the coordinate format

one-based indexing														
<i>values</i>	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
<i>rows</i>	=	(1	1	1	2	2	3	3	3	4	4	4	5	5)
<i>columns</i>	=	(1	2	3	1	2	3	4	5	1	3	4	2	5)
zero-based indexing														
<i>values</i>	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
<i>rows</i>	=	(0	0	0	1	1	2	2	2	3	3	3	4	4)
<i>columns</i>	=	(0	1	2	0	1	2	3	4	0	2	3	1	4)

Diagonal Storage Format

If the sparse matrix has diagonals containing only zero elements, then the diagonal storage format can be used to reduce the amount of information needed to locate the non-zero elements. This storage format is particularly useful in many applications where the matrix arises from a finite element or finite difference discretization. The Intel MKL diagonal storage format is specified by two arrays: *values* and *distance*, and two parameters: *ndiag*, which is the number of non-empty diagonals, and *lval*, which is the declared leading dimension in the calling (sub)programs. The following table describes the arrays *values* and *distance*:

values

A real or complex two-dimensional array is dimensioned as *lval* by *ndiag*. Each column of it contains the non-zero elements of certain diagonal of *A*. The key point of the storage is that each element in *values* retains the row number of the original matrix. To achieve this diagonals in the lower triangular part of the matrix are padded from the top, and those in the upper triangular part are padded from the bottom. Note that the value of *distance(I)* is the number of elements to be padded for diagonal *I*.

distance

An integer array with dimension *ndiag*. Element *I* of the array *distance* is the distance between *I*-diagonal and the main diagonal. The distance is positive if the diagonal is above the main diagonal, and negative if the diagonal is below the main diagonal. The main diagonal has a distance equal to zero.

The above matrix *C* can be represented in the diagonal storage format as follows:

$$\begin{aligned}
 &\text{distance} = (-3 \quad -1 \quad 0 \quad 1 \quad 2) \\
 &\text{values} = \begin{bmatrix} * & * & 1 & -1 & -3 \\ * & -2 & 5 & 0 & 0 \\ * & 0 & 4 & 6 & 4 \\ -4 & 2 & 7 & 0 & * \\ 8 & 0 & -5 & * & * \end{bmatrix}
 \end{aligned}$$

where the asterisks denote padded elements.

When storing symmetric, Hermitian, or skew-symmetric matrices, it is necessary to store only the upper or the lower triangular part of the matrix.

For the Intel MKL triangular solver routines elements of the array *distance* must be sorted in increasing order. In all other cases the diagonals and distances can be stored in arbitrary order.

Skyline Storage Format

The skyline storage format is important for the direct sparse solvers, and it is well suited for Cholesky or LU decomposition when no pivoting is required.

The skyline storage format accepted in Intel MKL can store only triangular matrix or triangular part of a matrix. This format is specified by two arrays: *values* and *pointers*. The following table describes these arrays:

<i>values</i>	A scalar array. For a lower triangular matrix it contains the set of elements from each row of the matrix starting from the first non-zero element to and including the diagonal element. For an upper triangular matrix it contains the set of elements from each column of the matrix starting with the first non-zero element down to and including the diagonal element. Encountered zero elements are included in the sets.
<i>pointers</i>	An integer array with dimension $(m+1)$, where m is the number of rows for lower triangle (columns for the upper triangle). $pointers(I) - pointers(1)+1$ gives the index of element in <i>values</i> that is first non-zero element in row (column) I . The value of $pointers(m+1)$ is set to $nz+pointers(1)$, where nnz is the number of elements in the array <i>values</i> .

For example, the low triangle of the matrix C given above can be stored as follows:

```
values = ( 1  -2  5  4  -4  0  2  7  8  0  0  -5 )
```

```
pointers = ( 1  2  4  5  9  13 )
```

and the upper triangle of this matrix C can be stored as follows:

```
values = ( 1  -1  5  -3  0  4  6  7  4  0  -5 )
```

```
pointers = ( 1  2  4  7  9  12 )
```

This storage format is supported by the NIST Sparse BLAS library [[Rem05](#)].

Note that the Intel MKL Sparse BLAS routines operating with the skyline storage format does not support general matrices.

BSR Format

The Intel MKL block compressed sparse row (BSR) format for sparse matrices is specified by four arrays: *values*, *columns*, *pointerB*, and *pointerE*. The following table describes these arrays.

<i>values</i>	A real array that contains the elements of the non-zero blocks of a sparse matrix. The elements are stored block-by-block in row-major order. A non-zero block is the block that contains at least one non-zero element. All elements of non-zero blocks are stored, even if some of

	them is equal to zero. Within each non-zero block elements are stored in column-major order in the case of one-based indexing, and in row-major order in the case of the zero-based indexing.
<i>columns</i>	Element <i>i</i> of the integer array <i>columns</i> is the number of the column in the block matrix that contains the <i>i</i> -th non-zero block.
<i>pointerB</i>	Element <i>j</i> of this integer array gives the index of the element in the <i>columns</i> array that is first non-zero block in a row <i>j</i> of the block matrix.
<i>pointerE</i>	Element <i>j</i> of this integer array gives the index of the element in the <i>columns</i> array that contains the last non-zero block in a row <i>j</i> of the block matrix plus 1.

The length of the *values* array is equal to the number of all elements in the non-zero blocks, the length of the *columns* array is equal to the number of non-zero blocks. The length of the *pointerB* and *pointerE* arrays is equal to the number of block rows in the block matrix.



NOTE. Note that the Intel MKL Sparse BLAS routines support BSR format both with one-based indexing and zero-based indexing.

For example, consider the sparse matrix *D*

$$D = \begin{bmatrix} 1 & 0 & 6 & 7 & * & * \\ 2 & 1 & 8 & 2 & * & * \\ * & * & 1 & 4 & * & * \\ * & * & 5 & 2 & * & * \\ * & * & 4 & 3 & 7 & 2 \\ * & * & 0 & 0 & 0 & 0 \end{bmatrix}$$

If the size of the block equals 2, then the sparse matrix *D* can be represented as a 3x3 block matrix *E* with the following structure:

$$E = \begin{bmatrix} L & M & * \\ * & N & * \\ * & P & Q \end{bmatrix}$$

where

$$L = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}, M = \begin{bmatrix} 6 & 7 \\ 8 & 2 \end{bmatrix}, N = \begin{bmatrix} 1 & 4 \\ 5 & 1 \end{bmatrix}, P = \begin{bmatrix} 4 & 3 \\ 0 & 0 \end{bmatrix}, Q = \begin{bmatrix} 7 & 2 \\ 0 & 0 \end{bmatrix}$$

The matrix D can be represented in the BSR format as follows:

one-based indexing

`values = (1 2 0 1 6 8 7 2 1 5 4 1 4 0 3 0 7 0 2 0)`

`columns = (1 2 2 2 3)`

`pointerB = (1 3 4)`

`pointerE = (3 4 6)`

zero-based indexing

`values = (1 0 2 1 6 7 8 2 1 4 5 1 4 3 0 0 7 2 0 0)`

`columns = (0 1 1 1 2)`

`pointerB = (0 2 3)`

`pointerE = (2 3 5)`

This storage format is supported by the NIST Sparse BLAS library [[Rem05](#)].

Intel MKL supports the variation of the BSR format that is specified by three arrays: `values`, `columns`, and `rowIndex`. The following table describes these arrays.

<code>values</code>	A real array that contains the elements of the non-zero blocks of a sparse matrix. The elements are stored block by block in row-major order. A non-zero block is the block that contains at least one non-zero element. All elements of non-zero blocks are stored, even if some of them is equal to zero. Within each non-zero block the elements are stored in column major order in the case of the one-based indexing, and in row major order in the case of the zero-based indexing.
<code>columns</code>	Element i of the integer array <code>columns</code> is the number of the column in the block matrix that contains the i -th non-zero block.
<code>rowIndex</code>	Element j of this integer array gives the index of the element in the <code>columns</code> array that is first non-zero block in a row j of the block matrix.

The length of the `values` array is equal to the number of all elements in the non-zero blocks, the length of the `columns` array is equal to the number of non-zero blocks.

As the *rowIndex* array gives the location of the first non-zero block within a row, and the non-zero blocks are stored consecutively, the number of non-zero blocks in the *I*-th row is equal to the difference of *rowIndex*(*I*) and *rowIndex*(*I*+1).

To retain this relationship for the last row of the block matrix, an additional entry (dummy entry) is added to the end of *rowIndex* with value equal to the number of non-zeros blocks plus one. This makes the total length of the *rowIndex* array one larger than the number of rows of the block matrix.

The above matrix *D* can be represented in this 3-array variation of the BSR format as follows:

one-based indexing

values = (1 2 0 1 6 8 7 2 1 5 4 2 4 0 3 0 7 0 2 0)

columns = (1 2 2 2 3)

rowIndex = (1 3 4 6)

zero-based indexing

values = (1 0 2 1 6 7 8 2 1 4 5 1 4 3 0 0 7 2 0 0)

columns = (0 1 1 1 2)

rowIndex = (0 2 3 5)

When storing symmetric matrices, it is necessary to store only the upper or the lower triangular part of the matrix.

For example, consider the symmetric sparse matrix *F*:

$$F = \begin{bmatrix} 1 & 0 & 6 & 7 & * & * \\ 2 & 1 & 8 & 2 & * & * \\ 6 & 8 & 1 & 4 & * & * \\ 7 & 2 & 5 & 2 & * & * \\ * & * & * & * & 7 & 2 \\ * & * & * & * & 0 & 0 \end{bmatrix}$$

If the size of the block equals 2, then the sparse matrix *F* can be represented as a 3x3 block matrix *G* with the following structure:

$$G = \begin{bmatrix} L & M & * \\ M' & N & * \\ * & * & Q \end{bmatrix}$$

where

$$L = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}, M = \begin{bmatrix} 6 & 7 \\ 8 & 2 \end{bmatrix}, M' = \begin{bmatrix} 6 & 8 \\ 7 & 2 \end{bmatrix}, N = \begin{bmatrix} 1 & 4 \\ 5 & 2 \end{bmatrix}, Q = \begin{bmatrix} 7 & 2 \\ 0 & 0 \end{bmatrix}$$

The symmetric matrix F can be represented in this 3-array variation of the BSR format (storing only upper triangular) as follows:

one-based indexing

`values = (1 2 0 1 6 8 7 2 1 5 4 2 7 0 2 0)`

`columns = (1 2 2 3)`

`rowIndex = (1 3 4 5)`

zero-based indexing

`values = (1 0 2 1 6 7 8 2 1 4 5 2 7 2 0 0)`

`columns = (0 1 1 2)`

`rowIndex = (0 2 3 4)`

Interval Linear Systems

Intervals

An interval is a compact connected subset of the real axis $\mathbf{R}$. It is thus completely defined by two numbers, namely, its *lower endpoint* and *upper endpoint* (sometimes called *left endpoint* and *right endpoint* respectively), so that $[a, b]$ denotes the interval $\{x \in \mathbf{R} \mid a \leq x \leq b\}$. The set of all real intervals is denoted by $\mathbf{IR}$. In mathematical notation, taking the lower and upper endpoints of an interval is usually denoted by

$\inf[a, b] = a, \sup[a, b] = b.$

In the discussion below, intervals and interval objects are denoted by boldface letters, while underscores and overscores designate the lower and upper endpoints of the interval $\mathbf{x} = [\underline{\mathbf{x}}, \overline{\mathbf{x}}]$

Every interval is uniquely determined by its *midpoint*,

$$\text{mid } \mathbf{a} = \frac{1}{2} (\bar{\mathbf{a}} + \underline{\mathbf{a}}),$$

and radius,

$$\text{rad } \mathbf{a} = \frac{1}{2} (\bar{\mathbf{a}} - \underline{\mathbf{a}})$$

the latter being equivalent to the width $\text{wid } \mathbf{a} = \bar{\mathbf{a}} - \underline{\mathbf{a}}$. Intervals of the form $[\mathbf{a}, \mathbf{a}]$ that have equal lower and upper endpoints, that is, intervals of zero width, are called *degenerate* or *point* or *thin*, and they coincide with usual real numbers so that it can be implied $\mathbf{R} \subset \mathbf{IR}$. On the contrary, the intervals with nonzero width are called *thick* intervals.

Since intervals are sets, set-theoretical relations and operations between them are applicable, for example, inclusion, intersection, and so on. In particular, a point $t \in \mathbf{R}$ is a member of the interval $\mathbf{a}$ (written as $t \in \mathbf{a}$) if $\underline{\mathbf{a}} \leq t \leq \bar{\mathbf{a}}$. Also, the inclusion is defined as $\mathbf{a} \subseteq \mathbf{b}$ if and only if $\underline{\mathbf{a}} \geq \underline{\mathbf{b}}$ and $\bar{\mathbf{a}} \leq \bar{\mathbf{b}}$.

Intervals and interval objects (vectors, matrices, etc.) are a convenient tool to represent the so-called *bounded* uncertainty and ambiguity, when only the lower and upper bounds of the possible variation of some value are known. In this sense, intervals provide an alternative to probabilistic and fuzzy approaches for describing quantitative uncertainty.

Arithmetic operations, such as addition, subtraction, multiplication and division, can be extended to intervals according to the fundamental principle

$$\mathbf{a} * \mathbf{b} = \{a * b \mid a \in \mathbf{a}, b \in \mathbf{b}\}, \quad * \in \{+, -, \cdot, /\}, \quad (1)$$

which makes it possible to define the so-called *classical interval arithmetic*. Note that the empty interval $[\emptyset]$ is often incorporated into the computer interval arithmetic structures.

Interval vectors and matrices

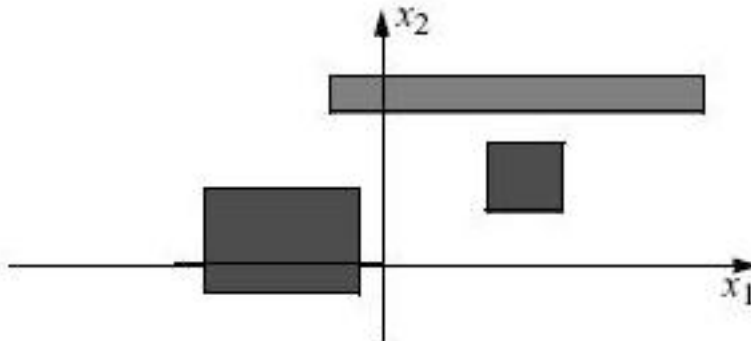
An interval vector is an ordered tuple of intervals placed vertically(column vector) or horizontally(row vector). So, if $a_1, a_2, \dots, a_n$ are intervals, then

$$a = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} \text{ is a column vector,}$$

and

$a_1, a_2, \dots, a_n$ is a row vector.

The set of all interval n -vectors is denoted later in the text by $\mathbf{IR}^n$.



The interval vectors can be associated with their geometric images, namely rectangular boxes of the space $\mathbb{R}^n$, whose sides are parallel to the coordinate axes. For this reason, interval vectors are often called *boxes* for brevity.

An *interval matrix* is a rectangular table composed of the intervals:

$$\mathbf{A} := \begin{pmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \vdots \\ \mathbf{a}_m \end{pmatrix} = \begin{pmatrix} \mathbf{a}_{11} & \mathbf{a}_{12} & \dots & \mathbf{a}_{1n} \\ \mathbf{a}_{21} & \mathbf{a}_{22} & \dots & \mathbf{a}_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{a}_{m1} & \mathbf{a}_{m2} & \dots & \mathbf{a}_{mn} \end{pmatrix},$$

or $\mathbf{A} = (\mathbf{a}_{ij})$. Interval vectors can be identified with interval matrices either of the size $n \times 1$ (column vectors) or $1 \times n$ (row vectors). The set of all interval $m \times n$ -matrices is denoted by $\mathbf{IR}^{m \times n}$. Arithmetic operations between interval vectors and matrices can be introduced on the basis of the relation that generalizes (1) (see [Alefeld83], [Neumaier90]).

An interval square matrix $\mathbf{A} \in \mathbf{IR}^{n \times n}$ is referred to as *regular* (nonsingular) if and only if all the point matrices $\mathbf{A} \in \mathbf{A}$ are regular (nonsingular), that is, have nonzero determinants.

Otherwise, the interval matrix $\mathbf{A} \in \mathbf{IR}^{n \times n}$ is called *singular*, which means that it contains at least one singular point matrix.

Generally, recognition of whether an interval matrix is regular or singular is an NP-hard problem, which implies that there may be no relatively simple (polynomially complex) algorithms that completely solve the problem in a reasonable time.

For practical needs, it is important to have a set of workable sufficient criteria for testing regularity of a wide range of interval matrices. Intel MKL provides routines that implement Ris-Beeck spectral criterion, Rump singular value criterion, as well as Rohn-Rex singular value criterion for testing regularity/singularity of interval matrices.

Sometimes, a related property (called *strong regularity*) needs to be checked for interval matrices. Strong regularity requires that the product of the interval matrix by its midpoint inverse is regular. The routine `?gerbr` enables to check the strong regularity judging by the value of its output parameter `sr`.

Interval Linear Systems

Solving systems of linear algebraic equations of the form

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1, \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2, \\ \vdots \quad \quad \quad \ddots \quad \quad \quad \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n = b_m, \end{cases} \quad (2)$$

or, concisely,

$$Ax = b$$

with an $m \times n$ matrix A and a right-hand side m -vector b , is one of the key problems in science and engineering. If a_{ij} and b_i are not defined exactly but rather belong to known intervals $\mathbf{a}_{ij}$ and $\mathbf{b}_i$ respectively, the system is called an *interval linear system* and can be written as

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1, \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2, \\ \vdots \quad \quad \quad \ddots \quad \quad \quad \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n = b_m, \end{cases} \quad (3)$$

with intervals $\mathbf{a}_{ij}$ and $\mathbf{b}_i$, or in a short form as

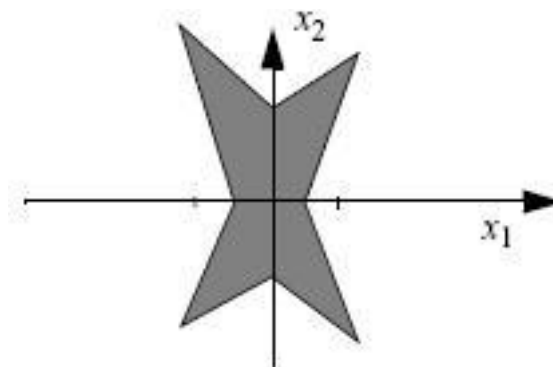
$$\mathbf{A}x = \mathbf{b} \quad (4)$$

with an interval matrix $\mathbf{A} = (\mathbf{a}_{ij})$ and interval right-hand side vector $\mathbf{b} = (\mathbf{b}_i)$. An interval linear system (3)–(4) is considered as a set of point linear systems of the same form $Ax = b$ with the parameters a_{ij} and b_i such that $a_{ij} \in \mathbf{a}_{ij}$ and $b_i \in \mathbf{b}_i$.

When a_{ij} and b_i are changing within intervals $\mathbf{a}_{ij}$ and $\mathbf{b}_i$, the solutions to the corresponding point systems $Ax = b$ with $A = (a_{ij})$ and $b = (b_i)$ form a set in the space $\mathbb{R}^n$, namely

$$\Xi(\mathbf{A}, \mathbf{b}) := \{x \in \mathbb{R}^n \mid (\exists A \in \mathbf{A}) (\exists b \in \mathbf{b}) (Ax = b)\}. \quad (5)$$

The set (5), made up of solutions to all the point systems $Ax = b$ with $A \in \mathbf{A}$ and $b \in \mathbf{b}$, is called a *solution set* to the interval linear system (3)–(4). Usually, the solution set is a solid polyhedron in $\mathbb{R}^n$ for independent a_{ij} and b_i , $1 \leq i, j \leq n$, sometimes star-shaped as in the figure below.



NOTE. The above described set $\Xi(\mathbf{A}, \mathbf{b})$ is often called a *untied solution set*, since there exist a variety of other solution sets to interval systems of equations (see [[Shary02](#)]).

An exact description of the solution set is practically impossible for dimensions n larger than several tens, since its complexity grows exponentially with n . On the other hand, such an exact description is not really necessary in most cases. Usually, one needs to compute some *estimates*, in a prescribed sense, of the solution set. The most popular in practice is the following problem of *outer* (by supersets) interval estimation:

For an interval system of linear equations $\mathbf{Ax} = \mathbf{b}$ (6)

find an interval enclosure of the solution set $\Xi(\mathbf{A}, \mathbf{b})$.

Frequently, a component-wise form of the problem (6) is considered:

For an interval system of linear equations $\mathbf{Ax} = \mathbf{b}$ (7)

find estimates for $\min\{x_v | x \in \Xi(\mathbf{A}, \mathbf{b})\}$ from below,

for $\max\{x_v | x \in \Xi(\mathbf{A}, \mathbf{b})\}$ from above, $v = 1, 2, \dots, n$.

In particular, Intel MKL [?gepps](#) routines operate with this type of the problem statement.

The problem (6)–(7) is one of the historically first and most popular in modern interval analysis. You can find an extensive bibliography on this problem, for example, in [\[Alefeld83\]](#), [\[Kearfott96\]](#), [\[Neumaier90\]](#).

Thus, solving an interval linear system is understood here as computing an outer interval estimate of the solution set to an interval linear system (3)–(4). The matrix $\mathbf{A}$ of the system is usually assumed to be square nonsingular.

Unlike classical computational linear algebra, solving interval linear systems proves to be very computationally hard in general. Computing the optimal (smallest) interval enclosures of the solution in (6), or, equivalently, computing exact estimates of the solution set in (7), is an NP-hard problem (see [\[Kreinovich97\]](#)), if there are no restrictions on the widths of the intervals in the system and/or the structure of nonzero elements in the matrix $\mathbf{A}$. Moreover, the problem remains NP-hard even if we weaken the requirements on the solution and compute estimates of the solution sets that must be precise to within a predetermined absolute or relative accuracy.

From the practical standpoint, NP-hardness means that with a high probability a general problem cannot be solved in polynomial time with respect to problem size.

For this reason, numerical algorithms employed in Intel MKL for solving interval linear systems are divided into two classes depending on whether or not they provide a guaranteed accuracy of the result. “Fast” algorithms work fast and compute an enclosure of the solution set in a reasonable time, but without any accuracy assumptions. “Optimal”, or “sharp” algorithms may take a lot of time to complete execution, but the results they obtain are less crude and may satisfy some accuracy requirements.

Intel MKL includes interval solver routines that implement algorithms of both types. For example, fast methods, such as interval Gauss method, interval Householder method, Hansen-Bliek-Rohn method, and Krawczyk iteration, are implemented in routines `?gegas`, `?gehss`, `?gehbs`, and `?gekws`, respectively. Parameter partitioning method (PPS-method) implemented in `?gepps` routine is an example of a sharp method. The routine `?trtrs` is subsumed under both categories due to a very special matrix structure.

Preconditioning

Preconditioning of interval linear system (4) is multiplying both the matrix $\mathbf{A}$ and the right-hand side vector $\mathbf{b}$ by a point matrix, with the intension to improve the properties of the system. So the system $\mathbf{Ax} = \mathbf{b}$ is substituted by the following system

$$(\mathbf{CA})\mathbf{x} = \mathbf{Cb}$$

where $\mathbf{C}$ is some square point matrix. Preconditioning is widely used in classical computational linear algebra, and many interval solver algorithms (for example, interval Gauss method, interval Gauss-Seidel method and some others) also require a suitable preconditioning prior to their use.

One of the widely used preconditioning methods for the interval linear systems is preconditioning done by the inverse of the midpoint matrix, often called *midpoint-inverse preconditioning*. In Intel MKL, the midpoint inverse preconditioning is implemented in the routine `?gemip`.

Inverting interval matrices

Given an interval square matrix $\mathbf{A}$, an enclosure for the set of all inverse point matrices in $\mathbf{A}$ is called the *inverse interval matrix* $\mathbf{A}^{-1}$, that is,

$$\mathbf{A}^{-1} \supseteq \{A^{-1} | A \in \mathbf{A}\}.$$

In classical linear algebra, the solution to a system of linear algebraic equations $\mathbf{Ax} = \mathbf{b}$ with square nonsingular matrix $\mathbf{A}$ can be expressed as the product of the inverse $\mathbf{A}^{-1}$ by the right-hand side vector, or $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$.

In interval analysis, the similar product $\mathbf{A}^{-1}\mathbf{b}$ also produces an enclosure for the solution set $\Xi\mathbf{A}, \mathbf{b}$ of the interval linear system $\mathbf{Ax} = \mathbf{b}$. However, this method usually causes substantial overestimation and is not recommended. Using specialized procedures for outer estimation of the solution sets is preferable.

Nevertheless, computing tight enclosures for inverse interval matrices is essential in sensitivity-like analysis of equation systems and the like.

Computing the inverse interval matrix may be carried out as finding an enclosure for the solution set of the following interval matrix equation

$\mathbf{A}\mathbf{Y} = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix,

by applying n times (for every column of the matrix $\mathbf{Y}$) any method to solve the interval linear systems.

Note also that direct iterative procedures for finding the inverse interval matrix exist, such as Schulz method (see [[Herzberger94](#)]), which is included into Intel MKL as `?geszi` routine.

Routine and Function Arguments

B

The major arguments in the BLAS routines are vector and matrix, whereas VML functions work on vector arguments only. The sections that follow discuss each of these arguments and provide examples.

Vector Arguments in BLAS

Vector arguments are passed in one-dimensional arrays. The array dimension (length) and vector increment are passed as integer variables. The length determines the number of elements in the vector. The increment (also called stride) determines the spacing between vector elements and the order of the elements in the array in which the vector is passed.

A vector of length n and increment $incx$ is passed in a one-dimensional array x whose values are defined as

$x(1), x(1+|incx|), \dots, x(1+(n-1)*|incx|)$

If $incx$ is positive, then the elements in array x are stored in increasing order. If $incx$ is negative, the elements in array x are stored in decreasing order with the first element defined as $x(1+(n-1)*|incx|)$. If $incx$ is zero, then all elements of the vector have the same value, $x(1)$. The dimension of the one-dimensional array that stores the vector must always be at least

$idimx = 1 + (n-1)*|incx|$

Example B-1 One-dimensional Real Array

Let $x(1:7)$ be the one-dimensional real array

$x = (1.0, 3.0, 5.0, 7.0, 9.0, 11.0, 13.0)$.

If $incx = 2$ and $n = 3$, then the vector argument with elements in order from first to last is $(1.0, 5.0, 9.0)$.

If $incx = -2$ and $n = 4$, then the vector elements in order from first to last is $(13.0, 9.0, 5.0, 1.0)$.

If $incx = 0$ and $n = 4$, then the vector elements in order from first to last is $(1.0, 1.0, 1.0, 1.0)$.

One-dimensional substructures of a matrix, such as the rows, columns, and diagonals, can be passed as vector arguments with the starting address and increment specified. In Fortran, storing the m -by- n matrix is based on column-major ordering where the increment between elements in the same column is 1, the increment between elements in the same row is m , and the increment between elements on the same diagonal is $m + 1$.

Example B-2 Two-dimensional Real Matrix

Let a be the real 5 x 4 matrix declared as `REAL A (5,4)`.

To scale the third column of a by 2.0, use the BLAS routine `sscal` with the following calling sequence:

```
callsscal (5, 2.0, a(1,3), 1)
```

To scale the second row, use the statement:

```
callsscal (4, 2.0, a(2,1), 5)
```

To scale the main diagonal of A by 2.0, use the statement:

```
callsscal (5, 2.0, a(1,1), 6)
```



NOTE. The default vector argument is assumed to be 1.

Vector Arguments in VML

Vector arguments of VML mathematical functions are passed in one-dimensional arrays with unit vector increment. It means that a vector of length n is passed contiguously in an array a whose values are defined as $a[0]$, $a[1]$, ..., $a[n-1]$ (for C- interface).

To accommodate for arrays with other increments, or more complicated indexing, VML contains auxiliary pack/unpack functions that gather the array elements into a contiguous vector and then scatter them after the computation is complete.

Generally, if the vector elements are stored in a one-dimensional array a as

```
a[m0], a[m1], ..., a[mn-1]
```

and need to be regrouped into an array y as

```
y[k0], y[k1], ..., y[kn-1],
```

VML pack/unpack functions can use one of the following indexing methods:

Positive Increment Indexing

```
kj = incy * j, mj = inca * j, j = 0, ..., n-1
```

Constraint: $incy > 0$ and $inca > 0$.

For example, setting $incy = 1$ specifies gathering array elements into a contiguous vector.

This method is similar to that used in BLAS, with the exception that negative and zero increments are not permitted.

Index Vector Indexing

$kj = iy[j], mj = ia[j], j = 0, \dots, n-1,$

where ia and iy are arrays of length n that contain index vectors for the input and output arrays a and y , respectively.

Mask Vector Indexing

Indices kj, mj are such that:

$my[kj] \neq 0, ma[mj] \neq 0, j = 0, \dots, n-1,$

where ma and my are arrays that contain mask vectors for the input and output arrays a and y , respectively.

Matrix Arguments

Matrix arguments of the Intel® Math Kernel Library routines can be stored in either one- or two-dimensional arrays, using the following storage schemes:

- conventional full storage (in a two-dimensional array)
- packed storage for Hermitian, symmetric, or triangular matrices (in a one-dimensional array)
- band storage for band matrices (in a two-dimensional array).

Full storage is the following obvious scheme: a matrix A is stored in a two-dimensional array a , with the matrix element a_{ij} stored in the array element $a(i, j)$.

If a matrix is triangular (upper or lower, as specified by the argument *uplo*), only the elements of the relevant triangle are stored; the remaining elements of the array need not be set.

Routines that handle symmetric or Hermitian matrices allow for either the upper or lower triangle of the matrix to be stored in the corresponding elements of the array:

if $uplo = 'U'$, a_{ij} is stored in $a(i, j)$ for $i \leq j$, other elements of a need not be set.

if $uplo = 'L'$, a_{ij} is stored in $a(i, j)$ for $j \leq i$, other elements of a need not be set.

Packed storage allows you to store symmetric, Hermitian, or triangular matrices more compactly: the relevant triangle (again, as specified by the argument *uplo*) is packed by columns in a one-dimensional array ap :

if $uplo = 'U'$, a_{ij} is stored in $ap(i+j(j-1)/2)$ for $i \leq j$

if `uplo = 'L'`, a_{ij} is stored in $ap(i + (2*n-j)*(j-1)/2)$ for $j \leq i$.

In descriptions of LAPACK routines, arrays with packed matrices have names ending in *p*.

Band storage is as follows: an m -by- n band matrix with kl non-zero sub-diagonals and ku non-zero super-diagonals is stored compactly in a two-dimensional array *ab* with $kl+ku+1$ rows and n columns. Columns of the matrix are stored in the corresponding columns of the array, and diagonals of the matrix are stored in rows of the array. Thus,

a_{ij} is stored in $ab(kl+1+i-j, j)$ for $\max(1, j-ku) \leq i \leq \min(n, j+kl)$.

Use the band storage scheme only when kl and ku are much less than the matrix size n . Although the routines work correctly for all values of kl and ku , using the band storage is inefficient if your matrices are not really banded.

The band storage scheme is illustrated by the following example, when

$m = n = 6$, $kl = 2$, $ku = 1$

Array elements marked * are not used by the routines:

Banded matrix A						Band storage of A					
a_{11}	a_{12}	0	0	0	0						
a_{21}	a_{22}	a_{23}	0	0	0	*	a_{12}	a_{23}	a_{34}	a_{45}	a_{56}
a_{31}	a_{42}	a_{43}	a_{34}	0	0	a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}
0	a_{42}	a_{43}	a_{44}	a_{45}	0	a_{21}	a_{32}	a_{43}	a_{54}	a_{65}	*
0	0	a_{53}	a_{54}	a_{55}	a_{56}	a_{31}	a_{42}	a_{53}	a_{64}	*	*
0	0	0	a_{64}	a_{65}	a_{66}						

When a general band matrix is supplied for *LU factorization*, space must be allowed to store kl additional super-diagonals generated by fill-in as a result of row interchanges. This means that the matrix is stored according to the above scheme, but with $kl + ku$ super-diagonals. Thus,

a_{ij} is stored in $ab(kl+ku+1+i-j, j)$ for $\max(1, j-ku) \leq i \leq \min(n, j+kl)$.

The band storage scheme for LU factorization is illustrated by the following example, when $m = n = 6$, $kl = 2$, $ku = 1$:

Banded matrix A						Band storage of A					
a_{11}	a_{12}	0	0	0	0	*	*	*	+	+	+
a_{21}	a_{22}	a_{23}	0	0	0	*	*	+	+	+	+
a_{31}	a_{42}	a_{43}	a_{34}	0	0	*	a_{12}	a_{23}	a_{34}	a_{45}	a_{56}
0	a_{42}	a_{43}	a_{44}	a_{45}	0	a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}
0	0	a_{53}	a_{54}	a_{55}	a_{56}	a_{21}	a_{32}	a_{43}	a_{54}	a_{65}	*
0	0	0	a_{64}	a_{65}	a_{66}	a_{31}	a_{42}	a_{53}	a_{64}	*	*

Array elements marked * are not used by the routines; elements marked + need not be set on entry, but are required by the LU factorization routines to store the results. The input array will be overwritten on exit by the details of the LU factorization as follows:

*	*	*	u_{14}	u_{25}	u_{36}
*	*	u_{13}	u_{24}	u_{35}	u_{46}
*	u_{12}	u_{23}	u_{34}	u_{45}	u_{56}
u_{11}	u_{22}	u_{33}	u_{44}	u_{55}	u_{66}
m_{21}	m_{32}	m_{43}	m_{54}	m_{65}	*
m_{31}	m_{42}	m_{53}	m_{64}	*	*

where u_{ij} are the elements of the upper triangular matrix U, and m_{ij} are the multipliers used during factorization.

Triangular band matrices are stored in the same format, with either $kl = 0$ if upper triangular, or $ku = 0$ if lower triangular. For symmetric or Hermitian band matrices with k sub-diagonals or super-diagonals, you need to store only the upper or lower triangle, as specified by the argument *uplo*:

if *uplo* = 'U', a_{ij} is stored in $ab(k+1+i-j, j)$ for $\max(1, j-k) \leq i \leq j$

if $uplo = 'L'$, a_{ij} is stored in $ab(1+i-j, j)$ for $j \leq i \leq \min(n, j+k)$.

In descriptions of LAPACK routines, arrays that hold matrices in band storage have names ending in b .

In Fortran, column-major ordering of storage is assumed. This means that elements of the same column occupy successive storage locations.

Three quantities are usually associated with a two-dimensional array argument: its leading dimension, which specifies the number of storage locations between elements in the same row, its number of rows, and its number of columns. For a matrix in full storage, the leading dimension of the array must be at least as large as the number of rows in the matrix.

A character transposition parameter is often passed to indicate whether the matrix argument is to be used in normal or transposed form or, for a complex matrix, if the conjugate transpose of the matrix is to be used.

The values of the transposition parameter for these three cases are the following:

'N' or 'n'	normal (no conjugation, no transposition)
'T' or 't'	transpose
'C' or 'c'	conjugate transpose.

Example B-3 Two-Dimensional Complex Array

Suppose $A(1:5, 1:4)$ is the complex two-dimensional array presented by matrix

$$\begin{bmatrix} (1.1, 0.11) & (1.2, 0.12) & (1.3, 0.13) & (1.4, 0.14) \\ (2.1, 0.21) & (2.2, 0.22) & (2.3, 0.23) & (1.4, 0.24) \\ (3.1, 0.31) & (3.2, 0.32) & (3.3, 0.33) & (1.4, 0.34) \\ (4.1, 0.41) & (4.2, 0.42) & (4.3, 0.43) & (1.4, 0.44) \\ (5.1, 0.51) & (5.2, 0.52) & (5.3, 0.53) & (1.4, 0.54) \end{bmatrix}$$

Let $transa$ be the transposition parameter, m be the number of rows, n be the number of columns, and lda be the leading dimension. Then if

$transa = 'N'$, $m = 4$, $n = 2$, and $lda = 5$, the matrix argument would be

$$\begin{bmatrix} (1.1, 0.11) & (1.2, 0.12) \\ (2.1, 0.21) & (2.2, 0.22) \\ (3.1, 0.31) & (3.2, 0.32) \\ (4.1, 0.41) & (4.2, 0.42) \end{bmatrix}$$

If `transa = 'T'`, `m = 4`, `n = 2`, and `lda = 5`, the matrix argument would be

$$\begin{bmatrix} (1.1, 0.11) & (2.1, 0.21) & (3.1, 0.31) & (4.1, 0.41) \\ (1.2, 0.12) & (2.2, 0.22) & (3.2, 0.32) & (4.2, 0.42) \end{bmatrix}$$

If `transa = 'C'`, `m = 4`, `n = 2`, and `lda = 5`, the matrix argument would be

$$\begin{bmatrix} (1.1, -0.11) & (2.1, -0.21) & (3.1, -0.31) & (4.1, -0.41) \\ (1.2, -0.12) & (2.2, -0.22) & (3.2, -0.32) & (4.2, -0.42) \end{bmatrix}$$

Note that care should be taken when using a leading dimension value which is different from the number of rows specified in the declaration of the two-dimensional array. For example, suppose the array `A` above is declared as `COMPLEX A (5,4)`.

Then if `transa = 'N'`, `m = 3`, `n = 4`, and `lda = 4`, the matrix argument will be

$$\begin{bmatrix} (1.1, 0.11) & (5.1, 0.51) & (4.2, 0.42) & (3.3, 0.33) \\ (2.1, 0.21) & (1.2, 0.12) & (5.2, 0.52) & (4.3, 0.43) \\ (3.1, 0.31) & (2.2, 0.22) & (1.3, 0.13) & (5.3, 0.53) \end{bmatrix}$$

Code Examples



This appendix presents code examples of using some Intel MKL routines and functions. You can find here example code written in both Fortran and C.

Currently, the appendix includes the following sections:

- [BLAS Code Examples](#)
- [PARDISO Code Examples](#)
- [Direct Sparse Solver Code Examples](#)
- [Iterative Sparse Solver Code Examples](#)
- [Preconditioner Code Examples](#)
- [Fourier Transform Functions Code Examples](#)
- [Interval Linear Solvers Code Examples](#)
- [PDE Support Code Examples](#).

Please refer to respective chapters in the manual for detailed descriptions of function parameters and operation.

BLAS Code Examples

Example C-1. Using BLAS Level 1 Function

The following example illustrates a call to the BLAS Level 1 function `sdot`. This function performs a vector-vector operation of computing a scalar product of two single-precision real vectors x and y .

Parameters

n	Specifies the number of elements in vectors x and y .
$incx$	Specifies the increment for the elements of x .

Parameters

incy Specifies the increment for the elements of *y*.

```
program dot_main
real x(10), y(10), sdot, res
integer n, incx, incy, i
external sdot
n = 5
incx = 2
incy = 1
do i = 1, 10
    x(i) = 2.0e0
    y(i) = 1.0e0
end do
res = sdot (n, x, incx, y, incy)
print*, `SDOT = `, res
end
```

As a result of this program execution, the following line is printed:

SDOT = 10.000

Example C-2. Using BLAS Level 1 Routine

The following example illustrates a call to the BLAS Level 1 routine *scopy*. This routine performs a vector-vector operation of copying a single-precision real vector *x* to a vector *y*.

Parameters

n Specifies the number of elements in vectors *x* and *y*.

incx Specifies the increment for the elements of *x*.

Parameters

incy Specifies the increment for the elements of *y*.

```
program copy_main
real x(10), y(10)
integer n, incx, incy, i
n = 3
incx = 3
incy = 1
do i = 1, 10
    x(i) = i
end do
call scopy (n, x, incx, y, incy)
print*, `Y = `, (y(i), i = 1, n)
end
```

As a result of this program execution, the following line is printed:

Y = 1.00000 4.00000 7.00000

Example C-3. Using BLAS Level 2 Routine

The following example illustrates a call to the BLAS Level 2 routine `sger`. This routine performs a matrix-vector operation

$a := \alpha x y' + a.$

Parameters

alpha Specifies a scalar *alpha*.
x *m*-element vector.
y *n*-element vector.

Parameters

a *m*-by-*n* matrix.

```
program ger_main
real a(5,3), x(10), y(10), alpha
integer m, n, incx,
incy, i, j, lda
m = 2
n = 3
lda = 5
incx = 2
incy = 1
alpha = 0.5
do i = 1, 10
    x(i) = 1.0
    y(i) = 1.0
end do
do i = 1, m
    do j = 1, n
        a(i,j) = j
    end do
end do
call sger (m, n, alpha, x, incx, y, incy, a, lda)
print*, `Matrix A: `
do i = 1, m
    print*, (a(i,j), j = 1, n)
end do
end
```

As a result of this program execution, matrix *a* is printed as follows:

Matrix A:

```
1.50000 2.50000 3.50000
```

```
1.50000 2.50000 3.50000
```

Example C-4. Using BLAS Level 3 Routine

The following example illustrates a call to the BLAS Level 3 routine `ssymm`. This routine performs a matrix-matrix operation

$c := \alpha a a' + \beta c.$

Parameters

alpha Specifies a scalar *alpha*.

beta Specifies a scalar *beta*.

a Symmetric matrix

b *m*-by-*n* matrix

Parameters

c *m*-by-*n* matrix

```
program symm_main
real a(3,3), b(3,2), c(3,3), alpha, beta
integer m, n, lda, ldb, ldc, i, j
character uplo, side
uplo = 'u'
side = 'l'
m = 3
n = 2
lda = 3
ldb = 3
ldc = 3
alpha = 0.5
beta = 2.0
do i = 1, m
    do j = 1, m
        a(i,j) = 1.0
    end do
end do
do i = 1, m
    do j = 1, n
        c(i,j) = 1.0
        b(i,j) = 2.0
    end do
end do
call ssymm (side, uplo, m, n, alpha,
a, lda, b, ldb, beta, c, ldc)
```

```
print*, `Matrix C: `  
do i = 1, m  
    print*, (c(i,j), j = 1, n)  
end do  
end
```

As a result of this program execution, matrix *c* is printed as follows:

Matrix C:

```
5.00000 5.00000  
5.00000 5.00000  
5.00000 5.00000
```

The following example illustrates a call from a C program to the complex BLAS Level 1 function `zdotc()`. This function computes the dot product of two double-precision complex vectors.

Example C-5. Calling a Complex BLAS Level 1 Function from C

In this example, the complex dot product is returned in the structure *c*.

```
#define N 5

void main()
{

    int n, inca = 1, incb = 1, i;

    typedef struct{ double re; double im; } complex16;
    complex16 a[N], b[N], c;

    void zdotc();

    n = N;
    for( i = 0; i < n; i++ ){
        a[i].re = (double)i; a[i].im = (double)i * 2.0;
        b[i].re = (double)(n - i); b[i].im = (double)i * 2.0;
    }

    zdotc( &c, &n, a, &inca, b, &incb );

    printf( "The complex dot product is: ( %6.2f, %6.2f )\n", c.re, c.im );
}
```



NOTE. Instead of calling BLAS directly from C programs, you might wish to use the CBLAS interface; this is the supported way of calling BLAS from C. For more information about CBLAS, see [Appendix D](#), which presents CBLAS, the C interface to the Basic Linear Algebra Subprograms (BLAS) implemented in Intel® MKL.

PARDISO Code Examples

This section presents code examples of using the PARDISO direct solver for computing solutions of linear systems with sparse matrices. For description of this solver, refer to Chapter 8 of the manual.

Examples for Sparse Symmetric Linear Systems

In this section two examples (Fortran, C) are provided to solve symmetric linear systems with PARDISO. To solve the systems of equations $Ax = b$, where

$$A = \begin{bmatrix} 7.0 & 0.0 & 1.0 & 0.0 & 0.0 & 2.0 & 7.0 & 0.0 \\ 0.0 & -4.0 & 8.0 & 0.0 & 2.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 8.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 5.0 \\ 0.0 & 0.0 & 0.0 & 7.0 & 0.0 & 0.0 & 9.0 & 0.0 \\ 0.0 & 2.0 & 0.0 & 0.0 & 5.0 & 1.0 & 5.0 & 0.0 \\ 2.0 & 0.0 & 0.0 & 0.0 & 1.0 & -1.0 & .0 & 5.0 \\ 7.0 & 0.0 & 0.0 & 9.0 & 5.0 & 0.0 & 11.0 & 0.0 \\ 0.0 & 0.0 & 5.0 & 0.0 & 0.0 & 5.0 & 0.0 & 5.0 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{bmatrix}$$

Example Results for Symmetric Systems

Upon successful execution of the solver, the result of the solution x is as follows

Reordering completed ...

Number of nonzeros in factors = 30

Number of factorization MFLOPS = 0

Factorization completed ...

Solve completed ...

The solution of the system is

$x(1) = -0.0418602013$

$x(2) = -0.00341312416$

$x(3) = 0.117250377$

$x(4) = -0.11263958$

$x(5) = 0.0241722445$

$x(6) = -0.10763334$

$x(7) = 0.198719673$

$x(8) = 0.190382964$

Example C-6. pardiso_sym.f for Symmetric Linear Systems

```

C-----
C Example program to show the use of the "PARDISO" routine
C for symmetric linear systems
C-----
C This program can be downloaded from the following site:
C http://www.pardiso-project.org
C (C) Olaf Schenk, Department of Computer Science,
C University of Basel, Switzerland.
C Email: olaf.schenk@unibas.ch
C
C-----

      PROGRAM pardiso_sym
      IMPLICIT NONE

C.. Internal solver memory pointer for 64-bit architectures
C.. INTEGER*8 pt(64)
C.. Internal solver memory pointer for 32-bit architectures
C.. INTEGER*4 pt(64)
C.. This is OK in both cases
      INTEGER*8 pt(64)

C.. All other variables
      INTEGER maxfct, mnum, mtype, phase, n, nrhs, error, msglvl
      INTEGER iparm(64)
      INTEGER ia(9)
      INTEGER ja(18)
      REAL*8 a(18)
      REAL*8 b(8)
      REAL*8 x(8)

```

```

    INTEGER i, idum

    REAL*8 waltime1, waltime2, ddum

C.. Fill all arrays containing matrix data.

    DATA n /8/, nrhs /1/, maxfct /1/, mnum /1/

    DATA ia /1,5,8,10,12,15,17,18,19/

    DATA ja
1 /1, 3, 6,7,
2 2,3, 5,
3 3, 8,
4 4, 7,
5 5,6,7,
6 6, 8,
7 7,
8 8/

    DATA a
1 /7.d0, 1.d0, 2.d0,7.d0,
2 -4.d0,8.d0, 2.d0,
3 1.d0, 5.d0,
4 7.d0, 9.d0,
5 5.d0,1.d0,5.d0,
6 -1.d0, 5.d0,
7 11.d0,
8 5.d0/

    integer omp_get_max_threads

    external omp_get_max_threads

C..

C.. Set up PARDISO control parameter

C..

```

```

do i = 1, 64
    iparm(i) = 0
end do
iparm(1) = 1 ! no solver default
iparm(2) = 2 ! fill-in reordering from METIS
iparm(3) = mkl_get_max_threads() !numbers of processors, value of MKL_NUM_THREADS
iparm(4) = 0 ! no iterative-direct algorithm
iparm(5) = 0 ! no user fill-in reducing permutation
iparm(6) = 0 ! =0 solution on the first n components of x
iparm(7) = 16 ! default logical fortran unit number for output
iparm(8) = 9 ! numbers of iterative refinement steps
iparm(9) = 0 ! not in use
iparm(10) = 13 ! perturb the pivot elements with 1E-13
iparm(11) = 1 ! use nonsymmetric permutation and scaling MPS
iparm(12) = 0 ! not in use
iparm(13) = 0 ! not in use
iparm(14) = 0 ! Output: number of perturbed pivots
iparm(15) = 0 ! not in use
iparm(16) = 0 ! not in use
iparm(17) = 0 !
not in use
iparm(18) = -1 ! Output: number of nonzeros in the factor LU
iparm(19) = -1 ! Output: Mflops for LU factorization
iparm(20) = 0 ! Output: Numbers of CG Iterations
error = 0 ! initialize error flag
msglvl = 0 ! don't print statistical information
mtype = -2 ! unsymmetric matrix symmetric, indefinite, no pivoting
C.. Initialize the internal solver memory pointer. This is only
C necessary for the FIRST call of the PARDISO solver.

```

```

do i = 1, 64
    pt(i) = 0
end do

```

C.. Reordering and Symbolic Factorization, This step also allocates
C all memory that is necessary for the factorization

```

phase = 11 ! only reordering and symbolic factorization
CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja,
1 idum, nrhs, iparm, msglvl, ddum, ddum, error)
WRITE(*,*) 'Reordering completed ... '
IF (error .NE. 0) THEN
    WRITE(*,*) 'The following ERROR was detected: ', error
    STOP
END IF
WRITE(*,*) 'Number of nonzeros in factors = ',iparm(18)
WRITE(*,*) 'Number of factorization MFLOPS = ',iparm(19)

```

C.. Factorization.

```

phase = 22 ! only factorization
CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja,
1 idum, nrhs, iparm, msglvl, ddum, ddum, error)
WRITE(*,*) 'Factorization completed ... '
IF (error .NE. 0) THEN
    WRITE(*,*) 'The following ERROR was detected: ', error
    STOP
ENDIF

```

C.. Back substitution and iterative refinement

```

iparm(8) = 2 ! max numbers of iterative refinement steps
phase = 33 ! only factorization
do i = 1, n

```

```
        b(i) = 1.d0
    end do

    CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja,
1 idum, nrhs, iparm, msglvl, b, x, error)
    WRITE(*,*) 'Solve completed ... '
    WRITE(*,*) 'The solution of the system is '
    DO i = 1, n
        WRITE(*,*) ' x(',i,') = ', x(i)
    END DO

C.. Termination and release of memory
    phase = -1 ! release internal memory
    CALL pardiso (pt, maxfct, mnum, mtype, phase, n, ddum, idum, idum,
1 idum, nrhs, iparm, msglvl, ddum, ddum, error)
    END
```

Example C-7. pardiso_sym.c for Symmetric Linear Systems

```

/* -----*/
/* Example program to show the use of the "PARDISO" routine */
/* on symmetric linear systems*/
/* -----*/
/* This program can be downloaded from the following site: */
/* http://www.pardiso-project.org*/
/* */
/* (C) Olaf Schenk, Department of Computer Science, */
/* University of Basel, Switzerland.*/
/* Email: olaf.schenk@unibas.ch*/
/* -----*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
extern int omp_get_max_threads();
/* PARDISO prototype. */
extern int PARDISO
    (void *, int *, int *, int *, int *, int *,
     double *, int *, int *, int*, int *, int *,
     int *, double *, double *, int*);

int main( void ) {
    /* Matrix data. */
    int n = 8;
    int ia[ 9] = { 1, 5, 8, 10, 12, 15, 17, 18, 19 };
    int ja[18] = { 1, 3, 6, 7,
                  2, 3, 5,

```



```
    3, 8,  
    4, 7,  
    5, 6, 7,  
    6, 8,  
    7,  
    8 };  
  
double a[18] = { 7.0, 1.0, 2.0, 7.0,  
    -4.0, 8.0, 2.0,  
    1.0, 5.0,  
    7.0, 9.0,  
    5.0, 1.0, 5.0,  
    -1.0, 5.0,  
    11.0,  
    5.0 };  
  
int mtype = -2; /* Real symmetric matrix */  
/* RHS and solution vectors.*/  
double b[8], x[8];  
  
int nrhs = 1; /* Number of right hand sides. */  
/* Internal solver memory pointer pt, */  
/* 32-bit: int pt[64]; 64-bit: long int pt[64] */  
/* or void *pt[64] should be OK on both architectures */  
void *pt[64];  
/* Pardiso control parameters.*/  
int iparm[64];  
int maxfct, mnum, phase, error, msglvl;  
/* Auxiliary variables. */  
int i;  
double ddum; /* Double dummy*/
```

```

    int idum; /* Integer dummy.*/

/* -----*/
/* .. Setup Pardiso control parameters.*/
/* -----*/

    for (i = 0; i < 64; i++) {
        iparm[i] = 0;
    }

    iparm[0] = 1; /* No solver default*/
    iparm[1] = 2; /* Fill-in reordering from METIS */

    /* Numbers of processors, value of MKL_N
UM_THREADS */
    iparm[2] = mkl_get_max_threads();
    iparm[3] = 0; /* No iterative-direct algorithm */
    iparm[4] = 0; /* No user fill-in reducing permutation */
    iparm[5] = 0; /* Write solution into x */
    iparm[6] = 16; /* Default logical fortran unit number for output */
    iparm[7] = 2; /* Max numbers of iterative refinement steps */
    iparm[8] = 0; /* Not in use*/
    iparm[9] = 13; /* Perturb the pivot elements with 1E-13 */
    iparm[10] = 1; /* Use nonsymmetric permutation and scaling MPS */
    iparm[11] = 0; /* Not in use*/
    iparm[12] = 0; /* Not in use*/
    iparm[13] = 0; /* Output: Number of perturbed pivots */
    iparm[14] = 0; /* Not in use*/
    iparm[15] = 0; /* Not in use*/
    iparm[16] = 0; /* Not in use*/
    iparm[17] = -1; /* Output: Number of nonzeros in the factor LU */
    iparm[18] = -1; /* Output: Mflops for LU factorization */
    iparm[19] = 0; /* Output: Numbers of CG Iterations */

```

```

    maxfct = 1; /* Maximum number of numerical factorizations. */
    mnum = 1; /* Which factorization to use. */
    msglvl = 0; /* Don't print statistical information in file */
    error = 0; /* Initialize error flag */
/* -----*/
/* .. Initialize the internal solver memory pointer. This is only */
/* necessary for the FIRST call of the PARDISO solver. */
/* -----*/
    for (i = 0; i < 64; i++) {
        pt[i] = 0;
    }
/* -----*/
/* .. Reordering and Symbolic Factorization. This step also allocates */
/* all memory that is necessary for the factorization. */
/* -----*/
    phase = 11;
    PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
             &n, a, ia, ja, &idum, &nrhs,
             iparm, &msglvl, &ddum, &ddum, &error);
    if (error != 0) {
        printf("\nERROR during symbolic factorization: %d", error);
        exit(1);
    }
    printf("\nReordering completed ... ");
    printf("\nNumber of nonzeros in factors = %d", iparm[17]);
    printf("\nNumber of factorization MFLOPS = %d", iparm[18]);
/* -----*/
/* .. Numerical factorization.*/

```

```

/* -----*/
phase = 22;
PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
         &n, a, ia, ja, &idum, &nrhs,
         iparm, &msglvl, &ddum, &ddum, &error);
if (error != 0) {
    printf("\nERROR during numerical factorization: %d", error);
    exit(2);
}
printf("\nFactorization completed ... ");
/* -----*/
/* .. Back substitution and iterative refinement. */
/* -----*/
phase = 33;
iparm[7] = 2; /* Max numbers of iterative refinement steps. */
/* Set right hand side to one.*/
for (i = 0; i < n; i++) {
    b[i] = 1;
}
PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
         &n, a, ia, ja, &idum, &nrhs,
         iparm, &msglvl, b, x, &error);
if (error != 0) {
    printf("\nERROR during solution: %d", error);
    exit(3);
}
printf("\nSolve completed ... ");
printf("\nThe solution of the system is: ");

```

```

    for (i = 0; i < n; i++) {
        printf("\n x [%d] = % f", i, x[i] );
    }
    printf ("\n");
/* -----*/
/* .. Termination and release of memory. */
/* -----*/

    phase = -1; /* Release internal memory. */
    PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
             &n, &ddum, ia, ja, &idum, &nrhs,
             iparm, &msglvl, &ddum, &ddum, &error);

    return 0;
}

```

Examples for Sparse Unsymmetric Linear Systems

In this section two examples (Fortran, C) are provided to solve unsymmetric linear systems with PARDISO. To solve the systems of equations $Ax = b$, where

$$A = \begin{bmatrix} 1.0 & -1.0 & 0.0 & -3.0 & 0.0 \\ -2.0 & 5.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 4.0 & 6.0 & 4.0 \\ -4.0 & 0.0 & 2.0 & 7.0 & 0.0 \\ 0.0 & 8.0 & 0.0 & 0.0 & -5.0 \end{bmatrix} \text{ and } B = \begin{bmatrix} 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{bmatrix}$$

Example Results for Unsymmetric Systems

Upon successful execution of the solver, the result of the solution x is as follows

Reordering completed ...

Number of nonzeros in factors = 21

Number of factorization MFLOPS = 0

Factorization completed ...

Solve completed ...

The solution of the system is

$x(1) = -0.522321429$

$x(2) = -0.00892857143$

$x(3) = 1.22098214$

$x(4) = -0.504464286$

$x(5) = -0.214285714$

Example C-8. pardiso_unsym.f for Unsymmetric Linear Systems

```
*****
*
*                               INTEL CONFIDENTIAL
*
*   Copyright(C) 2004-2008 Intel Corporation. All Rights Reserved.
*
*   The source code contained or described herein and all documents related to
*   the source code ("Material") are owned by Intel Corporation or its suppliers
*   or licensors. Title to the Material remains with Intel Corporation or its
*   suppliers and licensors. The Material contains trade secrets and proprietary
*   and confidential information of Intel or its suppliers and licensors. The
*   Material is protected by worldwide copyright and trade secret laws and
*   treaty provisions. No part of the Material may be used, copied, reproduced,
*   modified, published, uploaded, posted, transmitted, distributed or disclosed
*   in any way without Intel's prior express written permission.
*
*   No license under any patent, copyright, trade secret or other intellectual
*   property right is granted to or conferred upon you by disclosure or delivery
*   of the Materials, either expressly, by implication, inducement, estoppel or
*   otherwise. Any license under such intellectual property rights must be
*   express and approved by Intel in writing.
*
*****
*   Content : MKL PARDISO Fortran 77 example
*
*****
C-----
C Example program to show the use of the "PARDISO" routine
C for symmetric linear systems
C-----
C This program can be downloaded from the following site:
C http://www.pardiso-project.org
```

C

C (C) Olaf Schenk, Department of Computer Science,

C University of Basel, Switzerland.

C Email: olaf.schenk@unibas.ch

C

C-----

PROGRAM pardiso_unsym

IMPLICIT NONE

C.. Internal solver memory pointer for 64-bit architectures

C.. INTEGER*8 pt(64)

C.. Internal solver memory pointer for 32-bit architectures

C.. INTEGER*4 pt(64)

C.. This is OK in both cases

INTEGER*8 pt(64)

C.. All other variables

INTEGER maxfct, mnum, mtype, phase, n, nrhs, error, msglvl

INTEGER iparm(64)

INTEGER ia(6)

INTEGER ja(13)

REAL*8 a(13)

REAL*8 b(5)

REAL*8 x(5)

INTEGER i, idum

REAL*8 wertime1, wertime2, ddum

C.. Fill all arrays containing matrix data.

DATA n /5/, nrhs /1/, maxfct /1/, mnum /1/

DATA ia /1,4,6,9,12,14/

DATA ja

```

1 / 1,      2,          4,
2  1,      2,
3          3,      4,      5,
4  1,          3,      4,
5          2,          5/

DATA a
1 /1.d0, -1.d0,      -3.d0,
2 -2.d0,  5.d0,
3          4.d0,  6.d0, 4.d0,
4 -4.d0,      2.d0,  7.d0,
5          8.d0,          -5.d0/

integer omp_get_max_threads
external omp_get_max_threads

C..
C.. Set up PARDISO control parameter
C..

do i = 1, 64
    iparm(i) = 0
end do

iparm(1) = 1 ! no solver default
iparm(2) = 2 ! fill-in reordering from METIS
iparm(3) = mkl_get_max_threads() ! numbers of processors, value of MKL_NUM_THREADS
iparm(4) = 0 ! no iterative-direct algorithm
iparm(5) = 0 ! no user fill-in reducing permutation
iparm(6) = 0 ! =0 solution on the first n compoments of x
iparm(7) = 0 ! not in use
iparm(8) = 9 ! numbers of iterative refinement steps
iparm(9) = 0 ! not in use

```

```

iparm(10) = 13 ! perturb the pivot elements with 1E-13
iparm(11) = 1 ! use nonsymmetric permutation and scaling MPS
iparm(12) = 0 ! not in use
iparm(13) = 0 ! not in use
iparm(14) = 0 ! Output: number of perturbed pivots
iparm(15) = 0 ! not in use
iparm(16) = 0 ! not in use
iparm(17) = 0 ! not in use
iparm(18) = -1 ! Output: number of nonzeros in the factor LU
iparm(19) = -1 ! Output: Mflops for LU factorization
iparm(20) = 0 ! Output: Numbers of CG Iterations
error = 0 ! initialize error flag
msglvl = 1 ! print statistical information
mtype = 11 ! real unsymmetric

```

C.. Initialize the internal solver memory pointer. This is only
C necessary for the FIRST call of the PARDISO solver.

```

do i = 1, 64
    pt(i) = 0
end do

```

C.. Reordering and Symbolic Factorization, This step also allocates
C all memory that is necessary for the factorization

```

phase = 11 ! only reordering and symbolic factorization
CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja,
1 idum, nrhs, iparm, msglvl, ddum, ddum, error)
WRITE(*,*) 'Reordering completed ... '
IF (error .NE. 0) THEN
    WRITE(*,*) 'The following ERROR was detected: ', error
    STOP

```

```

        END IF

        WRITE(*,*) 'Number of nonzeros in factors = ',iparm(18)

        WRITE(*,*) 'Number of factorization MFLOPS = ',iparm(19)
C.. Factorization.
        phase = 22 ! only factorization
        CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja,
1 idum, nrhs, iparm, msglvl, ddum, ddum, error)
        WRITE(*,*) 'Factorization completed ... '
        IF (error .NE. 0) THEN
            WRITE(*,*) 'The following ERROR was detected: ', error
            STOP
        ENDIF
C.. Back substitution and iterative refinement
        iparm(8) = 2 ! max numbers of iterative refinement steps
        phase = 33 ! only factorization
        do i = 1, n
            b(i) = 1.d0
        end do
        CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja,
1 idum, nrhs, iparm, msglvl, b, x, error)
        WRITE(*,*) 'Solve completed ... '
        WRITE(*,*) 'The solution of the system is '
        DO i = 1, n
            WRITE(*,*) ' x('i,i') = ', x(i)
        END DO
C.. Termination and release of memory
        phase = -1 ! release internal memory
        CALL pardiso (pt, maxfct, mnum, mtype, phase, n, ddum, idum, idum,

```

```
1 idum, nrhs, iparm, msglvl, ddum, ddum, error)
```

```
END
```

Example C-9. pardiso_unsym.c for Unsymmetric Linear Systems

```

/*
*****
*
*               INTEL CONFIDENTIAL
*
* Copyright(C) 2004-2008 Intel Corporation. All Rights Reserved.
*
* The source code contained or described herein and all documents related to
* the source code ("Material") are owned by Intel Corporation or its suppliers
* or licensors. Title to the Material remains with Intel Corporation or its
* suppliers and licensors. The Material contains trade secrets and proprietary
* and confidential information of Intel or its suppliers and licensors. The
* Material is protected by worldwide copyright and trade secret laws and
* treaty provisions. No part of the Material may be used, copied, reproduced,
* modified, published, uploaded, posted, transmitted, distributed or disclosed
* in any way without Intel's prior express written permission.
*
* No license under any patent, copyright, trade secret or other intellectual
* property right is granted to or conferred upon you by disclosure or delivery
* of the Materials, either expressly, by implication, inducement, estoppel or
* otherwise. Any license under such intellectual property rights must be
* express and approved by Intel in writing.
*
*****
*   Content : MKL PARDISO C example
*
*****
*/
/* -----*/
/* Example program to show the use of the "PARDISO" routine */
/* on symmetric linear systems*/

```

```
/* -----*/
/* This program can be downloaded from the following site: */
/* http://www.pardiso-project.org*/
/* */
/* (C) Olaf Schenk, Department of Computer Science, */
/* University of Basel, Switzerland.*/
/* Email: olaf.schenk@unibas.ch*/
/* -----*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
extern int omp_get_max_threads();
/* PARDISO prototype. */
#if defined(_WIN32) || defined(_WIN64)
#define pardiso_ PARDISO
#else
#define PARDISO pardiso_
#endif
extern int PARDISO
        (void *, int *, int *, int *, int *, int *,
         double *, int *, int*, int *, int *, int *,
         int *, double *, double*, int *);

int main( void ) {
        /* Matrix data. */
        int n = 5;
        int ia[ 6] = { 1, 4, 6, 9, 12, 14 };
        int ja[13] = { 1, 2, 4,
```

```

        1, 2,
        3, 4, 5,
        1, 3, 4,
        2, 5 };

double a[18] = { 1.0, -1.0, -3.0,
                -2.0, 5.0,
                4.0, 6.0, 4.0,
                -4.0, 2.0, 7.0,
                8.0, -5.0 };

int mtype = 11; /* Real unsymmetric matrix */
/* RHS and solution vectors.*/
double b[5], x[5];
int nrhs = 1; /* Number of right hand sides. */
/* Internal solver memory pointer pt, */
/* 32-bit: int pt[64]; 64-bit: long int pt[64] */
/* or void *pt[64] should be OK on both architectures */
void *pt[64];
/* Pardiso control parameters.*/
int iparm[64];
int maxfct, mnum, phase, error, msglvl;
/* Auxiliary variables.*/
int i;
double ddum; /* Double dummy */
int idum; /* Integer dummy. */

/* -----*/
/* .. Setup Pardiso control parameters.*/
/* -----*/

for (i = 0; i < 64; i++) {

```

```

        iparm[i] = 0;
    }
    iparm[0] = 1; /* No solver default */
    iparm[1] = 2; /* Fill-in reordering from METIS
*/
    /* Numbers of processors, value of MKL_NUM_THREADS */
    iparm[2] = mkl_get_max_threads();
    iparm[3] = 0; /* No iterative-direct algorithm */
    iparm[4] = 0; /* No user fill-in reducing permutation */
    iparm[5] = 0; /* Write solution into x */
    iparm[6] = 0; /* Not in use */
    iparm[7] = 2; /* Max numbers of iterative refinement steps */
    iparm[8] = 0; /* Not in use */
    iparm[9] = 13; /* Perturb the pivot elements with 1E-13 */
    iparm[10] = 1; /* Use nonsymmetric permutation and scaling MPS */
    iparm[11] = 0; /* Not in use */
    iparm[12] = 0; /* Not in use */
    iparm[13] = 0; /* Output: Number of perturbed pivots */
    iparm[14] = 0; /* Not in use */
    iparm[15] = 0; /* Not in use */
    iparm[16] = 0; /* Not in use */
    iparm[17] = -1; /* Output: Number of nonzeros in the factor LU */
    iparm[18] = -1; /* Output: Mflops for LU factorization */
    iparm[19] = 0; /* Output: Numbers of CG Iterations */
    maxfct = 1; /* Maximum number of numerical factorizations. */
    mnum = 1; /* Which factorization to use. */
    msglvl = 1; /* Print statistical information in file */
    error = 0; /* Initialize error flag */
/* -----*/

```

```

/* .. Initialize the internal solver memory pointer. This is only */
/* necessary for the FIRST call of the PARDISO solver. */
/* -----*/
    for (i = 0; i < 64; i++) {
        pt[i] = 0;
    }
/* -----*/
/* .. Reordering and Symbolic Factorization. This step also allocates */
/* all memory that is necessary for the factorization. */
/* -----*/
    phase = 11;
    PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
             &n, a, ia, ja, &idum, &nrhs,
             iparm, &msglvl, &ddum, &ddum, &error);
    if (error != 0) {
        printf("\nERROR during symbolic factorization: %d", error);
        exit(1);
    }
    printf("\nReordering completed ... ");
    printf("\nNumber of nonzeros in factors = %d", iparm[17]);
    printf("\nNumber of factorization MFLOPS = %d", iparm[18]);
/* -----*/
/* .. Numerical factorization.*/
/* -----*/
    phase = 22;
    PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
             &n, a, ia, ja, &idum, &nrhs,
             iparm, &msglvl, &ddum, &ddum, &error);

```

```

        if (error != 0) {
            printf("\nERROR during numerical factorization: %d", error);
            exit(2);
        }

        printf("\nFactorization completed ... ");

/* -----*/
/* .. Back substitution and iterative refinement. */
/* -----*/

    phase = 33;
    iparm[7] = 2; /* Max numbers of iterative refinement steps. */
    /* Set right hand side to one. */
    for (i = 0; i < n; i++) {
        b[i] = 1;
    }

    PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
             &n, a, ia, ja, &idum, &nrhs,
             iparm, &msglvl, b, x, &error);

    if (error != 0) {
        printf("\nERROR during solution: %d", error);
        exit(3);
    }

    printf("\nSolve completed ... ");
    printf("\nThe solution of the system is: ");
    for (i = 0; i < n; i++) {
        printf("\n x [%d] = % f", i, x[i] );
    }

    printf ("\n");

/* -----*/

```

```

/* .. Termination and release of memory. */
/* -----*/
    phase = -1; /* Release internal memory. */
    PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
             &n, &ddum, ia, ja, &idum, &nrhs,
             iparm, &msglvl, &ddum, &ddum, &error);

    return 0;
}

```

Direct Sparse Solver Code Examples

This section contains example code in FORTRAN 77, Fortran 90 and C. For description of the sparse solver routines used in this code, refer to "Direct Sparse Solver (DSS) Interface Routines" in Chapter 8 of the manual. The example code solves the equations presented in Direct Method section of Appendix A - a symmetric positive definite system of equations $Ax = b$ with a sparse matrix, where

$$A = \begin{bmatrix} 9 & 1.5 & 6 & 0.75 & 3 \\ 1.5 & 0.5 & 0 & 0 & 0 \\ 6 & 0 & 12 & 0 & 0 \\ 0.75 & 0 & 0 & 0.625 & 0 \\ 3 & 0 & 0 & 0 & 16 \end{bmatrix} \text{ and } B = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

Example results for symmetric systems

Upon successful execution of the solver, the determinant and the result of the solution array are as follows

```

pow of determinant is      0.000
base of determinant is      2.250
Determinant is      2.250
Solution Array:  -326.333   983.000  163.417   398.000   61.500

```

Example C-10. FORTRAN 77 example to Solve Symmetric Positive Definite System

```
*****
*
*                               INTEL CONFIDENTIAL
*
*   Copyright(C) 2004-2008 Intel Corporation. All Rights Reserved.
*
*   The source code contained or described herein and all documents related to
*   the source code ("Material") are owned by Intel Corporation or its suppliers
*   or licensors. Title to the Material remains with Intel Corporation or its
*   suppliers and licensors. The Material contains trade secrets and proprietary
*   and confidential information of Intel or its suppliers and licensors. The
*   Material is protected by worldwide copyright and trade secret laws and
*   treaty provisions. No part of the Material may be used, copied, reproduced,
*   modified, published, uploaded, posted, transmitted, distributed or disclosed
*   in any way without Intel's prior express written permission.
*
*   No license under any patent, copyright, trade secret or other intellectual
*   property right is granted to or conferred upon you by disclosure or delivery
*   of the Materials, either expressly, by implication, inducement, estoppel or
*   otherwise. Any license under such intellectual property rights must be
*   express and approved by Intel in writing.
*
*****
*   Content : Intel MKL DSS Fortran 77 example
*
*****

C-----
C Example program for solving symmetric positive definite system of
C equations.
C-----

      PROGRAM solver_f77_test
```

```

IMPLICIT NONE

INCLUDE 'mkl_dss.f77'

C-----
C Define the array and rhs vectors
C-----

      INTEGER nRows, nCols, nNonZeros, i, nRhs
      PARAMETER (nRows = 5,
1 nCols = 5,
2 nNonZeros = 9,
3 nRhs = 1)

      INTEGER rowIndex(nRows + 1), columns(nNonZeros)
      DOUBLE PRECISION values(nNonZeros), rhs(nRows)
      DATA rowIndex / 1, 6, 7, 8, 9, 10 /
      DATA columns / 1, 2, 3, 4, 5, 2, 3, 4, 5 /
      DATA values / 9, 1.5, 6, .75, 3, 0.5, 12, .625, 16 /
      DATA rhs / 1, 2, 3, 4, 5 /

C-----
C Allocate storage for the solver handle and the solution vector
C-----

      DOUBLE PRECISION solution(nRows)
      INTEGER*8 handle
      INTEGER error
      CHARACTER*15 statIn
      DOUBLE PRECISION statOut(5)
      INTEGER bufLen
      PARAMETER(bufLen = 20)
      INTEGER buff(bufLen)

C-----

```

C Initialize the solver

```
C-----
    error = dss_create(handle, MKL_DSS_DEFAULTS)
    IF (error .NE. MKL_DSS_SUCCESS ) GOTO 999
```

C-----

C Define the non-zero structure of the matrix

```
C-----
    error = dss_define_structure( handle, MKL_DSS_SYMMETRIC,
    & rowIndex, nRows, nCols, columns, nNonZeros )
    IF (error .NE. MKL_DSS_SUCCESS ) GOTO 999
```

C-----

C Reorder the matrix

```
C-----
    error = dss_reorder( handle, MKL_DSS_DEFAULTS, 0)
    IF (error .NE. MKL_DSS_SUCCESS ) GOTO 999
```

C-----

C Factor the matrix

```
C-----
    error = dss_factor_real( handle,
    & MKL_DSS_DEFAULTS, VALUES)
    IF (error .NE. MKL_DSS_SUCCESS ) GOTO 999
```

C-----

C Get the solution vector

```
C-----
    error = dss_solve_real( handle, MKL_DSS_DEFAULTS,
    & rhs, nRhs, solution)
    IF (error .NE. MKL_DSS_SUCCESS ) GOTO 999
```

C-----

C Print Determinant of the matrix

```
C-----
      statIn = 'determinant'
      call mkl_cvt_to_null_terminated_str(buff,bufLen,statIn)
      error = dss_statistics(handle, MKL_DSS_DEFAULTS,
& buff,statOut)
      WRITE(*, "(' pow of determinant is ', 5(F10.3))") statOut(1)
      WRITE(*, "(' base of determinant is ', 5(F10.3))") statOut(2)
      WRITE(*, "(' Determinant is ', 5(F10.3))") (10**statOut(1))*
& statOut(2)
```

```
C-----
C Deallocate solver storage
```

```
C-----
      error = dss_delete( handle, MKL_DSS_DEFAULTS )
      IF (error .NE. MKL_DSS_SUCCESS ) GOTO 999
```

```
C-----
C Print solution vector
```

```
C-----
      WRITE(*,900) (solution(i), i = 1, nCols)
900 FORMAT(' Solution Array: ',5(F10.3))
      GOTO 1000
999 WRITE(*,*) "Solver returned error code ", error
1000 END
```

Example C-11. C Example to Solve Symmetric Positive Definite System

```

/*
*****
*
*                               INTEL CONFIDENTIAL
*
* Copyright(C) 2004-2008 Intel Corporation. All Rights Reserved.
*
* The source code contained or described herein and all documents related to
* the source code ("Material") are owned by Intel Corporation or its suppliers
* or licensors. Title to the Material remains with Intel Corporation or its
* suppliers and licensors. The Material contains trade secrets and proprietary
* and confidential information of Intel or its suppliers and licensors. The
* Material is protected by worldwide copyright and trade secret laws and
* treaty provisions. No part of the Material may be used, copied, reproduced,
* modified, published, uploaded, posted, transmitted, distributed or disclosed
* in any way without Intel's prior express written permission.
*
* No license under any patent, copyright, trade secret or other intellectual
* property right is granted to or conferred upon you by disclosure or delivery
* of the Materials, either expressly, by implication, inducement, estoppel or
* otherwise. Any license under such intellectual property rights must be
* express and approved by Intel in writing.
*
*****
*   Content : Intel MKL DSS C example
*
*****/

/*

** Example program to solve symmetric positive definite system of equations.
*/

```



```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include "mkl_dss.h"
/*
** Define the array and rhs vectors
*/
#define NROWS      5
#define NCOLS      5
#define NNONZEROS  9
#define NRHS       1
#if defined(MKL_ILP64)
#define MKL_INT long long
#else
#define MKL_INT int
#endif
static const MKL_INT nRows =      NROWS  ;
static const MKL_INT nCols =      NCOLS  ;
static const MKL_INT nNonZeros = NNONZEROS ;
static const MKL_INT nRhs =      NRHS   ;
static _INTEGER_t rowIndex[NROWS+1] = { 1, 6, 7, 8, 9, 10 };
static _INTEGER_t columns[NNONZEROS] = { 1, 2, 3, 4, 5, 2, 3, 4, 5 };
static _DOUBLE_PRECISION_t values[NNONZEROS] = { 9, 1.5, 6, .75, 3, 0.5, 12, .625, 16 };
static _DOUBLE_PRECISION_t rhs[NCOLS] = { 1, 2, 3, 4, 5 };

MKL_INT main() {
    MKL_INT i;
    /* Allocate storage for the solver handle and the right-hand side. */
```

```

_DOUBLE_PRECISION_t solValues[NROWS];
_MKL_DSS_HANDLE_t handle;
_INTEGER_t error;
_CHARACTER_STR_t statIn[] = "determinant";
_DOUBLE_PRECISION_t statOut[5];
MKL_INT opt = MKL_DSS_DEFAULTS;
MKL_INT sym = MKL_DSS_SYMMETRIC;
MKL_INT type = MKL_DSS_POSITIVE_DEFINITE;
/* -----*/
/* Initialize the solver */
/* -----*/
    error = dss_create(handle, opt );
    if ( error != MKL_DSS_SUCCESS ) goto printError;
/* -----*/
/* Define the non-zero structure of the matrix */
/* -----*/
    error = dss_define_structure(
        handle, sym, rowIndex, nRows, nCols,
        columns, nNonZeros );
    if ( error != MKL_DSS_SUCCESS ) goto printError;
/* -----*/
/* Reorder the matrix */
/* -----*/
    error = dss_reorder(handle, opt, 0);
    if ( error != MKL_DSS_SUCCESS ) goto printError;
/* -----*/
/* Factor the matrix */
/* -----*/

```

```
error = dss_factor_real( handle, type, values );
if ( error != MKL_DSS_SUCCESS ) goto printError;
/* -----*/
/* Get the solution vector */
/* -----*/
error = dss_solve_real( handle, opt, rhs, nRhs, solValues );
if ( error != MKL_DSS_SUCCESS ) goto printError;
/* -----*/
/* Get the determinant (not for a diagonal matrix) */
/*-----*/
if ( nRows < nNonZeros ) {
    error = dss_statistics(handle, opt, statIn, statOut);
    if ( error != MKL_DSS_SUCCESS ) goto printError;
/*-----*/
/* print determinant*/
/*-----*/
    printf(" determinant power is %g \n", statOut[0]);
    printf(" determinant base is %g \n", statOut[1]);
    printf(" Determinant is %g \n", (pow(10.0,statOut[0]))*statOut[1]);
}
/* -----*/
/* Deallocate solver storage */
/* -----*/
error = dss_delete( handle, opt );
if ( error != MKL_DSS_SUCCESS ) goto printError;
/* -----*/
/* Print solution vector */
/* -----*/
```

```
    printf(" Solution array: ");
    for(i = 0; i< nCols; i++)
        printf(" %g", solValues[i] );
    printf("\n");
    exit(0);
printError:
    printf("Solver returned error code %d\n", error);
    exit(1);
}
```

Example C-12. Fortran 90 Example to Solve Symmetric Positive Definite System

```

!*****
!
!                               INTEL CONFIDENTIAL
!
!   Copyright(C) 2004-2008 Intel Corporation. All Rights Reserved.
!
!   The source code contained or described herein and all documents related to
!   the source code ("Material") are owned by Intel Corporation or its suppliers
!   or licensors. Title to the Material remains with Intel Corporation or its
!   suppliers and licensors. The Material contains trade secrets and proprietary
!   and confidential information of Intel or its suppliers and licensors. The
!   Material is protected by worldwide copyright and trade secret laws and
!   treaty provisions. No part of the Material may be used, copied, reproduced,
!   modified, published, uploaded, posted, transmitted, distributed or disclosed
!   in any way without Intel's prior express written permission.
!
!   No license under any patent, copyright, trade secret or other intellectual
!   property right is granted to or conferred upon you by disclosure or delivery
!   of the Materials, either expressly, by implication, inducement, estoppel or
!   otherwise. Any license under such intellectual property rights must be
!   express and approved by Intel in writing.
!
!*****
!
!   Content : Intel MKL DSS Fortran 90 example
!
!*****
!-----
!
!   Example program for solving a symmetric positive definite system of
!   equations.
!

```

```

!-----
INCLUDE 'mkl_dss.f90' ! Include the standard DSS "header file."
PROGRAM solver_f90_test
use mkl_dss
IMPLICIT NONE
INTEGER, PARAMETER :: dp = KIND(1.0D0)
INTEGER :: error
INTEGER :: i
INTEGER, PARAMETER :: buflen = 20
! Define the data arrays and the solution and rhs vectors.
INTEGER, ALLOCATABLE :: columns( :
)
INTEGER :: nCols
INTEGER :: nNonZeros
INTEGER :: nRhs
INTEGER :: nRows
REAL(KIND=DP), ALLOCATABLE :: rhs( : )
INTEGER, ALLOCATABLE :: rowIndex( : )
REAL(KIND=DP), ALLOCATABLE :: solution( : )
REAL(KIND=DP), ALLOCATABLE :: values( : )
TYPE(MKL_DSS_HANDLE) :: handle ! Allocate storage for the solver handle.
REAL(KIND=DP),ALLOCATABLE::statOut( : )
CHARACTER*15 statIn
INTEGER perm(1)
INTEGER buff(buflen)
EXTERNAL MKL_CVT_TO_NULL_TERMINATED_STR
! Set the problem to be solved.
nRows = 5
nCols = 5

```

```
nNonZeros = 9
nRhs = 1
perm(1) = 0
ALLOCATE( rowIndex(nRows + 1 ) )
rowIndex = (/ 1, 6, 7, 8, 9, 10 /)
ALLOCATE( columns(nNonZeros ) )
columns = (/ 1, 2, 3, 4, 5, 2, 3, 4, 5 /)
ALLOCATE( values( nNonZeros ) )
values = (/ 9.0_DP, 1.5_DP, 6.0_DP, 0.75_DP, 3.0_DP, 0.5_DP, 12.0_DP, &
& 0.625_DP, 16.0_DP /)
ALLOCATE( rhs( nRows ) )
rhs = (/ 1.0_DP, 2.0_DP, 3.0_DP, 4.0_DP, 5.0_DP /)
! Initialize the solver.
error = dss_create( handle, MKL_DSS_DEFAULTS )
IF (error /= MKL_DSS_SUCCESS) GOTO 999
! Define the non-zero structure of the matrix.
error = dss_define_structure( handle, MKL_DSS_SYMMETRIC, rowIndex, nRows, &
& nCols, columns, nNonZeros )
IF (error /= MKL_DSS_SUCCESS) GOTO 999
! Reorder the matrix.
error = dss_reorder( handle, MKL_DSS_DEFAULTS, perm )
IF (error /= MKL_DSS_SUCCESS) GOTO 999
! Factor the matrix.
error = dss_factor_real( handle, MKL_DSS_DEFAULTS, values )
IF (error /= MKL_DSS_SUCCESS) GOTO 999
! Allocate the solution vector and solve the problem.
ALLOCATE( solution( nRows ) )
error = dss_solve_real(handle, MKL_DSS_DEFAULTS, rhs, nRhs, solution )
```

```

IF (error /= MKL_DSS_SUCCESS) GOTO 999
! Print Out the determinant of the matrix
ALLOCATE(statOut( 5 ) )
statIn = 'determinant'
call mkl_cvt_to_null_terminated_str(buff,bufLen,statIn)
error = dss_statistics(handle, MKL_DSS_DEFAULTS, buff, statOut )
IF (error /= MKL_DSS_SUCCESS) GOTO 999
WRITE(*, "('pow of determinant is '(5F10.3))") ( statOut(1) )
WRITE(*, "('base of determinant is '(5F10.3))") ( statOut(2) )
WRITE(*, "('Determinant is '(5F10.3))") ( (10**statOut(1))*statOut(2) )
END IF

! Deallocate solver storage and various local arrays.
error = DSS_DELETE( handle, MKL_DSS_DEFAULTS )
IF (error /= MKL_DSS_SUCCESS ) GOTO 999
IF ( ALLOCATED( rowIndex ) ) DEALLOCATE( rowIndex )
IF ( ALLOCATED( columns ) ) DEALLOCATE( columns )
IF ( ALLOCATED( values ) ) DEALLOCATE( values )
IF ( ALLOCATED( rhs ) ) DEALLOCATE( rhs )
IF ( ALLOCATED( statOut ) ) DEALLOCATE( statOut )
! Print the solution vector, deallocate it and exit
WRITE(*, "('Solution Array: '(5F10.3))") ( solution(i), i = 1, nCols )
IF ( ALLOCATED( solution ) ) DEALLOCATE( solution )
GOTO 1000

! Print an error message and exit
999 WRITE(*,*) "Solver returned error code ", error
1000 CONTINUE
END PROGRAM solver_f90_test

```


Iterative Sparse Solver Code Examples

This section contains example code in FORTRAN 77 and C. For description of the iterative sparse solver routines based on the reverse communication interface (RCI ISS) used in this code, refer to "Iterative Sparse Solvers based on Reverse Communication Interface (RCI ISS)" in Chapter 8 of the manual .

Example of Use RCI (Preconditioned) Conjugate Gradient Solver for Systems with Single Right Hand Side

Example results for symmetric positive definite systems. Upon successful execution of the solver, the result of the solution array is as follows:

The system is successfully solved

The following solution obtained

1.000	0.000	1.000	0.000
1.000	0.000	1.000	0.000

Expected solution

1.000	0.000	1.000	0.000
1.000	0.000	1.000	0.000

Number of iterations: 8

Example C-13. FORTRAN 77 Example to Solve Symmetric Positive Definite System

```
*****
*
*                               INTEL CONFIDENTIAL
*
* Copyright(C) 2005-2008 Intel Corporation. All Rights Reserved.
*
* The source code contained or described herein and all documents related to
* the source code ("Material") are owned by Intel Corporation or its suppliers
* or licensors. Title to the Material remains with Intel Corporation or its
* suppliers and licensors. The Material contains trade secrets and proprietary
* and confidential information of Intel or its suppliers and licensors. The
* Material is protected by worldwide copyright and trade secret laws and
* treaty provisions. No part of the Material may be used, copied, reproduced,
* modified, published, uploaded, posted, transmitted, distributed or disclosed
* in any way without Intel's prior express written permission.
*
* No license under any patent, copyright, trade secret or other intellectual
* property right is granted to or conferred upon you by disclosure or delivery
* of the Materials, either expressly, by implication, inducement, estoppel or
* otherwise. Any license under such intellectual property rights must be
* express and approved by Intel in writing.
*
*****
*
* Content : Intel MKL RCI (P)CG Fortran 77 example
*
*****
*
C-----
C Example program for solving symmetric positive definite system of
C equations.
```

```

C-----
      PROGRAM rci_pcg_f77_test
      IMPLICIT NONE
C-----
C Define arrays for the upper triangle of the coefficient matrix and rhs vector
C Compressed sparse row storage is used for sparse representation
C-----

      INTEGER N, RCI_request, itercount, i
      PARAMETER (N=8)
      DOUBLE PRECISION  rhs(N), solution(N)

      INTEGER ia(9)
      INTEGER ja(18)
      DOUBLE PRECISION a(18)
C Fill all arrays containing matrix data.
      DATA ia /1,5,8,10,12,15,17,18,19/
      DATA ja
1 /1,  3,  6,7,
2      2,  3,  5,
3      3,           8,
4          4,      7,
5          5,6,7,
6          6,  8,
7          7,
8          8/
      DATA a
1 /7.D0,      1.D0,      2.D0,  7.D0,
2      -4.D0,  8.D0,      2.D0,

```

```

3          1.D0,          5.D0,
4          7.D0,          9.D0,
5          5.D0,  1.D0,  5.D0,
6          -1.D0,          5.D0,
7
8
                        11.D0,
                        5.D0/

```

C-----

C Allocate storage for the solver ?par and the initial solution vector

C-----

```

      INTEGER length
      PARAMETER (length=128)
      DOUBLE PRECISION expected(N)
      DATA expected/1.D0, 0.D0, 1.D0, 0.D0, 1.D0, 0.D0, 1.D0, 0.D0/
      INTEGER ipar(length)
      DOUBLE PRECISION dpar(length),tmp(N,4)

```

C-----

C Initialize the right hand side through matrix-vector product

C-----

```

      CALL DCSRMV_SY('U', N, A, IA, JA, expected, rhs)

```

C-----

C Initialize the initial guess

C-----

```

      DO I=1, N
        solution(I)=1.D0
      ENDDO

```

C-----

C Initialize the solver

C-----

```
      CALL dcg_init(N,solution,rhs,RCI_request,ipar,dpar,tmp)
      IF (RCI_request .NE. 0 ) GOTO 999
C-----
C Set the desired parameters:
C LOGICAL parameters:
C do residual stopping test
C do not request for the user defined stopping test
C do Preconditioned Conjugate Gradient iterations
C DOUBLE PRECISION parameters
C set the relative tolerance to 1.0D-5 instead of default value 1.0D-6
C-----
      ipar(9)=1
      ipar(10)=0
      ipar(11)=1
      dpar(1)=1.D-5
C-----
C Check the correctness and consistency of the newly set parameters
C-----
      CALL dcg_check(N,solution,rhs,RCI_request,ipar,dpar,tmp)
      IF (RCI_request .NE. 0 ) GOTO 999
C-----
C Compute the solution by RCI PCG solver
C Reverse Communications starts here
C-----
1      CALL dcg(N,solution,rhs,RCI_request,ipar,dpar,tmp)
C-----
C If RCI_request=0, then the solution was found with the required precision
C-----
```

```

      IF (RCI_request .EQ. 0) THEN
          GOTO 700
C-----
C If RCI_request=1, then compute the vector A*tmp(:,1)
C and put the result in vector tmp(:,2)
C-----
      ELSE IF (RCI_request .EQ. 1) THEN
          CALL DCSRMV_SY('U', N, A, IA, JA, TMP, TMP(1, 2))
          GOTO 1
C-----
C If RCI_request=3, then compute vector preconditioner matrix on tmp(:,3)
C and put the result in vector tmp(:,4)
C-----
      ELSE IF (RCI_request .EQ. 3) THEN
          CALL DCOPY(N, TMP(1,3),1, TMP(1, 4), 1)
          GOTO 1
      ELSE
C-----
C If RCI_request=anything else, then dcg subroutine failed
C to compute the solution vector: solution(N)
C-----
          GOTO 999
      ENDIF
C-----
C Reverse Communication ends here
C Get the current iteration number
C-----
700  CALL dcg_get(N,solution,rhs,RCI_request,ipar,dpar,tmp,

```

```
        &                                itercount)
C-----
C Print solution vector: solution(N) and number of iterations: itercount
C-----
        WRITE(*, *) ' The system is successfully solved '
        WRITE(*, *) ' The following solution obtained '
        WRITE(*,800) (solution(i),i =1,N)
        WRITE(*, *) ' Expected solution '
        WRITE(*,800) (expected(i),i =1,N)
800   FORMAT(4(F10.3))
        WRITE(*,900) (itercount)
900   FORMAT(' Number of iterations: ',1(I2))
        GOTO 1000
999   WRITE(*,*) 'Solver returned error code ', RCI_request
        STOP
1000
CONTINUE
        read *
        END
```

Examples of Use RCI (Preconditioned) Conjugate Gradient Solver for Systems with Multiple Right Hand Sizes (MRHS)

Example results for symmetric positive definite systems. Upon successful execution of the solver, the result of the solution array is as follows:

The system is successfully solved

The following solution obtained

1.000	0.000	1.000	0.000
1.000	0.000	1.000	0.000
0.000	2.000	0.000	2.000
0.000	2.000	0.000	2.000

Expected solution

1.000	0.000	1.000	0.000
1.000	0.000	1.000	0.000
0.000	2.000	0.000	2.000
0.000	2.000	0.000	2.000

Example C-14a. Fortran Example to Solve Symmetric Positive Definite System with Multiple Right Hand Sizes (MRHS). No Preconditioning, No User-Defined Stopping Tests.

```

!*****
!
!                               INTEL CONFIDENTIAL
!
!   Copyright(C) 2005-2008 Intel Corporation. All Rights Reserved.
!
!   The source code contained or described herein and all documents related to
!   the source code ("Material") are owned by Intel Corporation or its suppliers
!   or licensors. Title to the Material remains with Intel Corporation or its
!   suppliers and licensors. The Material contains trade secrets and proprietary
!   and confidential information of Intel or its suppliers and licensors. The
!   Material is protected by worldwide copyright and trade secret laws and
!   treaty provisions. No part of the Material may be used, copied, reproduced,
!   modified, published, uploaded, posted, transmitted, distributed or disclosed
!   in any way without Intel's prior express written permission.
!
!   No license under any patent, copyright, trade secret or other intellectual
!   property right is granted to or conferred upon you by disclosure or delivery
!   of the Materials, either expressly, by implication, inducement, estoppel or
!   otherwise. Any license under such intellectual property rights must be
!   express and approved by Intel in writing.
!
!*****
!   Content: Intel MKL RCI (P)CG with MRHS Fortran-77 example
!
!*****
C-----
C   Example program for solving symmetric positive definite system of equations.
C   Simplest case: no preconditioning and no the user-defined stopping tests.
C-----

```

```

PROGRAM rci_pcgmrhs_f77_test_1

IMPLICIT NONE

INCLUDE 'mkl_solver.f77'

C-----
C Define arrays for the upper triangle of the coefficient matrix and rhs vector
C Compressed sparse row storage is used for sparse representation
C-----

INTEGER N, nRhs
PARAMETER (N=8,nRhs=2)

INTEGER RCI_request, itercount(nRhs), i
DOUBLE PRECISION rhs(N,nRhs)

C Matrix

INTEGER IA(9)
INTEGER JA(18)
DOUBLE PRECISION A(18)

C Fill all arrays containing matrix data.

DATA IA /1,5,8,10,12,15,17,18,19/
DATA JA
1 /1, 3, 6,7,
2 2,3, 5,
3 3, 8,
4 4, 7,
5 5,6,7,
6 6, 8,
7 7,
8 8/

DATA A
1 /7.D0, 1.D0, 2.D0, 7.D0,
```

```

2      -4.D0,8.D0,      2.D0,
3          1.D0,          5.D0,
4          7.D0,      9.D0,
5          5.D0, 1.D0, 5.D0,
6          -1.D0,      5.D0,
7          11.D0,
8          5.D0/

C-----
C Allocate storage for the solver ?par and the initial solution vector
C-----

      INTEGER length, method
      PARAMETER (length=128, method=1)
      DOUBLE PRECISION expected_sol(N,nRhs)
      DOUBLE PRECISION solution(N,nRhs)
      INTEGER ipar(length + 2*nRhs)
      DOUBLE PRECISION dpar(length + 2*nRhs),TMP(N,3 + nRhs)

C-----
C Some additional variables to use with the RCI (P)CG MRHS solver
C-----

      DATA expected_sol/1.D0, 0.D0, 1.D0, 0.D0, 1.D0, 0.D0, 1.D0, 0.D0,
+          0.D0, 2.D0, 0.D0, 2.D0, 0.D0, 2.D0, 0.D0, 2.D0/

C-----
C Initialize the right hand side through matrix-vector product
C-----

      CALL MKL_DCSRYSYMV('U', N, A, IA, JA, expected_sol, rhs)
      CALL MKL_DCSRYSYMV('U', N, A, IA, JA, expected_sol(1,2), rhs(1,2))

C-----
C Initialize the initial guess

```

```
C-----
      DO I=1, N
        solution(I,1)=1.D0
        solution(I,2)=2.D0
      ENDDO

C-----
C Initialize the solver
C-----

      DO I=1, length + 2*nRhs
        ipar(i) = 0;
        dpar(i) = 0;
      ENDDO

      CALL dcgmrhs_init(N,solution,nRhs,rhs,method,
+          RCI_request,ipar,dpar,TMP)
      IF (RCI_request .NE. 0 ) GOTO 999

C-----
C Set the desired parameters:
C LOGICAL parameters:
C do residual stopping test
C do not request for the user defined stopping test
C DOUBLE PRECISION parameters
C set the relative tolerance to 1.0D-5 instead of default value 1.0D-6
C-----
      ipar(9) =1
      ipar(10)=0
      dpar(1) =1.D-5

C-----
C Compute the solution by RCI (P)CG MRHS solver without preconditioning
```

```

C Reverse Communications starts here
C-----
1      CALL dcgmrhs(N,solution,nRhs,rhs,RCI_request,ipar,dpar,TMP)
C-----
C If RCI_request=0, then the solution was found with the required precision
C-----
      IF (RCI_request .EQ. 0) THEN
          GOTO 700
C-----
C If RCI_request=1, then compute the vector A*TMP(:,1)
C and put the result in vector TMP(:,2)
C-----
      ELSEIF (RCI_request .EQ. 1) THEN
          CALL MKL_DCSRSMV('U', N, A, IA, JA, TMP, TMP(1,2))
          GOTO 1
      ELSE
C-----
C If RCI_request=anything else, then dcg subroutine failed
C to compute the solution vector: solution
C-----
          GOTO 999
      ENDIF
C-----
C Reverse Communication ends here
C Get the current iteration number
C-----
700    CALL dcgmrhs_get(N,solution,nRhs,rhs,RCI_request,ipar,dpar,TMP,
      &                  itercount)

```

```
C-----  
C Print solution vector: solution(N) and number of iterations: itercount  
C-----  
      WRITE(*, *) ' The system is successfully solved '  
      WRITE(*, *) ' The following solution obtained '  
      WRITE(*,800) (solution(i,1),i =1,N)  
      WRITE(*,800) (solution(i,2),i =1,N)  
      WRITE(*, *) ' expected_sol solution '  
      WRITE(*,800) (expected_sol(i,1),i =1,N)  
      WRITE(*,800) (expected_sol(i,2),i =1,N)  
800  FORMAT(4(F10.3))  
      GOTO 1000  
999  WRITE(*,*) 'Solver returned error code ', RCI_request  
      STOP  
1000 CONTINUE  
      STOP  
      END
```

Example C-14b. Fortran Example to Solve Symmetric Positive Definite System with Multiple Right Hand Sizes (MRHS). Simple Preconditioning, No User-Defined Stopping Tests.

```

!*****
!
!                               INTEL CONFIDENTIAL
!
!   Copyright(C) 2005-2008 Intel Corporation. All Rights Reserved.
!
!   The source code contained or described herein and all documents related to
!   the source code ("Material") are owned by Intel Corporation or its suppliers
!   or licensors. Title to the Material remains with Intel Corporation or its
!   suppliers and licensors. The Material contains trade secrets and proprietary
!   and confidential information of Intel or its suppliers and licensors. The
!   Material is protected by worldwide copyright and trade secret laws and
!   treaty provisions. No part of the Material may be used, copied, reproduced,
!   modified, published, uploaded, posted, transmitted, distributed or disclosed
!   in any way without Intel's prior express written permission.
!
!   No license under any patent, copyright, trade secret or other intellectual
!   property right is granted to or conferred upon you by disclosure or delivery
!   of the Materials, either expressly, by implication, inducement, estoppel or
!   otherwise. Any license under such intellectual property rights must be
!   express and approved by Intel in writing.
!
!*****
!   Content: Intel MKL RCI (P)CG MRHS Fortran-77 example
!
!*****
C-----
C   Example program for solving symmetric positive definite system of equations.
C   Simplest case: simple preconditioning and no the user-defined stopping tests.
C-----

```

```

PROGRAM rci_pcgmrhs_f77_precond
IMPLICIT NONE

INCLUDE 'mkl_solver.f77'
C-----
C Define arrays for the upper triangle of the coefficient matrix and rhs vector
C Compressed sparse row storage is used for sparse representation
C-----

INTEGER N, nRhs
PARAMETER (N=8,nRhs=2)
INTEGER RCI_request, itercount(nRhs), i
DOUBLE PRECISION rhs(N,nRhs)

C Matrix
INTEGER IA(9)
INTEGER JA(18)
DOUBLE PRECISION A(18)
C Fill all arrays containing matrix data.
DATA IA /1,5,8,10,12,15,17,18,19/
DATA JA
1 /1, 3, 6,7,
2 2,3, 5,
3 3, 8,
4 4, 7,
5 5,6,7,
6 6, 8,
7 7,
8 8/

```



```

DATA A
1 /7.D0,      1.D0,      2.D0, 7.D0,
2      -4.D0,8.D0,      2.D0,
3          1.D0,              5.D0,
4          7.D0,      9.D0,
5          5.D0, 1.D0, 5.D0,
6          -1.D0,      5.D0,
7          11.D0,
8          5.D0/

C-----
C Allocate storage for the solver ?par and the initial solution vector
C-----

INTEGER length, method
PARAMETER (length=128, method=1)
DOUBLE PRECISION expected_sol(N,nRhs)
DOUBLE PRECISION solution(N,nRhs)
INTEGER ipar(length + 2*nRhs)
DOUBLE PRECISION dpar(length + 2*nRhs),TMP(N,3 + nRhs)

C-----
C Some additional variables to use with the RCI (P)CG MRHS solver
C-----

DATA expected_sol/1.D0, 0.D0, 1.D0, 0.D0, 1.D0, 0.D0, 1.D0, 0.D0,
+          0.D0, 2.D0, 0.D0, 2.D0, 0.D0, 2.D0, 0.D0, 2.D0/

C-----
C Initialize the right hand side through matrix-vector product
C-----

CALL MKL_DCSRYSMV('U', N, A, IA, JA, expected_sol, rhs)
CALL MKL_DCSRYSMV('U', N, A, IA, JA, expected_sol(1,2), rhs(1,2))

```

```

C-----
C Initialize the initial guess
C-----
      DO I=1, N
          solution(I,1)=1.D0
          solution(I,2)=2.D0
      ENDDO

C-----
C Initialize the solver
C-----
      DO I=1, length + 2*nRhs
          ipar(i) = 0;
          dpar(i) = 0;
      ENDDO

      CALL dcgmrhs_init(N,solution,nRhs,rhs,method,
+          RCI_request,ipar,dpar,TMP)
      IF (RCI_request .NE. 0 ) GOTO 999

C-----
C Set the desired parameters:
C INTEGER parameters:
C set the maximal number of iterations to 100
C LOGICAL parameters:
C run the Preconditioned version of RCI (P)CG MRHS with preconditioner
C DOUBLE PRECISION parameters
C -
C-----
      ipar(5) = 100
      ipar(11) = 1

```

```

    ipar(9)  = 1
    ipar(10) = 0
    dpar(1) =1.D-5
C-----
C Compute the solution by RCI (P)CG MRHS solver
C Reverse Communications starts here
C-----
1    CALL dcgmrhs(N,solution,nRhs,rhs,RCI_request,ipar,dpar,TMP)
C-----
C If RCI_request=0, then the solution was found with the required precision
C-----
    IF (RCI_request .EQ. 0) THEN
        GOTO 700
C-----
C If RCI_request=1, then compute the vector A*TMP(:,1)
C and put the result in vector TMP(:,2)
C-----
    ELSEIF (RCI_request .EQ. 1) THEN
        CALL MKL_DCSRSYMV('U', N, A, IA, JA, TMP, TMP(1,2))
        GOTO 1
C-----
C If RCI_request=3, then  apply the simplest preconditioning
C on vector TMP(:,3+par(3)) and put the result in vector TMP(:,3)
C-----
    ELSEIF (RCI_request .EQ. 3) THEN
        do i = 1, n
            tmp(i, 3) = tmp (i, 3+ipar(3));
        end do

```

```

        GOTO 1
    ELSE
C-----
C If RCI_request=anything else, then dcg subroutine failed
C to compute the solution vector: solution(N)
C-----
        GOTO 999
    ENDIF
C-----
C Reverse Communication ends here
C Get the current iteration number
C-----
700    CALL dcgmrhs_get(N,solution,nRhs,rhs,RCI_request,ipar,dpar,TMP,
        &                itercount)
C-----
C Print solution vector: solution(N) and number of iterations: itercount
C-----
        WRITE(*, *) ' The system is successfully solved '
        WRITE(*, *) ' The following solution obtained '
        WRITE(*,800) (solution(i,1),i =1,N)
        WRITE(*,800) (solution(i,2),i =1,N)
        WRITE(*, *) ' expected_sol solution '
        WRITE(*,800) (expected_sol(i,1),i =1,N)
        WRITE(*,800) (expected_sol(i,2),i =1,N)
800    FORMAT(4(F10.3))
        GOTO 1000
999    WRITE(*,*) 'Solver returned error code ', RCI_request
        STOP

```

```
1000  CONTINUE
      STOP
      END
```

Example C-14c. Fortran Example to Solve Symmetric Positive Definite System with Multiple Right Hand Sizes (MRHS). User-Defined Stopping Tests, No Preconditioning.

```
!*****
!
!                               INTEL CONFIDENTIAL
!
!   Copyright(C) 2005-2008 Intel Corporation. All Rights Reserved.
!
!   The source code contained or described herein and all documents related to
!   the source code ("Material") are owned by Intel Corporation or its suppliers
!   or licensors. Title to the Material remains with Intel Corporation or its
!   suppliers and licensors. The Material contains trade secrets and proprietary
!   and confidential information of Intel or its suppliers and licensors. The
!   Material is protected by worldwide copyright and trade secret laws and
!   treaty provisions. No part of the Material may be used, copied, reproduced,
!   modified, published, uploaded, posted, transmitted, distributed or disclosed
!   in any way without Intel's prior express written permission.
!
!   No license under any patent, copyright, trade secret or other intellectual
!   property right is granted to or conferred upon you by disclosure or delivery
!   of the Materials, either expressly, by implication, inducement, estoppel or
!   otherwise. Any license under such intellectual property rights must be
!   express and approved by Intel in writing.
!
!*****
!   Content: Intel MKL RCI (P)CG MRHS Fortran-77 example
!
!*****
C-----
C   Example program for solving symmetric positive definite system of equations.
C   Simplest case: no preconditioning and the user-defined stopping tests.
C-----
```

```

PROGRAM rci_pcgmrhs_f77_stop_crt
IMPLICIT NONE

INCLUDE 'mkl_solver.f77'

C-----
C Define arrays for the upper triangle of the coefficient matrix and rhs vector
C Compressed sparse row storage is used for sparse representation
C-----

INTEGER N, nRhs
PARAMETER (N=8,nRhs=2)
INTEGER RCI_request, itercount(nRhs), i
DOUBLE PRECISION rhs(N,nRhs)

C Matrix
INTEGER IA(9)
INTEGER JA(18)
DOUBLE PRECISION A(18)

C Fill all arrays containing matrix data.
DATA IA /1,5,8,10,12,15,17,18,19/
DATA JA
1 /1, 3, 6,7,
2 2,3, 5,
3 3, 8,
4 4, 7,
5 5,6,7,
6 6, 8,
7 7,
8 8/
DATA A

```

```

1  /7.D0,      1.D0,      2.D0, 7.D0,
2      -4.D0,8.D0,      2.D0,
3          1.D0,                      5.D0,
4              7.D0,      9.D0,
5                  5.D0, 1.D0, 5.D0,
6                      -1.D0,      5.D0,
7                          11.D0,
8                              5.D0/

```

C-----

C Allocate storage for the solver ?par and the initial solution vector

C-----

```

    INTEGER length, method
    PARAMETER (length=128, method=1)
    DOUBLE PRECISION expected_sol(N,nRhs)
    DOUBLE PRECISION solution(N,nRhs)

    INTEGER ipar(length + 2*nRhs)
    DOUBLE PRECISION dpar(length + 2*nRhs),TMP(N,3 + nRhs)

```

C-----

C Some additional variables to use with the RCI (P)CG MRHS solver

C-----

```

    DATA expected_sol/1.D0, 0.D0, 1.D0, 0.D0, 1.D0, 0.D0, 1.D0, 0.D0,
+          0.D0, 2.D0, 0.D0, 2.D0, 0.D0, 2.D0, 0.D0, 2.D0/

    DOUBLE PRECISION DNRM2, Euclidean_norm, temp(N)
    EXTERNAL DNRM2

```

C-----

C Initialize the right hand side through matrix-vector product


```

C-----
      CALL MKL_DCSRSYMV('U', N, A, IA, JA, expected_sol, rhs)
      CALL MKL_DCSRSYMV('U', N, A, IA, JA, expected_sol(1,2), rhs(1,2))
C-----
C Initialize the initial guess
C-----
      DO I=1, N
          solution(I,1)=1.D0
          solution(I,2)=2.D0
      ENDDO
C-----
C Initialize the solver
C-----
      DO I=1, length + 2*nRhs
          ipar(i) = 0;
          dpar(i) = 0;
      ENDDO
      CALL dcgmrhs_init(N,solution,nRhs,rhs,method,
+          RCI_request,ipar,dpar,TMP)
      IF (RCI_request .NE. 0 ) GOTO 999
C-----
C Set the desired parameters:
C LOGICAL parameters:
C set the maximal number of iterations to 100
C request for the user defined stopping test
C-----
      ipar(5) = 100
      ipar(10) = 1

```

```

        dpar(1) = 1.d-5
C-----
C Compute the solution by RCI (P)CG MRHS solver without preconditioning
C Reverse Communications starts here
C-----
1      CALL dcgmrhs(N,solution,nRhs,rhs,RCI_request,ipar,dpar,TMP)
C-----
C If RCI_request=0, then the solution was found with the required precision
C-----
        IF (RCI_request .EQ. 0) THEN
            GOTO 700
C-----
C If RCI_request=1, then compute the vector A*TMP(:,1)
C and put the result in vector TMP(:,2)
C-----
        ELSEIF (RCI_request .EQ. 1) THEN
            CALL MKL_DCSRSYMV('U', N, A, IA, JA, TMP, TMP(1,2))
            GOTO 1
C-----
C If RCI_request=2, then do the user-defined stopping test: compute the
C Euclidean norm of the actual residual using MKL routines and check if
C it is less than 1.D-8
C-----
        ELSEIF (RCI_request .EQ. 2) THEN
            CALL MKL_DCSRSYMV('U', N, A, IA, JA, solution(1,ipar(3)), temp)
            CALL DAXPY(N,-1.D0,rhs(1,ipar(3)),1,temp,1)
            Euclidean_norm = DNRM2(N,temp,1)
            IF (Euclidean_norm .GT. 1.D-4) THEN

```

```

C-----
C The solution has not been found yet according to the user-defined stopping
C test. Continue RCI (P)CG MRHS iterations.
C-----

        GOTO 1
    ELSE
        GOTO 700
    END IF
ELSE
    GOTO 999
ENDIF

C-----
C Reverse Communication ends here
C Get the current iteration number
C-----
700  CALL dcgmrhs_get(N,solution,nRhs,rhs,RCI_request,ipar,dpar,TMP,
        &            itercount)
C-----
C Print solution vector: solution(N) and number of iterations: itercount
C-----

    WRITE(*, *) ' The system is successfully solved '
    WRITE(*, *) ' The following solution obtained '
    WRITE(*,800) (solution(i,1),i =1,N)
    WRITE(*,800) (solution(i,2),i =1,N)
    WRITE(*, *) ' expected_sol solution '
    WRITE(*,800) (expected_sol(i,1),i =1,N)
    WRITE(*,800) (expected_sol(i,2),i =1,N)
800  FORMAT(4(F10.3))

```

```

        GOTO 1000
999    WRITE(*,*) 'Solver returned error code ', RCI_request
        STOP
1000   CONTINUE
        STOP
        END

```

Fortran Example of Using RCI (Preconditioned) Flexible Generalized Minimal Residual Solver.

Fortran example results for a non-symmetric indefinite system. Upon successful execution of the solver, the following result is printed (up to rounding errors that depend on the computer system used):

```

-----
The SIMPLEST example of usage of RCI FGMRES solver
to solve a non-symmetric indefinite non-degenerate
algebraic system of linear equations
-----

Some info about the current run of RCI FGMRES method:
As IPAR(8)=1, the automatic test for the maximal number of
iterations will be performed
As IPAR(9)=1, the automatic residual test will be performed
+++
As IPAR(10)=0, the user-defined stopping test will not be
requested, thus, RCI_REQUEST will not take the value 2
+++
As IPAR(11)=0, the Preconditioned FGMRES iterations will not
be performed, thus, RCI_REQUEST will not take the value 3
+++
As IPAR(12)=1, the automatic test for the norm of the next
generated vector is not equal to zero up to rounding and computational

```

errors will be performed, thus, RCI_REQUEST will not take the value 4

+++

The system has been SUCCESSFULLY solved

The following solution has been obtained:

COMPUTED_SOLUTION(1)=-0.100E+01

COMPUTED_SOLUTION(2)= 0.100E+01

COMPUTED_SOLUTION(3)= 0.305E-15

COMPUTED_SOLUTION(4)= 0.100E+01

COMPUTED_SOLUTION(5)=-0.100E+01

The expected solution is:

EXPECTED_SOLUTION(1)=-0.100E+01

EXPECTED_SOLUTION(2)= 0.100E+01

EXPECTED_SOLUTION(3)= 0.000E+00

EXPECTED_SOLUTION(4)= 0.100E+01

EXPECTED_SOLUTION(5)=-0.100E+01

+++

Number of iterations: 5

+++

Example C-15a. Fortran Example to Solve Non-Symmetric Indefinite System

```

C*****
C
C                                INTEL CONFIDENTIAL
C Copyright(C) 2005-2008 Intel Corporation. All Rights Reserved.
C The source code contained or described herein and all documents related to
C the source code ("Material") are owned by Intel Corporation or its suppliers
C or licensors. Title to the Material remains with Intel Corporation or its
C suppliers and licensors. The Material contains trade secrets and proprietary
C and confidential information of Intel or its suppliers and licensors. The
C Material is protected by worldwide copyright and trade secret laws and
C treaty provisions. No part of the Material may be used, copied, reproduced,
C modified, published, uploaded, posted, transmitted, distributed or disclosed
C in any way without Intel's prior express written permission.
C No license under any patent, copyright, trade secret or other intellectual
C property right is granted to or conferred upon you by disclosure or delivery
C of the Materials, either expressly, by implication, inducement, estoppel or
C otherwise. Any license under such intellectual property rights must be
C express and approved by Intel in writing.
C
C*****
C Content:
C Intel MKL RCI (P)FGMRES ((Preconditioned) Flexible Generalized Minimal
C RESidual method) example
C*****
C
C-----
C Example program for solving non-symmetric indefinite system of equations
C Simplest case: no preconditioning and no user-defined stopping tests

```

```

C-----
      PROGRAM FGMRES_NO_PRECON_F

      INCLUDE "mkl_rci.fi"

      INTEGER N
      PARAMETER(N=5)
      INTEGER SIZE
      PARAMETER (SIZE=128)

C-----
C Define arrays for the upper triangle of the coefficient matrix
C Compressed sparse row storage is used for sparse representation
-----

      INTEGER IA(6)
      DATA IA /1,3,6,9,12,14/
      INTEGER JA(13)
      DATA JA / 1,      3,
1             1,  2,      4,
2                    2,  3,      5,
3                          3,  4,      5,
4                                4,  5 /

      DOUBLE PRECISION A(13)
      DATA A / 1.0,      -1.0,
1             -1.0,  1.0,      -1.0,
2                    1.0,-2.0,      1.0,
3             -1.0,  2.0, -1.0,
4                                -1.0, -3.0 /

```

```

C-----
C Allocate storage for the ?par parameters and the solution/rhs vectors
C-----

      INTEGER IPAR(SIZE)

      DOUBLE PRECISION DPAR(SIZE), TMP(N*(2*N+1)+(N*(N+9))/2+1)

      DOUBLE PRECISION EXPECTED_SOLUTION(N)

      DATA EXPECTED_SOLUTION /-1.0,1.0,0.0,1.0,-1.0/

      DOUBLE PRECISION RHS(N)

      DOUBLE PRECISION COMPUTED_SOLUTION(N)

C-----
C Some additional variables to use with the RCI (P)FGMRES solver
C-----

      INTEGER ITERCOUNT

      INTEGER RCI_REQUEST, I

      PRINT *, '-----'
      PRINT *, 'The SIMPLEST example of usage of RCI FGMRES solver'
      PRINT *, 'to solve a non-symmetric indefinite non-degenerate'
      PRINT *, '      algebraic system of linear equations'
      PRINT *, '-----'

C-----
C Initialize variables and the right hand side through matrix-vector product
C-----

      CALL MKL_DCSRGEMV('N', N, A, IA, JA, EXPECTED_SOLUTION, RHS)

C-----
C Initialize the initial guess
C-----

      DO I=1,N

```



```

        COMPUTED_SOLUTION(I)=1.0
    ENDDO

C-----
C Initialize the solver
C-----

        CALL DFGMRES_INIT(N, COMPUTED_SOLUTION, RHS, RCI_REQUEST, IPAR,
    1 DPAR, TMP)
        IF (RCI_REQUEST.NE.0) GOTO 999

C-----
C Set the desired parameters:
C LOGICAL parameters:
C do residual stopping test
C do not request for the user defined stopping test
C do the check of the norm of the next generated vector automatically
C DOUBLE PRECISION parameters
C set the relative tolerance to 1.0D-3 instead of default value 1.0D-6
C-----

        IPAR(9)=1
        IPAR(10)=0
        IPAR(12)=1
        DPAR(1)=1.0D-3

C-----
C Check the correctness and consistency of the newly set parameters
C-----

        CALL DFGMRES_CHECK(N, COMPUTED_SOLUTION, RHS, RCI_REQUEST,
    1 IPAR, DPAR, TMP)
        IF (RCI_REQUEST.NE.0) GOTO 999

```

C-----

C Print the info about the RCI FGMRES method

C-----

```

PRINT *, ''
PRINT *, 'Some info about the current run of RCI FGMRES method:'
PRINT *, ''
IF (IPAR(8).NE.0) THEN
    WRITE(*, '(A,I1,A)') 'As IPAR(8)=' ,IPAR(8), ', the automatic',
1 ' test for the maximal number of iterations will be'
    PRINT *, 'performed'
ELSE
    WRITE(*, '(A,I1,A)') 'As IPAR(8)=' ,IPAR(8), ', the automatic ',
1 ' test for the maximal number of iterations will be'
    PRINT *, 'skipped'
ENDIF
PRINT *, '+++'
IF (IPAR(9).NE.0) THEN
    WRITE(*, '(A,I1,A)') 'As IPAR(9)=' ,IPAR(9), ', the automatic',
1 ' residual test will be performed'
ELSE
    WRITE(*, '(A,I1,A)') 'As IPAR(9)=' ,IPAR(9), ', the automatic',
1 ' residual test will be skipped'
ENDIF
PRINT *, '+++'
IF (IPAR(10).NE.0) THEN
    WRITE(*, '(A,I1,A)') 'As IPAR(10)=' ,IPAR(10), ', the user-defined',
1 ' stopping test will be requested via'
    PRINT *, 'RCI_REQUEST=2'

```

```

ELSE
    WRITE(*,'(A,I1,A)') 'As IPAR(10)=' ,IPAR(10) ,', the user-defined',
1 ' stopping test will not be requested, thus,'
    PRINT *,'RCI_REQUEST will not take the value 2'
ENDIF
PRINT *,'+++'
IF (IPAR(11).NE.0) THEN
    WRITE(*,'(A,I1,A)') 'As IPAR(11)=' ,IPAR(11) ,', the',
1 ' Preconditioned FGMRES iterations will be performed, thus,'
    WRITE(*,'(A,A)') 'the preconditioner action will be requested via',
1 '
RCI_REQUEST=3'
ELSE
    WRITE(*,'(A,I1,A)') 'As IPAR(11)=' ,IPAR(11) ,', the',
1 ' Preconditioned FGMRES iterations will not be performed,'
    WRITE(*,'(A)') 'thus, RCI_REQUEST will not take the value 3'
ENDIF
PRINT *,'+++'
IF (IPAR(12).NE.0) THEN
    WRITE(*,'(A,I1,A)') 'As IPAR(12)=' ,IPAR(12) ,', the automatic',
1 ' test for the norm of the next generated vector is'
    WRITE(*,'(A,A)') 'not equal to zero up to rounding and computational',
1 ' errors will be performed,'
    PRINT *,'thus, RCI_REQUEST will not take the value 4'
ELSE
    WRITE(*,'(A,I1,A)') 'As IPAR(12)=' ,IPAR(12) ,', the automatic',
1 ' test for the norm of the next generated vector is'
    WRITE(*,'(A,A)') 'not equal to zero up to rounding and computational',
1 ' errors will be skipped,'

```

```

        WRITE(*,'(A,A)') 'thus, the user-defined test will be requested via',
1  ' RCI_REQUEST=4'
        ENDIF
        PRINT *, '+++'

C-----
C Compute the solution by RCI (P)FGMRES solver without preconditioning
C Reverse Communication starts here
C-----
1      CALL DFGMRES(N, COMPUTED_SOLUTION, RHS, RCI_REQUEST, IPAR,
1      DPAR, TMP)
C-----
C If RCI_REQUEST=0, then the solution was found with the required precision
C-----
        IF (RCI_REQUEST.EQ.0) GOTO 3
C-----
C If RCI_REQUEST=1, then compute the vector A*TMP(IPAR(22))
C and put the result in vector TMP(IPAR(23))
C-----
        IF (RCI_REQUEST.EQ.1) THEN
            CALL MKL_DCSRGMV('N',N, A, IA, JA, TMP(IPAR(22)), TMP(IPAR(23)))
            GOTO 1
C-----
C If RCI_REQUEST=anything else, then DFGMRES subroutine failed
C to compute the solution vector: COMPUTED_SOLUTION(N)
C-----
        ELSE
            GOTO 999
        ENDIF

```

```

C-----
C Reverse Communication ends here
C Get the current iteration number and the FGMRES solution (DO NOT FORGET to
C call DFGMRES_GET routine as COMPUTED_SOLUTION is still containing
C the initial guess!)
C-----
3      CALL DFGMRES_GET(N, COMPUTED_SOLUTION, RHS, RCI_REQUEST, IPAR,
      1 DPAR, TMP, ITERCOUNT)
C-----
C Print solution vector: COMPUTED_SOLUTION(N) and
C the number of iterations: ITERCOUNT
C-----

      PRINT *, ''
      PRINT *, ' The system has been SUCCESSFULLY solved'
      PRINT *, ''
      PRINT *, ' The following solution has been obtained:'
      DO I=1,N
          WRITE(*, '(A18,I1,A2,E10.3)') 'COMPUTED_SOLUTION(',I,')=',
1 COMPUTED_SOLUTION(I)
      ENDDO
      PRINT *, ''
      PRINT *, ' The expected solution is:'
      DO I=1,N
          WRITE(*, '(A18,I1,A2,E10.3)') 'EXPECTED_SOLUTION(',I,')=',
1 EXPECTED_SOLUTION(I)
      ENDDO
      PRINT *, ''
      PRINT *, ' Number of iterations: ',ITERCOUNT

```

```
GOTO 1000
```

```
999  PRINT *, 'The solver has returned the ERROR code ', RCI_REQUEST
```

```
1000 CONTINUE
```

```
END
```

C Example of Using RCI (Preconditioned) Flexible Generalized Minimal Residual Solver.

C example results for the same non-symmetric indefinite system as in the previous example. The results are the same up to the notational convention between C and Fortran. Please pay special attention to how it is recommended to handle the differences between the Fortran and C arrays. Specifically, in this example the addresses for input/result are adjusted for user-defined

operations from `IPAR(22)` to `ipar[21]-1`, and from `IPAR(23)` to `ipar[22]-1`, respectively. Upon successful execution of the solver, the following result is printed (up to rounding errors that depend on the computer system used):

```
-----  
The SIMPLEST example of usage of RCI FGMRES solver  
  to solve a non-symmetric indefinite non-degenerate  
  algebraic system of linear equations  
-----  
  Some info about the current run of RCI FGMRES method:  
  
As ipar[7]=1, the automatic test for the maximal number of iterations will be performed  
+++  
As ipar[8]=1, the automatic residual test will be performed  
+++  
As ipar[9]=0, the user-defined stopping test will not be requested, thus,  
RCI_request will not take the value 2  
+++  
As ipar[10]=0, the Preconditioned FGMRES iterations will not  
be performed, thus, RCI_request will not take the value 3  
+++  
As ipar[11]=1, the automatic test for the norm of the next  
generated vector is not equal to zero up to rounding and computational  
errors will be performed, thus, RCI_request will not take the value 4  
+++  
  
The system has been SUCCESSFULLY solved  
  
The following solution has been obtained:  
computed_solution[0]=-1.000000e+000
```

```
computed_solution[1]=1.000000e+000  
computed_solution[2]=3.053113e-016  
computed_solution[3]=1.000000e+000  
computed_solution[4]=-1.000000e+000
```

The expected solution is:

```
expected_solution[0]=-1.000000e+000  
expected_solution[1]=1.000000e+000  
expected_solution[2]=0.000000e+000  
expected_solution[3]=1.000000e+000  
expected_solution[4]=-1.000000e+000
```

Number of iterations: 5

Example C-15b. C Example to Solve Non-Symmetric Indefinite System

```

/*****
/*
/*                               INTEL CONFIDENTIAL
/*
/*   Copyright(C) 2005-2008 Intel Corporation. All Rights Reserved.
/*
/*   The source code contained or described herein and all documents related to
/*   the source code ("Material")are owned by Intel Corporation or its suppliers
/*   or licensors. Title to the Material remains with Intel Corporation or its
/*   suppliers and licensors.The Material contains trade secrets and proprietary
/*   and confidential information of Intel or its suppliers and licensors. The
/*   Material is protected by worldwide copyright and trade secret laws and
/*   treaty provisions. No part of the Material may be used, copied, reproduced,
/*   modified,published, uploaded, posted, transmitted, distributed or disclosed
/*   in any way without Intel's prior express written permission.
/*
/*   No license under any patent, copyright, trade secret or other intellectual
/*   property right is granted to or conferred upon you by disclosure or delivery
/*   of the Materials, either expressly, by implication, inducement, estoppel or
/*   otherwise. Any license under such intellectual property rights must be
/*   express and approved by Intel in writing.
/*
/*
/*****
/*   Content:
/*
/*   Intel MKL RCI (P)FGMRES ((Preconditioned) Flexible Generalized Minimal
/*   RESidual method) example
/*****/

/*-----
/*   Example program for solving non-symmetric indefinite system of equations
/*   Simplest case: no preconditioning and no user-defined stopping tests

```

```

/*-----*/

#include <stdio.h>
#include "mkl_blas.h"
#include "mkl_spblas.h"
#include "mkl_rci.h"
#define N 5
#define size 128

int main(void)
{

    /*-----
    /* Define arrays for the upper triangle of the coefficient matrix
    /* Compressed sparse row storage is used for sparse representation
    /*-----*/
    MKL_INT ia[6]={1,3,6,9,12,14};
    MKL_INT ja[13]={    1,        3,
                      1,    2,        4,
                      2,    3,        5,
                      3,    4,    5,
                      4,        5

    };
    double A[13]={ 1.0,        -1.0,
                  -1.0, 1.0,        -1.0,
                  1.0, -2.0,        1.0,
                  -1.0, 2.0,-1.0,
                  -1.0,-3.0 };

    /*-----
    /* Allocate storage for the ?par parameters and the solution/rhs vectors

```

```

/*-----*/
MKL_INT ipar[size];
double dpar[size], tmp[N*(2*N+1)+(N*(N+9))/2+1];
double expected_solution[N]={-1.0,1.0,0.0,1.0,-1.0};
double rhs[N];
double computed_solution[N];

/*-----
/* Some additional variables to use with the RCI (P)FGMRES solver
/*-----*/
MKL_INT itercount;
MKL_INT RCI_request, i, ivar;
double dvar;
char cvar;

printf("-----\n");
printf("The SIMPLEST example of usage of RCI FGMRES solver\n");
printf("to solve a non-symmetric indefinite non-degenerate\n");
printf("      algebraic system of linear equations\n");
printf("-----\n\n");
/*-----
/* Initialize variables and the right hand side through matrix-vector product
/*-----*/
ivar=N;
cvar='N';
mkl_dcsrgemv(&cvar, &ivar, A, ia, ja, expected_solution, rhs);
/*-----
/* Initialize the initial guess
/*-----

```

```

for(i=0;i<N;i++)
{
    computed_solution[i]=1.0;
}
/*-----
/* Initialize the solver
/*-----
dfgmres_init(&ivar, computed_solution, rhs, &RCI_request, ipar, dpar, tmp);
if (RCI_request!=0) goto FAILED;
/*-----
/* Set the desired parameters:
/* LOGICAL parameters:
/* do residual stopping test
/* do not request for the user defined stopping test
/* do the check of the norm of the next generated vector automatically
/* DOUBLE PRECISION parameters
/* set the relative tolerance to 1.0D-3 instead of default value 1.0D-6
/*-----*/
ipar[8]=1;
ipar[9]=0;
ipar[11]=1;
dpar[0]=1.0E-3;
/*-----
/* Check the correctness and consistency of the newly set parameters
/*-----*/
dfgmres_check(&ivar, computed_solution, rhs, &RCI_request, ipar, dpar, tmp);
if (RCI_request!=0) goto FAILED;
/*-----

```

```
/* Print the info about the RCI FGMRES method
/*-----*/
printf("Some info about the current run of RCI FGMRES method:\n\n");
if (ipar[7])
{
    printf("As ipar[7]=%d, the automatic test for the maximal number of iterations will
be\n", ipar[7]);
    printf("performed\n");
}
else
{
    printf("As ipar[7]=%d, the automatic test for the maximal number of iterations will
be\n", ipar[7]);
    printf("skipped\n");
}
printf("+++\\n");
if (ipar[8])
{
    printf("As ipar[8]=%d, the automatic residual test will be performed\\n", ipar[8]);
}
else
{
    printf("As ipar[8]=%d, the automatic residual test will be skipped\\n", ipar[8]);
}
printf("+++\\n");
if (ipar[9])
{
    printf("As ipar[9]=%d, the user-defined stopping test will be requested via\\n", ipar[9]);
    printf("RCI_request=2\\n");
}
```

```
    }
    else
    {
        printf("As ipar[9]=%d, the user-defined stopping test will not be requested, thus,\n",
            ipar[9]);
        printf("RCI_request will not take the value 2\n");
    }
    printf("+++\\n");
    if (ipar[10])
    {
        printf("As ipar[10]=%d, the Preconditioned FGMRES iterations will be performed, thus,\n",
            ipar[10]);
        printf("the preconditioner action will be requested via RCI_request=3\\n");
    }
    else
    {
        printf("As ipar[10]=%d, the Preconditioned FGMRES iterations will not be performed,\n",
            ipar[10]);
        printf("thus, RCI_request will not take the value 3\\n");
    }
    printf("+++\\n");
    if (ipar[11])
    {
        printf("As ipar[11]=%d, the automatic test for the norm of the next generated vector
            is\\n", ipar[11]);
        printf("not equal to zero up to rounding and computational errors will be performed,\n");
        printf("thus, RCI_request will not take the value 4\\n");
    }
    else
    {
```

```

        printf("As ipar[11]=%d, the automatic test for the norm of the next generated vector
is\n", ipar[11]);

        printf("not equal to zero up to rounding and computational errors will be skipped,\n");
        printf("thus, the user-defined test will be requested via RCI_request=4\n");
    }
    printf("+++\n\n");

    /*-----
    /* Compute the solution by RCI (P)FGMRES solver without preconditioning
    /* Reverse Communication starts here
    /*-----*/
ONE:  dfgmres(&ivar, computed_solution, rhs, &RCI_request, ipar, dpar, tmp);

    /*-----
    /* If RCI_request=0, then the solution was found with the required precision
    /*-----*/
    if (RCI_request==0) goto COMPLETE;

    /*-----
    /* If RCI_request=1, then compute the vector A*tmp[ipar[21]-1]
    /* and put the result in vector tmp[ipar[22]-1]
    /*-----
    /* NOTE that ipar[21] and ipar[22] contain FORTRAN style addresses,
    /* therefore, in C code it is required to subtract 1 from them to get
    /* C style addresses

    /*-----*/
    if (RCI_request==1)
    {
        mkl_dcsrgemv(&cvar, &ivar, A, ia, ja, &tmp[ipar[21]-1], &tmp[ipar[22]-1]);
        goto ONE;
    }

```

```

/*-----
/* If RCI_request=anything else, then dfgmres subroutine failed
/* to compute the solution vector: computed_solution[N]
/*-----*/
else
{
    goto FAILED;
}
/*-----
/* Reverse Communication ends here
/* Get the current iteration number and the FGMRES solution (DO NOT FORGET
/* to call dfgmres_get routine as computed_solution is still containing
/* the initial guess!)
/*-----*/
COMPLETE:  dfgmres_get(&ivar, computed_solution, rhs, &RCI_request, ipar, dpar, tmp, &itercount);
/*
/*-----
/* Print solution vector: computed_solution[N] and the number of iterations: itercount
/*-----*/
printf(" The system has been SUCCESSFULLY solved \n");
printf("\n The following solution has been obtained: \n");
for (i=0;i<N;i++)
{
    printf("computed_solution[%d]=",i);
    printf("%e\n", computed_solution[i]);
}
printf("\n The expected solution[%d]=",i);
for (i=0;i<N;i++)

```



```
{
    printf("expected_solution[%d]=" ,i);
    printf("%e\n", expected_solution[i]);
}
printf("\n Number of iterations: %d",itercount);
goto SUCCEEDED;
FAILED: printf("The solver has returned the ERROR code %d", RCI_request);

SUCCEEDED: return 0;
}
```

Fourier Transform Functions Code Examples

This section presents code examples of functions described in the “[FFT Functions](#)” and “[Cluster FFT Functions](#)” sections in the “Fourier Transform Functions” chapter. The examples are grouped in subsections

- [Examples for FFT Functions](#), including [Examples of Using Multi-Threading for FFT Computation](#)
- [Examples for Cluster FFT Functions](#)
- [Auxiliary data transformations](#).

FFT Code Examples

This section presents code examples of using the FFT interface functions described in “[Fourier Transform Functions](#)” chapter. Here are the examples of two one-dimensional computations. These examples use the default settings for all of the configuration parameters, which are specified in “[Configuration Settings](#)”.

Example C-16 One-dimensional In-place FFT (Fortran Interface)

```
! Fortran example.

! 1D complex to complex, and real to conjugate even
Use MKL_DFTI
Complex :: X(32)
Real :: Y(34)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc1_Handle, My_Desc2_Handle
Integer :: Status

!...put input data into X(1),...,X(32); Y(1),...,Y(32)


! Perform a complex to complex transform
Status = DftiCreateDescriptor( My_Desc1_Handle, DFTI_SINGLE,&
    DFTI_COMPLEX, 1, 32 )
Status = DftiCommitDescriptor( My_Desc1_Handle )
Status = DftiComputeForward( My_Desc1_Handle, X )
Status = DftiFreeDescriptor(My_Desc1_Handle)
! result is given by {X(1),X(2),...,X(32)}


! Perform a real to complex conjugate even transform
Status = DftiCreateDescriptor(My_Desc2_Handle, DFTI_SINGLE,&
    DFTI_REAL, 1, 32)
Status = DftiCommitDescriptor(My_Desc2_Handle)
Status = DftiComputeForward(My_Desc2_Handle, Y)
Status = DftiFreeDescriptor(My_Desc2_Handle)
! result is given in CCS format.
```

Example C-16a One-dimensional Out-of-place FFT (Fortran Interface)

```
! Fortran example.

! 1D complex to complex, and real to conjugate even

Use MKL_DFTI

Complex :: X_in(32)
Complex :: X_out(32)
Real :: Y_in(32)
Real :: Y_out(34)

type(DFTI_DESCRIPTOR), POINTER :: My_Desc1_Handle, My_Desc2_Handle
Integer :: Status

...put input data into X_in(1),...,X_in(32); Y_in(1),...,Y_in(32)

! Perform a complex to complex transform

Status = DftiCreateDescriptor( My_Desc1_Handle, DFTI_SINGLE,
DFTI_COMPLEX, 1, 32 )

Status = DftiSetValue( My_Desc1_Handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE)
Status = DftiCommitDescriptor( My_Desc1_Handle )
Status = DftiComputeForward( My_Desc1_Handle, X_in, X_out )
Status = DftiFreeDescriptor(My_Desc1_Handle)

! result is given by {X_out(1),X_out(2),...,X_out(32)}

! Perform a real to complex conjugate even transform

Status = DftiCreateDescriptor(My_Desc2_Handle, DFTI_SINGLE,
DFTI_REAL, 1, 32)

Status = DftiSetValue( My_Desc2_Handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE)
Status = DftiCommitDescriptor(My_Desc2_Handle)
Status = DftiComputeForward(My_Desc2_Handle, Y_in, Y_out)
Status = DftiFreeDescriptor(My_Desc2_Handle)

! result is given by Y_out in CCS format.
```

Example C-17 One-dimensional In-place FFT (C Interface)

```

/* C example, float _Complex is defined in C9X */

#include "mkl_dfti.h"

float _Complex x[32];
float y[32];

DFTI_DESCRIPTOR *my_desc1_handle,*my_desc2_handle;

/* .... or alternatively
dfti_descriptor_handle my_desc1_handle, my_desc2_handle; */

long status;

//...put input data into x[0],...,x[31]; y[0],...,y[31]
status = DftiCreateDescriptor( &my_desc1_handle, DFTI_SINGLE,
                             DFTI_COMPLEX, 1, 32);

status = DftiCommitDescriptor( my_desc1_handle );
status = DftiComputeForward( my_desc1_handle, x);
status = DftiFreeDescriptor(&my_desc1_handle);

/* result is x[0], ..., x[31]*/

status = DftiCreateDescriptor( &my_desc2_handle, DFTI_SINGLE,
                             DFTI_REAL, 1, 32);

status = DftiCommitDescriptor( my_desc2_handle);
status = DftiComputeForward( my_desc2_handle, y);
status = DftiFreeDescriptor(&my_desc2_handle);

/* result is given in CCS format*/

```

Example C-17a One-dimensional Out-of-place FFT (C Interface)

```
/* C example, float _Complex is defined in C9X */
#include "mkl_dfti.h"
float _Complex x_in[32];
float _Complex x_out[32];
float y_in[32];
float y_out[34];
DFTI_DESCRIPTOR *my_desc1_handle,*my_desc2_handle;
/* .... or alternatively
dfti_descriptor_handle my_desc1_handle, my_desc2_handle; */
long status;
//...put input data into x_in[0],...,x_in[31]; y_in[0],...,y_in[31]
status = DftiCreateDescriptor( &my_desc1_handle, DFTI_SINGLE,
                             DFTI_COMPLEX, 1, 32);
status = DftiSetValue( my_desc1_handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE);
status = DftiCommitDescriptor( my_desc1_handle );
status = DftiComputeForward( my_desc1_handle, x_in, x_out);
status = DftiFreeDescriptor(&my_desc1_handle);
/* result is x_out[0], ..., x_out[31]*/
status = DftiCreateDescriptor( &my_desc2_handle, DFTI_SINGLE,
                             DFTI_REAL, 1, 32);
Status = DftiSetValue( My_Desc2_Handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE);
status = DftiCommitDescriptor( my_desc2_handle);
status = DftiComputeForward( my_desc2_handle, y_in, y_out);
status = DftiFreeDescriptor(&my_desc2_handle);
/* result is given by y_out in CCS format*
```

Example C-18 Two-dimensional FFT (Fortran Interface)

The following is an example of two simple two-dimensional transforms. Notice that the data and result parameters in computation functions are all declared as assumed-size rank-1 array `DIMENSION(0:*)`. Therefore two-dimensional array must be transformed to one-dimensional array by `EQUIVALENCE` statement or other facilities of Fortran.

! Fortran example.

! 2D complex to complex, and real to conjugate even

Use MKL_DFTI

Complex :: X_2D(32,100)

Real :: Y_2D(34, 102)

Complex :: X(3200)

Real :: Y(3468)

Equivalence (X_2D, X)

Equivalence (Y_2D, Y)

type(DFTI_DESCRIPTOR), POINTER :: My_Desc1_Handle, My_Desc2_Handle

Integer :: Status, L(2)

!...put input data into X_2D(j,k), Y_2D(j,k), 1<=j=32,1<=k<=100

!...set L(1) = 32, L(2) = 100

!...the transform is a 32-by-100

! Perform a complex to complex transform

```
Status = DftiCreateDescriptor( My_Desc1_Handle, DFTI_SINGLE,&
                             DFTI_COMPLEX, 2, L)
```

```
Status = DftiCommitDescriptor( My_Desc1_Handle)
```

```
Status = DftiComputeForward( My_Desc1_Handle, X)
```

```
Status = DftiFreeDescriptor(My_Desc1_Handle)
```

! result is given by X_2D(j,k), 1<=j<=32, 1<=k<=100

! Perform a real to complex conjugate even transform

```
Status = DftiCreateDescriptor( My_Desc2_Handle, DFTI_SINGLE,&
                               DFTI_REAL, 2, L)
Status = DftiCommitDescriptor( My_Desc2_Handle)
Status = DftiComputeForward( My_Desc2_Handle, Y)
Status = DftiFreeDescriptor(My_Desc2_Handle)
! result is given by the complex value z(j,k) 1<=j<=32; 1<=k<=100
! and is stored in CCS format
```

Example C-19 Two-dimensional FFT (C Interface)

```
/* C99 example */

#include "mkl_dfti.h"

float _Complex x[32][100];
float y[34][102];

DFTI_DESCRIPTOR *my_desc1_handle, my_desc2_handle;
long status, l[2];

//...put input data into x[j][k] 0<=j<=31, 0<=k<=99
//...put input data into y[j][k] 0<=j<=31, 0<=k<=99
l[0] = 32; l[1] = 100;

status = DftiCreateDescriptor( &my_desc1_handle, DFTI_SINGLE,
                             DFTI_COMPLEX, 2, l);
status = DftiCommitDescriptor( my_desc1_handle);
status = DftiComputeForward( my_desc1_handle, x);
status = DftiFreeDescriptor(&my_desc1_handle);
/* result is the complex value x[j][k], 0<=j<=31, 0<=k<=99 */
status = DftiCreateDescriptor( &my_desc2_handle, DFTI_SINGLE,
                             DFTI_REAL, 2, l);
status = DftiCommitDescriptor( my_desc2_handle);
status = DftiComputeForward( my_desc2_handle, y);
status = DftiFreeDescriptor(&my_desc2_handle);
/* result is the complex value z(j,k) 0<=j<=31; 0<=k<=99
/* and is stored in CCS format*/
```

The following examples demonstrate how you can change the default configuration settings by using the `DftiSetValue` function.

For instance, to preserve the input data after the FFT computation, the configuration of the `DFTI_PLACEMENT` should be changed to "not in place" from the default choice of "in place."

Example C-20 Changing Default Settings (Fortran)

The code below illustrates how this can be done:

```
! Fortran example
! 1D complex to complex, not in place
Use MKL_DFTI
Complex :: X_in(32), X_out(32)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc_Handle
Integer :: Status
!...put input data into X_in(j), 1<=j<=32
Status = DftiCreateDescriptor( My_Desc_Handle,&
DFTI_SINGLE, DFTI_COMPLEX, 1, 32)
Status = DftiSetValue( My_Desc_Handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE)
Status = DftiCommitDescriptor( My_Desc_Handle)
Status = DftiComputeForward( My_Desc_Handle, X_in, X_out)
Status = DftiFreeDescriptor (My_Desc_Handle)
! result is X_out(1),X_out(2),...,X_out(32)
```

Example C-21 Changing Default Settings (C)

```
/* C99 example */

#include "mkl_dfti.h"

float _Complex x_in[32], x_out[32];
DFTI_DESCRIPTOR_HANDLE my_desc_handle;

/* or alternatively
DFTI_DESCRIPTOR *my_desc_handle;*/

long status;

//...put input data into x_in[j], 0 <= j < 32

status = DftiCreateDescriptor( &my_desc_handle, DFTI_SINGLE,
    DFTI_COMPLEX, 1, 32);

status = DftiSetValue( my_desc_handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE);
status = DftiCommitDescriptor( my_desc_handle);
status = DftiComputeForward( my_desc_handle, x_in, x_out);
status = DftiFreeDescriptor(&my_desc_handle);

/* result is x_out[0], x_out[1], ..., x_out[31] */
```

Example C-22 Using Status Checking Function

The [Example C-22](#) illustrates the use of the status checking functions described in [Chapter 11](#).

```
/* C */

DFTI_DESCRIPTOR_HANDLE desc;
long status, class_error, value;
char* error_message;
// . . . descriptor creation and other code
status = DftiGetValue( desc, DFTI_PRECISION, &value); //

class_error = DftiErrorClass(status, DFTI_NO_ERROR);
if (! class_error) {
    printf ("DftiGetValue() fixes the wrong situation and
           returns the corresponding value n");
    error_message = DftiErrorMessage(status);
    printf("error_message = %s \n", error_message);
}
// . . .

! Fortran
type(DFTI_DESCRIPTOR), POINTER :: desc
integer value, status
character(DFTI_MAX_MESSAGE_LENGTH) error_message
logical class_error
! . . . descriptor creation and other code
status = DftiGetValue( desc, DFTI_PRECISION, value)
class_error = DftiErrorClass(status, DFTI_NO_ERROR)
if (.not. class_error) then
    print *, ' DftiGetValue() fixes the wrong situation and ',&
```

```
        'returns the corresponding value '  
    error_message = DftiErrorMessage(status)  
    print *, 'error_message = ', error_message  
endif
```

Example C-23. Computing 2D FFT by One-Dimensional Transforms

Below is an example where a 20-by-40 two-dimensional FFT is computed explicitly using one-dimensional transforms. Notice that the data and result parameters in computation functions are all declared as assumed-size rank-1 array `DIMENSION(0:*)`. Therefore two-dimensional array must be transformed to one-dimensional array by `EQUIVALENCE` statement or other facilities of Fortran.

```
! Fortran

use mkl_dfti

Complex :: X_2D(20,40)
Complex :: X(800)

Equivalence (X_2D, X)

INTEGER :: STRIDE(2)

type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle_Dim1
type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle_Dim2

! ...

Status = DftiCreateDescriptor(Desc_Handle_Dim1, DFTI_SINGLE,&
                             DFTI_COMPLEX, 1, 20 )

Status = DftiCreateDescriptor(Desc_Handle_Dim2, DFTI_SINGLE,&
                             DFTI_COMPLEX, 1, 40 )

! perform 40 one-dimensional transforms along 1st dimension
Status = DftiSetValue( Desc_Handle_Dim1, DFTI_NUMBER_OF_TRANSFORMS, 40 )
Status = DftiSetValue( Desc_Handle_Dim1, DFTI_INPUT_DISTANCE, 20 )
Status = DftiSetValue( Desc_Handle_Dim1, DFTI_OUTPUT_DISTANCE, 20 )
Status = DftiCommitDescriptor( Desc_Handle_Dim1 )
Status = DftiComputeForward( Desc_Handle_Dim1, X )

! perform 20 one-dimensional transforms along 2nd dimension
Stride(1) = 0; Stride(2) = 20

Status = DftiSetValue( Desc_Handle_Dim2, DFTI_NUMBER_OF_TRANSFORMS, 20 )
Status = DftiSetValue( Desc_Handle_Dim2, DFTI_INPUT_DISTANCE, 1 )
```

```

Status = DftiSetValue( Desc_Handle_Dim2, DFTI_OUTPUT_DISTANCE, 1 )
Status = DftiSetValue( Desc_Handle_Dim2, DFTI_INPUT_STRIDES, Stride )
Status = DftiSetValue( Desc_Handle_Dim2, DFTI_OUTPUT_STRIDES, Stride )
Status = DftiCommitDescriptor( Desc_Handle_Dim2 )
Status = DftiComputeForward( Desc_Handle_Dim2, X )
Status = DftiFreeDescriptor( Desc_Handle_Dim1 )
Status = DftiFreeDescriptor( Desc_Handle_Dim2 )

/* C */
#include "mkl_dfti.h"
float _Complex x[20][40];
long stride[2];
long status;
DFTI_DESCRIPTOR_HANDLE desc_handle_dim1;
DFTI_DESCRIPTOR_HANDLE desc_handle_dim2;
//...

status = DftiCreateDescriptor( &desc_handle_dim1, DFTI_SINGLE,
    DFTI_COMPLEX, 1, 20 );
status = DftiCreateDescriptor( &desc_handle_dim2, DFTI_SINGLE,
    DFTI_COMPLEX, 1, 40 );

/* perform 40 one-dimensional transforms along 1st dimension */
/* note that the 1st dimension data are not unit-stride */
stride[0] = 0; stride[1] = 40;

status = DftiSetValue( desc_handle_dim1, DFTI_NUMBER_OF_TRANSFORMS, 40 );
status = DftiSetValue( desc_handle_dim1, DFTI_INPUT_DISTANCE, 1 );
status = DftiSetValue( desc_handle_dim1, DFTI_OUTPUT_DISTANCE, 1 );
status = DftiSetValue( desc_handle_dim1, DFTI_INPUT_STRIDES, stride );

```

```
status = DftiSetValue( desc_handle_dim1, DFTI_OUTPUT_STRIDES, stride );
status = DftiCommitDescriptor( desc_handle_dim1 );
status = DftiComputeForward( desc_handle_dim1, x );
/* perform 20 one-dimensional transforms along 2nd dimension */
/* note that the 2nd dimension is unit stride */
status = DftiSetValue( desc_handle_dim2, DFTI_NUMBER_OF_TRANSFORMS, 20 );
status = DftiSetValue( desc_handle_dim2,
DFTI_INPUT_DISTANCE, 40 );
status = DftiSetValue( desc_handle_dim2,
DFTI_OUTPUT_DISTANCE, 40 );
status = DftiCommitDescriptor( desc_handle_dim2 );
status = DftiComputeForward( desc_handle_dim2, x );
status = DftiFreeDescriptor( &Desc_Handle_dim1 );
status = DftiFreeDescriptor( &Desc_Handle_dim2 );
```

The following are examples of real multi-dimensional transforms with CCE format storage of conjugate-even complex matrix. [Example C-24](#) is two-dimensional in-place transform and [Example C-24a](#) is two-dimensional out-of-place transform in Fortran interface. [Example C-25](#) is three-dimensional out-of-place transform in C interface. Note that the data and result parameters in computation functions are all declared as assumed-size rank-1 array `DIMENSION(0:*)`. Therefore two-dimensional array must be transformed to one-dimensional array by `EQUIVALENCE` statement or other facilities of Fortran.

Example C-24. Two-Dimensional REAL In-place FFT (Fortran Interface)

```
! Fortran example.
! 2D and real to conjugate even
Use MKL_DFTI
Real :: X_2D(34,100) ! 34 = (32/2 + 1)*2
Real :: X(3400)
Equivalence (X_2D, X)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc_Handle
Integer :: Status, L(2)
Integer :: strides_in(3)
Integer :: strides_out(3)
! ...put input data into X_2D(j,k), 1<=j=32,1<=k<=100
! ...set L(1) = 32, L(2) = 100
! ...set strides_in(1) = 0, strides_in(2) = 1, strides_in(3) = 34
! ...set strides_out(1) = 0, strides_out(2) = 1, strides_out(3) = 17
! ...the transform is a 32-by-100
! Perform a real to complex conjugate even transform
Status = DftiCreateDescriptor( My_Desc_Handle, DFTI_SINGLE,&
DFTI_REAL, 2, L )
Status = DftiSetValue(My_Desc_Handle, DFTI_CONJUGATE_EVEN_STORAGE,&
DFTI_COMPLEX_COMPLEX)
Status = DftiSetValue(My_Desc_Handle, DFTI_INPUT_STRIDES, strides_in)
Status = DftiSetValue(My_Desc_Handle, DFTI_OUTPUT_STRIDES, strides_out)
Status = DftiCommitDescriptor( My_Desc_Handle)
Status = DftiComputeForward( My_Desc_Handle, X )
Status = DftiFreeDescriptor(My_Desc_Handle)
! result is given by the complex value z(j,k) 1<=j<=17; 1<=k<=100 and
! is stored in real matrix X_2D in CCE format.
```


Example C-24a. Two-Dimensional REAL Out-of-place FFT (Fortran Interface)

```

! Fortran example.
! 2D and real to conjugate even
Use MKL_DFTI
Real :: X_2D(32,100)
Complex :: Y_2D(17, 100) ! 17 = 32/2 + 1
Real :: X(3200)
Complex :: Y(1700)
Equivalence (X_2D, X)
Equivalence (Y_2D, Y)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc_Handle
Integer :: Status, L(2)
Integer :: strides_out(3)

! ...put input data into X_2D(j,k), 1<=j=32,1<=k<=100
! ...set L(1) = 32, L(2) = 100
! ...set strides_out(1) = 0, strides_out(2) = 1, strides_out(3) = 17

! ...the transform is a 32-by-100
! Perform a real to complex conjugate even
transform

Status = DftiCreateDescriptor( My_Desc_Handle, DFTI_SINGLE,&
DFTI_REAL, 2, L )
Status = DftiSetValue(My_Desc_Handle,&
DFTI_CONJUGATE_EVEN_STORAGE, DFTI_COMPLEX_COMPLEX)
Status = DftiSetValue( My_Desc_Handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE )
Status = DftiSetValue(My_Desc_Handle,&
DFTI_OUTPUT_STRIDES, strides_out)

```

```
Status = DftiCommitDescriptor(My_Desc_Handle)
Status = DftiComputeForward(My_Desc_Handle, X, Y)
Status = DftiFreeDescriptor(My_Desc_Handle)
! result is given by the complex value z(j,k) 1<=j<=17; 1<=k<=100 and
! is stored in complex matrix Y_2D in CCE format.
```

Example C-25 Three-Dimensional REAL FFT (C Interface)

```
/* C99 example */
#include "mkl_dfti.h"
float x[32][100][19];
float _Complex y[32][100][10]; /* 10 = 19/2 + 1 */
DFTI_DESCRIPTOR_HANDLE my_desc_handle;
/* or alternatively
DFTI_DESCRIPTOR *my_desc_handle*/
long status, l[3];
long strides_out[4];

//...put input data into x[j][k][s] 0<=j<=31, 0<=k<=99, 0<=s<=18
l[0] = 32; l[1] = 100; l[2] = 19;

strides_out[0] = 0; strides_out[1] = 1000;
strides_out[2] = 10; strides_out[3] = 1;

status = DftiCreateDescriptor( &my_desc_handle, DFTI_SINGLE,
DFTI_REAL, 3, l );
status = DftiSetValue(my_desc_handle,
DFTI_CONJUGATE_EVEN_STORAGE, DFTI_COMPLEX_COMPLEX);
status = DftiSetValue( my_desc_handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE );
status = DftiSetValue(my_desc_handle,
DFTI_OUTPUT_STRIDES, strides_out);

status = DftiCommitDescriptor(my_desc_handle);
status = DftiComputeForward(my_desc_handle, x, y);
```

```
status = DftiFreeDescriptor(&my_desc_handle);  
/* result is the complex value z(j,k,s) 0<=j<=31; 0<=k<=99, 0<=s<=9  
and is stored in complex matrix y in CCE format. */
```

Examples of Using Multi-Threading for FFT Computation

The following sample program shows how to employ internal threading in Intel MKL for FFT computation (see case "a" in ["Number of user threads"](#)).

To specify the number of threads inside Intel MKL, use the following settings:

set MKL_NUM_THREADS = 1 for one-threaded mode;

set MKL_NUM_THREADS = 4 for multi-threaded mode.

Note that the configuration parameter DFTI_NUMBER_OF_USER_THREADS must be equal to its default value 1.

Example C-26 Using Intel MKL Internal Threading Mode

```
#include "mkl_dfti.h"

void main () {

float x[200][100];
DFTI_DESCRIPTOR_HANDLE my_desc1_handle;
long status, len[2];
//...put input data into x[j][k] 0<=j<=199, 0<=k<=99

len[0] = 200; len[1] = 100;
status = DftiCreateDescriptor( &my_desc1_handle, DFTI_SINGLE,DFTI_REAL, 2,
    len);
status = DftiCommitDescriptor(
my_desc1_handle);
status = DftiComputeForward(
my_desc1_handle, x);
status = DftiFreeDescriptor(&my_desc1_handle);
}
```

The following [Example C-27](#) and [Example C-27a](#) illustrate a parallel customer program with each descriptor instance used only in a single thread (see cases "b" and "c" in [Number of user threads](#)).

Specify the number of threads for [Example C-27](#) like this:

set `MKL_SERIAL = yes` (or `YES`) for Intel MKL to work in the single-threaded mode (recommended);

set `OMP_NUM_THREADS = 4` for the customer program to work in the multi-threaded mode.

The configuration parameter `DFTI_NUMBER_OF_USER_THREADS` must have its default value of 1.

Example C-27 Using Parallel Mode with Multiple Descriptors Initialized in a Parallel Region

Note that in this example, the program can be transformed to become single-threaded at the customer level but using parallel mode within Intel MKL (case "a"). To achieve this, you need to set the parameter `DFTI_NUMBER_OF_TRANSFORMS = 4` and to set the corresponding parameter `DFTI_INPUT_DISTANCE = 5000`.

```
#include "mkl_dfti.h"
#include "omp.h"

void main ()
{
    float _Complex x[200][100];
    long len[2];
    //...put input data into x[j][k] 0<=j<=199, 0<=k<=99
    len[0] = 50; len[1] = 100;

    // each thread calculates real FFT for matrix (50*100)
    #pragma omp parallel
    {
        DFTI_DESCRIPTOR_HANDLE my_desc_handle;
        long myStatus;
        int myID = omp_get_thread_num ();

myStatus    =  DftiCreateDescriptor (&my_desc_handle, DFTI_SINGLE,
DFTI_COMPLEX, 2, len);

        myStatus = DftiCommitDescriptor (my_desc_handle);
        myStatus = DftiComputeForward (my_desc_handle, &x[myID * len[0] * len[1]]);
        myStatus = DftiFreeDescriptor (&my_desc_handle);
    } /* End OpenMP parallel region */
}
```

```
}
```

Specify the number of threads for [Example C-27a](#) like this:

set `MKL_NUM_THREADS = 1` for Intel MKL to work in the single-threaded mode (obligatory);

set `OMP_NUM_THREADS = 4` for the customer program to work in the multi-threaded mode.

The configuration parameter `DFTI_NUMBER_OF_USER_THREADS` must have the default value of 1.

Example C-27a Using Parallel Mode with Multiple Descriptors Initialized in One Thread

```
#include "mkl_dfti.h"

#include "omp.h"

void main ()
{
    float _Complex x[200][100];
    long len[2];
    long i;
    //...put input data into x[j][k] 0<=j<=199, 0<=k<=99
    len[0] = 50; len[1] = 100;
    DFTI_DESCRIPTOR_HANDLE my_desc_handle[4];
    long myStatus;

    for (i=0;i<3;i++) myStatus = DftiCreateDescriptor (&my_desc_handle[i], DFTI_SINGLE, DFTI_COMPLEX,
        2, len);

    // each thread calculates real FFT for matrix (50*100)
    #pragma omp parallel
    {
        int myID = omp_get_thread_num ();
        myStatus = DftiCommitDescriptor (my_desc_handle[myID]);
        myStatus = DftiComputeForward (my_desc_handle[myID], &x[myID * len[0] * len[1]]);
    } /* End OpenMP parallel region */
    for (i=0;i<3;i++) myStatus = DftiFreeDescriptor (&my_desc_handle[i]);
}
```

The following [Example C-28](#) illustrates a parallel customer program with a common descriptor used in several threads (see case "d" in ["Number of user threads"](#)).

In this case, the number of threads, as well as any other configuration parameter, must not be changed after FFT initialization by the `DftiCommitDescriptor()` function is done.

Example C-28 Using Parallel Mode with a Common Descriptor

```
// set number of threads inside Intel MKL:
//rem set MKL_SERIAL = YES    -    is not required
// since one-threaded mode for Intel MKL is forced automatically
// set OMP_NUM_THREADS = 4    -    multi-threaded mode for customer

#include "mkl_dfti.h"
#include "omp.h"
void main ()
{
    float _Complex x[200][100];
    long status;
    DFTI_DESCRIPTOR_HANDLE desc_handle;
    int nThread = omp_get_max_threads ();
    long len[2];
    //...put input data into x[j][k] 0<=j<=199, 0<=k<=99
    len[0] = 50; len[1] = 100;

    status =
DftiCreateDescriptor (&desc_handle, DFTI_SINGLE, DFTI_COMPLEX, 2, len);
    status = DftiSetValue (desc_handle,
DFTI_NUMBER_OF_USER_THREADS, nThread);
    status = DftiCommitDescriptor (desc_handle);

    // each thread calculates real FFT for matrix (50*100)
    #pragma omp parallel num_threads(nThread)
    {
```

```
long myStatus;

int myID = omp_get_thread_num ();

myStatus = DftiComputeForward (desc_handle, &x[myID * len[0] * len[1]]);
} /* End OpenMP parallel region */

status = DftiFreeDescriptor (&desc_handle);
}
```

Examples for Cluster FFT Functions

The following C example computes a 2-dimensional out-of-place FFT using the cluster FFT interface:

Example 29. 2D Out-of-place Cluster FFT Computation

```

DFTI_DESCRIPTOR_DM_HANDLE desc;

long len[2],v,i,j,n,s;

Complex *in,*out;

MPI_Init(...);

// Create descriptor for 2D FFT
len[0]=nx;
len[1]=ny;
DftiCreateDescriptorDM(MPI_COMM_WORLD,&desc,DFTI_DOUBLE,DFTI_COMPLEX,2,1 en);
// Ask necessary length of in and out arrays and allocate memory
DftiGetValueDM(desc,CDFT_LOCAL_SIZE,&v);
in=(Complex*)malloc(v*sizeof(Complex));
out=(Complex*)malloc(v*sizeof(Complex));
// Fill local array with initial data. Current process performs n rows,
// 0 row of in corresponds to s row of virtual global array
DftiGetValueDM(desc,CDFT_LOCAL_NX,&n);
DftiGetValueDM(desc,CDFT_LOCAL_X_START,&s);
// Virtual global array globalIN is defined by function f as
// globalIN[i*ny+j]=f(i,j)
for(i=0;i<n;i++)
    for(j=0;j<ny;j++) in[i*ny+j]=f(i+s,j);
// Set that we want out-of-place transform (default is DFTI_INPLACE)
DftiSetValueDM(desc,DFTI_PLACEMENT,DFTI_NOT_INPLACE);
// Commit descriptor, calculate FFT, free descriptor
DftiCommitDescriptorDM(desc);
DftiComputeForwardDM(desc,in,out);

```

```
// Virtual global array globalOUT is defined by function g as
// globalOUT[i*ny+j]=g(i,j)
// Now out contains result of FFT. out[i*ny+j]=g(i+s,j)
DftiFreeDescriptorDM(&desc);
free(in);
free(out);
MPI_Finalize();
```

Example 30. 1D In-place Cluster FFT Computations

The C example below illustrates one-dimensional in-place cluster FFT computations effected with a user-defined workspace:

```
DFTI_DESCRIPTOR_DM_HANDLE desc;

long len,v,i,n_out,s_out;

Complex *in,*work;

MPI_Init(...);

// Create descriptor for 1D FFT
DftiCreateDescriptorDM(MPI_COMM_WORLD,&desc,DFTI_DOUBLE,DFTI_COMPLEX,1,1,en);

// Ask necessary length of array and workspace and allocate memory
DftiGetValueDM(desc,CDFT_LOCAL_SIZE,&v);
in=(Complex*)malloc(v*sizeof(Complex));
work=(Complex*)malloc(v*sizeof(Complex));

// Fill local array with initial data. Local array has n elements,
// 0 element of in corresponds to s element of virtual global array
DftiGetValueDM(desc,CDFT_LOCAL_NX,&n);
DftiGetValueDM(desc,CDFT_LOCAL_X_START,&s);

// Set work array as a workspace
DftiSetValueDM(desc,CDFT_WORKSPACE,work);

// Virtual global array globalIN is defined by function f as globalIN[i]=f(i)
for(i=0;i<n;i++) in[i]=f(i+s);

// Commit descriptor, calculate FFT, free descriptor
DftiCommitDescriptorDM(desc);

DftiComputeForwardDM(desc,in);

DftiGetValueDM(desc,CDFT_LOCAL_OUT_NX,&n_out);
DftiGetValueDM(desc,CDFT_LOCAL_OUT_X_START,&s_out);

// Virtual global array globalOUT is defined by function g as globalOUT[i]=g(i)
// Now in contains result of FFT. Local array has n_out elements,
```

```
// 0 element of in corresponds to s_out element of virtual global array.
// in[i]==g(i+s_out)
DftiFreeDescriptorDM(&desc);
free(in);
free(work);
MPI_Finalize();
```

Auxiliary Data Transformations

This section presents code examples for conversion from the Cartesian to polar representation of complex data and vice versa.

Example C-30a Conversion from Cartesian to polar representation of complex data

```
// Cartesian->polar conversion of complex data
// Cartesian representation: z = re + I*im
// Polar representation: z = r * exp( I*phi )
#include <mkl_vml.h>

void

variant1_Cartesian2Polar(int n,const double *re,const double *im,
                        double *r,double *phi)
{
    vdHypot(n,re,im,r);          // compute radii r[]
    vdAtan2(n,re,im,phi);        // compute phases phi[]
}

void
```

```
variant2_Cartesian2Polar(int n,const MKL_Complex16 *z,double *r,double *phi,
                        double *temp_re,double *temp_im)
{
    vzAbs(n,z,r);           // compute radii r[]
    vdPackI(n, (double*)z + 0, 2, temp_re);
    vdPackI(n, (double*)z + 1, 2, temp_im);
    vdAtan2(n,temp_re,temp_im,phi); // compute phases phi[]
}
```

Example C-30b Conversion from polar to Cartesian representation of complex data

```
// Polar->Cartesian conversion of complex data.
// Polar representation:  $z = r * \exp(I * \phi)$ 
// Cartesian representation:  $z = \text{re} + I * \text{im}$ 
#include <mkl_vml.h>

void
variant1_Polar2Cartesian(int n, const double *r, const double *phi,
                        double *re, double *im)
{
    vdSinCos(n, phi, im, re);    // compute direction, i.e.  $z[]/abs(z[])$ 
    vdMul(n, r, re, re);        // scale real part
    vdMul(n, r, im, im);        // scale imaginary part
}

void
variant2_Polar2Cartesian(int n, const double *r, const double *phi,
                        MKL_Complex16 *z,
                        double *temp_re, double *temp_im)
{
    vdSinCos(n, phi, temp_im, temp_re); // compute direction, i.e.  $z[]/abs(z[])$ 
    vdMul(n, r, temp_im, temp_im); // scale imaginary part
    vdMul(n, r, temp_re, temp_re); // scale real part
    vdUnpackI(n, temp_re, (double*)z + 0, 2); // fill in result.re
    vdUnpackI(n, temp_im, (double*)z + 1, 2); // fill in result.im
}
```


Interval Linear Solvers Code Examples

This section presents code examples of using the routines described in “[Interval Linear Solvers](#)”. These routines are intended for computing enclosures and estimates of the solution sets to interval linear systems of equations as well as for checking properties of interval matrices and their inversion.

Example C-31. Interval Gauss-Seidel method

Given an interval system of linear algebraic equations, interval Gauss-Seidel method (implemented as `?gegss` routine) is often applied for enclosing a desired portion of the solution set that is bounded by a prescribed interval box.

Consider the following interval linear system of equations

$$\begin{pmatrix} [2, 3] & [0, 1] \\ [1, 2] & [2, 3] \end{pmatrix} x = \begin{pmatrix} [0, 120] \\ [60, 240] \end{pmatrix}$$

proposed first by E. Hansen (see [Hansen92](#)). Does its solution set intersect the interval box

$$\begin{pmatrix} [0, 200] \\ [0, 200] \end{pmatrix} ?$$

The following sample program answers the above question.

```

PROGRAM DIGEGSS_EXAMPLE
!
! Example program enclosing the solution set to a square interval
! linear system by interval Gauss-Seidel iterative method
!
!-----!
USE INTERVAL_ARITHMETIC
IMPLICIT NONE
!-----!
INTEGER, PARAMETER :: DIM = 2
INTEGER :: NRHS, LDA, LDB, NITS,
INFO, I, J
REAL(8) :: EPSILON
TYPE(D_INTERVAL) :: A(DIM,DIM), B(DIM,1),
ENCL(DIM,1)
CHARACTER(1) :: TRANS
!-----!
PRINT 300
!-----!
! !
! Initializing the input data - !
! !
TRANS = 'N'
NRHS = 2
A(1,1) = DINTERVAL(2.,3.); A(1,2) = DINTERVAL(0.,1.);
A(2,1) = DINTERVAL(0.,1.); A(2,2) = DINTERVAL(2.,3.);
LDA = 2

```

```

B(1,1) = DINTERVAL(0.,120.); B(2,1) = DINTERVAL(60.,240.);

LDB = 2

EPSILON = 1.D-6

NITS = 20

!-----!

!

! Assigning the bounding box for the solution set -
DO I = 1, DIM
ENCL(I,1) = DINTERVAL(0.,200.)
END DO

!-----!

CALL DIGEGSS(TRANS,
DIM, NRHS, A, LDA, B, LDB, ENCL, EPSILON, NITS, INFO )

!-----!

!

! Outputting the solution
IF( INFO /= 0 ) THEN
PRINT 400
ELSE
PRINT 600
DO I = 1, DIM
PRINT *, '[', B(I,1), ']'
END DO
END IF

!-----!

300 FORMAT (/, ' **** SOLVING INTERVAL LINEAR SYSTEM **** ', /, &
' by interval Gauss-Seidel method ')
400 FORMAT (/, ' The interval Gauss-Seidel method fails. ')

```

```
600 FORMAT (/, ' Outer interval estimate of the solution set:',/)
!-----!
END PROGRAM DIGEGSS_EXAMPLE
```

Assigning double-precision intervals to the entries of the matrix A and right-hand side vector B is carried out by `DINTERVAL` function that turns two real numbers into the interval having these reals as endpoints. Running the above code produces the answer

```
**** SOLVING INTERVAL LINEAR SYSTEM****
by interval Gauss-Seidel method
Outer interval estimate of the solution set:
[ 0.000000000000000E+000 60.0000000000000 ]
[ 0.000000000000000E+000 120.000000000000 ]
```

One can make sure that the resulting box really encloses the required portion of the solution set after having a look at the corresponding graph from the paper [Hansen92](#). Moreover, it is even the tightest possible enclosure.

Example C-32. Hansen-Bliek-Rohn procedure

The following Fortran 90 program illustrates the use of `digehbs` routine implementing “semiinterval” Hansen-Bliek-Rohn procedure for outer interval estimation of the solution sets to interval linear systems.

```
PROGRAM DIGEHBS_EXAMPLE
!
! Example program for enclosing the solution set to square interval
! interval system of equations by Hansen-Bliek-Rohn procedure
!
!-----!
USE INTERVAL_ARITHMETIC
IMPLICIT NONE
!-----!
INTEGER, PARAMETER :: DIM = 2
INTEGER :: LDA, LDB, INFO, I, J
TYPE(D_INTERVAL), ALLOCATABLE ::
A(:, :), B(:)
CHARACTER(1) :: TRANS
!-----!
PRINT 300
!-----!
!
! Initializing the input data -
!
TRANS = 'N'
ALLOCATE( A(DIM,DIM), B(DIM) )
A(1,1) = DINTERVAL(2.,4.); A(1,2) = DINTERVAL(-2.,1.)
A(2,1) = DINTERVAL(-1.,2.); A(2,2) = DINTERVAL(2.,2.)
LDA = 2
```

```

B(1) = DINTERVAL(0.,2.); B(2) = DINTERVAL(0.,2.)

LDB = 2

!-----!

CALL DIGEHBS( TRANS, DIM, A, LDA,
B, LDB, INFO )

!-----!

IF( INFO /= 0 ) THEN
PRINT 400
ELSE
PRINT 600
DO I = 1, DIM
PRINT *, I, ' ) [', B(I), ']'
END DO
END IF

!-----!

DEALLOCATE( A, B )

!-----!

300 FORMAT (/, ' **** SOLVING INTERVAL LINEAR SYSTEM ****',/, &
' by Hansen-Bliek-Rohn procedure ',/)
400 FORMAT (/, ' The matrix of the system is not an H-matrix',/, &
' Hansen-Bliek-Rohn procedure fails. ',/)
600 FORMAT (/, ' Enclosure of the solution set: ',/)

!-----!

END PROGRAM DIGEHBS_EXAMPLE

```

However, the output of the program looks like

```
**** SOLVING INTERVAL LINEAR SYSTEM****
```

by Hansen-Bliek-Rohn procedure

The matrix of the system is not an H-matrix,

Hansen-Bliek-Rohn procedure fails.

This result is obtained because the program is applied to the interval linear system

$$\begin{pmatrix} [2, 4] & [-2, 1] \\ [-1, 2] & [2, 4] \end{pmatrix} \mathbf{x} = \begin{pmatrix} [0, 2] \\ [0, 2] \end{pmatrix}$$

where the interval matrix is not an H-matrix (that is, it does not have diagonal dominance).

However, preconditioning by `digemip` routine helps to resolve the problem. The following modified program, which incorporates preliminary preconditioning of the interval linear system under solution, makes the matrix diagonally dominant and produces an acceptable answer to the problem.

```
PROGRAM DIGEMIP_DIGEHS_EXAMPLE
!
! Example program for enclosing the solution set to square interval
! interval system of equations by Hansen-Bliek-Rohn procedure
!
!-----!
USE INTERVAL_ARITHMETIC
IMPLICIT NONE
!-----!
INTEGER, PARAMETER :: DIM = 2, NRHS = 1
INTEGER :: LDA, LDB, INFO, I, J
TYPE(D_INTERVAL), ALLOCATABLE ::
A(:, :), B(:)
CHARACTER(1) :: TRANS
!-----!
PRINT 300
!-----!
!
! Initializing the input data -
!
TRANS = 'N'
ALLOCATE( A(DIM,DIM), B(DIM) )
A(1,1) = DINTERVAL(2.,4.); A(1,2) = DINTERVAL(-2.,1.)
A(2,1) = DINTERVAL(-1.,2.); A(2,2) = DINTERVAL(2.,2.)
LDA = 2
B(1) = DINTERVAL(0.,2.); B(2) = DINTERVAL(0.,2.)
```



```

LDB = 2

!-----!

CALL DIGEMIP( DIM, NRHS, A, LDA,
B, LDB, INFO )
CALL DIGEHBS( TRANS, DIM, A, LDA,
B, LDB, INFO )

!-----!

IF( INFO /= 0 ) THEN
PRINT 400
ELSE
PRINT 600
DO I = 1, DIM
PRINT *, I, ' ) [', B(I), ']'
END DO
END IF

DEALLOCATE( A, B )

!-----!

300 FORMAT (/, ' **** SOLVING INTERVAL LINEAR SYSTEM ****',/, &
' by Hansen-Bliek-Rohn procedure ',/)
400 FORMAT (/, ' The matrix of the system is not an H-matrix',/, &
' Hansen-Bliek-Rohn procedure fails. ',/)
600 FORMAT (/, ' Enclosure of the solution set: ',/)

!-----!

END PROGRAM DIGEMIP_DIGEHS_EXAMPLE

```

This time, the output of the program is

```
**** SOLVING INTERVAL LINEAR SYSTEM****
```

by Hansen-Bliek-Rohn procedure

Enclosure of the solution set:

```
1 ) [ -4.23529411764708 10.7058823529412 ]
```

```
2 ) [ -6.70588235294119 10.8235294117647 ]
```

(the last digits may change for various computer architectures).

Example C-33. Computing enclosure for inverse interval matrix

Given an interval 2×2 matrix

$$\begin{pmatrix} 3 & [0, 1] \\ [1, 2] & [2, 3] \end{pmatrix}$$

the following Fortran 90 code computes an enclosure of its inverse interval matrix:

```

PROGRAM SIGESZI_EXAMPLE
!
!Example program inverting an interval matrix by Sczulz iterative procedure
!
!-----!
USE INTERVAL_ARITHMETIC
IMPLICIT NONE
!-----!
INTEGER, PARAMETER :: DIM = 2, LDA = 2
INTEGER :: INFO, I, J
TYPE(S_INTERVAL), ALLOCATABLE ::
A(:, :)
!-----!
PRINT 300
!-----!
!
! Initializing the input data -
!
ALLOCATE( A(LDA,DIM) )
A(1,1) = SINTERVAL(3.,3.); A(1,2) = SINTERVAL(0.,1.)
A(2,1) = SINTERVAL(1.,2.); A(2,2) = SINTERVAL(2.,3.)
!-----!
CALL SIGESZI ( DIM, A, LDA, INFO )
!-----!
PRINT 600
DO I = 1, DIM
PRINT *, ( '[', A(I,J), ']', J = 1,

```

```

DIM )
END DO
DEALLOCATE( A )
!-----!
300 FORMAT (/, ' **** INVERTING INTERVAL MATRIX ****',/, &
' by interval Schulz method ' )
400 FORMAT (/, ' Schulz inversion procedure failed. ',/)
600 FORMAT (/, ' Enclosure of the inverse matrix ',/)
!-----!
END PROGRAM SIGESZI_EXAMPLE

```

The output listing (with small variations depending on the architecture) looks as follows:

```

**** INVERTING INTERVAL MATRIX ****
by interval Schulz method
Enclosure of the inverse matrix
[ 0.2407409 0.5000001 ][ -0.2500000 0.1018518 ]
[ -0.5000000 5.5555239E-02 ][ 0.1388889 0.7500001 ]

```

At the same time, if we widen the (1,1) entry of the matrix to the interval [2, 3], the `sigeszi` procedure fails to compute a finite enclosure of the inverse to the new interval matrix

$$\begin{pmatrix} [2, 3] & [0, 1] \\ [1, 2] & [2, 3] \end{pmatrix}$$

Nevertheless, the interval linear system with such matrix can be successfully solved by specialized routines, for example, by interval Gauss method or interval Gauss-Seidel method (see [Example C-31](#)).

PDE Support Code Examples

This section presents code examples for routines described in the in the [“Partial Differential Equations Support” chapter](#). The examples are grouped in subsections

- [Trigonometric Transform Code Examples](#)
- [Poisson Library Code Examples](#).

Trigonometric Transforms Interface Code Examples

Code presented in this section computes solutions of three simple 1D Helmholtz problems with different boundary conditions: DD, NN and ND cases, where “D” denotes a Dirichlet boundary condition and “N” stands for a Neumann boundary condition. [Example C-34](#) implements the computations in C and [Example C-35](#) provides Fortran 90 code.

The algorithm of computing the solution uses [Trigonometric Transform routines](#), described in chapter 13. In the DD case, the sine transform is computed, the NN case uses the cosine transform and the ND case corresponds to the staggered cosine transform.

Other details of the Helmholtz problems being solved are printed out along with the computed solutions.

Upon successful execution of [Example C-34](#) the following text is printed out ([Example C-35](#) generates similar output):

Example of use of MKL Trigonometric Transforms

This example gives the solutions of the 1D differential problems
 with the equation $-u''+u=f(x)$, $0 < x < 1$,
 and with 3 types of boundary conditions:
 DD case: $u(0)=u(1)=0$,
 NN case: $u'(0)=u'(1)=0$,
 ND case: $u'(0)=u(1)=0$.

 In general, the error should be of order $O(1.0/n^{**2})$
 For this example, the value of n is 8
 The approximation error should be of order 5.0e-002 if everything is OK

 Note that n should be even to use Trigonometric Transforms !

DOUBLE PRECISION COMPUTATIONS

=====

The computed solution of DD problem is

u[0]= 0.000
 u[1]= 0.153
 u[2]= 0.524
 u[3]= 0.895
 u[4]= 1.049
 u[5]= 0.895

```
u[6]= 0.524
```

```
u[7]= 0.153
```

```
u[8]= 0.000
```

```
Error=4.873e-002
```

The computed solution of NN problem is

```
u[0]==-0.026
```

```
u[1]= 0.128
```

```
u[2]= 0.500
```

```
u[3]= 0.872
```

```
u[4]= 1.026
```

```
u[5]= 0.872
```

```
u[6]= 0.500
```

```
u[7]= 0.128
```

```
u[8]==-0.026
```

```
Error=2.583e-002
```

The computed solution of ND problem is

```
u[0]==-0.009
```

```
u[1]= 0.145
```

```
u[2]= 0.517
```

```
u[3]= 0.890
```

```
u[4]= 1.045
```

```
u[5]= 0.892
```

u[6]= 0.522

u[7]= 0.152

u[8]= 0.000

Error=4.470e-002

Example C-34 shows C code of the computations.

Example C-34. C Example to Solve a Set of 1D Helmholtz Problems

```

/*****
!
!                               INTEL CONFIDENTIAL
!
!   Copyright(C) 2005-2008 Intel Corporation. All Rights Reserved.
!
!   The source code contained or described herein and all documents related to
!   the source code ("Material") are owned by Intel Corporation or its suppliers
!   or licensors. Title to the Material remains with Intel Corporation or its
!   suppliers and licensors. The Material contains trade secrets and proprietary
!   and confidential information of Intel or its suppliers and licensors. The
!   Material is protected by worldwide copyright and trade secret laws and
!   treaty provisions. No part of the Material may be used, copied, reproduced,
!   modified, published, uploaded, posted, transmitted, distributed or disclosed
!   in any way without Intel's prior express written permission.
!
!   No license under any patent, copyright, trade secret or other intellectual
!   property right is granted to or conferred upon you by disclosure or delivery
!   of the Materials, either expressly, by implication, inducement, estoppel or
!   otherwise. Any license under such intellectual property rights must be
!   express and approved by Intel in
!   writing.
!
!
! *****/
! Content:
! Double precision C test example for trigonometric transforms
! *****/
!
! This example gives the solution of the 1D differential problems
! with the equation  $-u''+u=f(x)$ ,  $0<x<1$ , and with 3 types of boundary conditions:
!  $u(0)=u(1)=0$  (DD case), or  $u'(0)=u'(1)=0$  (NN case), or  $u'(0)=u(1)=0$  (ND case)
*/

```

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mkl_trig_transforms.h"

int main(void)
{
    MKL_INT n=8, i, k, tt_type;
    MKL_INT ir, ipar[128];
    double pi=3.14159265358979324, xi, c;
    double c1, c2, c3, c4, c5, c6;
    double *u, *f, *dpar, *lambda;
    DFTI_DESCRIPTOR_HANDLE handle = 0;

    /* Printing the header for the example */
    printf("\n Example of use of MKL Trigonometric Transforms\n");
    printf(" *****\n\n");
    printf(" This example gives the solutions of the 1D differential problems\n");
    printf(" with the equation -u''+u=f(x), 0<x<1, \n");
    printf(" and with 3 types of boundary conditions:\n");
    printf(" DD case: u(0)=u(1)=0,\n");
    printf(" NN case: u'(0)=u'(1)=0,\n");
    printf(" ND case: u'(0)=u(1)=0.\n");
    printf(" -----");
    printf(" In general, the error should be of order O(1.0/n**2)\n");
    printf(" For this example, the value of n is %d\n", n);
    printf(" The approximation error should be of order 5.0e-002 if everything is OK\n");
}
```

```

printf(" -----\n");
printf("          DOUBLE PRECISION COMPUTATIONS          \n");
printf(" =====\n\n");

u=(double*)malloc((n+1)*sizeof(double));
f=(double*)malloc((n+1)*sizeof(double));
/* NOTE: This example uses shorter dpar array of size 3n/2+2 instead of 5n/2+2
as only sine, cosine, and staggered cosine transforms are used. More details
can be found in Chapter 13 of the Intel MKL Reference Manual. */
dpar=(double*)malloc((3*n/2+2)*sizeof(double));
lambda=(double*)malloc((n+1)*sizeof(double));

for(i=0;i<=2;i++)
{
    /* Varying the type of the transform */
    tt_type=i;

    /* Computing test solutions u(x) */
    for(k=0;k<=n;k++)
    {
        xi=1.0E0*k/n;
        u[k]=pow(sin(pi*xi),2.0E0);
    }
    /* Computing the right-hand side f(x) */
    for(k=0;k<=n;k++)
    {
        f[k]=(4.0E0*(pi*pi)+1.0E0)*u[k]-2.0E0*(pi*pi);
    }
}

```

```
/* Computing the right-hand side for the algebraic system */
for(k=0;k<=n;k++)
{
    f[k]=f[k]/(n*n);
}
if (tt_type==0)
{
    /* The Dirichlet boundary conditions */
    f[0]=0.0E0;
    f[n]=0.0E0;
}
if (tt_type==2)
{
    /* The mixed Neumann-Dirichlet boundary conditions */
    f[n]=0.0E0;
}

/* Computing the eigenvalues for the three-point finite-difference problem */
if (tt_type==0||tt_type==1)
{
    for(k=0;k<=n;k++)
    {
        lambda[k]=pow(2.0E0*sin(0.5E0*pi*k/n),2.0E0)+1.0E0/(n*n);
    }
}
if (tt_type==2)
{
    for(k=0;k<=n;k++)
```

```
        {
            lambda[k]=pow(2.0E0*sin(0.25E0*pi*(2*k+1)/n),2.0E0)+1.0E0/(n*n);
        }
    }

    /* Computing the solution of 1D problem using trigonometric transforms
    First we initialize the transform */
    d_init_trig_transform(&n,&tt_type,ipar,dpar,&ir);
    if (ir!=0) goto FAILURE;
    /* Then we commit the transform. Note that the data in f will be changed
    at this stage
!
    If you want to keep them, save them in some other array
    before the call to the routine */
    d_commit_trig_transform(f,&handle,ipar,dpar,&ir);
    if (ir!=0) goto FAILURE;
    /* Now we can apply trigonometric transform */
    d_forward_trig_transform(f,&handle,ipar,dpar,&ir);
    if (ir!=0) goto FAILURE;

    /* Scaling the solution by the eigenvalues */
    for(k=0;k<=n;k++)
    {
        f[k]=f[k]/lambda[k];
    }

    /* Now we can apply trigonometric transform once again
    as ONLY input vector f has changed */
    d_backward_trig_transform(f,&handle,ipar,dpar,&ir);
```

```
if (ir!=0) goto FAILURE;

/* Cleaning the memory used by handle

Now we can use handle for other kind of trigonometric transform */
free_trig_transform(&handle,ipar,&ir);
if (ir!=0) goto FAILURE;

/* Performing the error analysis */
c1=0.0E0;
c2=0.0E0;
c3=0.0E0;
for(k=0;k<=n;k++)
{
    /* Computing the absolute value of the exact solution */
    c4=fabs(u[k]);
    /* Computing the absolute value of the computed solution
    Note that the solution is now in place of the former right-hand side ! */
    c5=fabs(f[k]);
    /* Computing the absolute error */
    c6=fabs(f[k]-u[k]);
    /* Computing the maximum among the above 3 values c4-c6 */
    if (c4>c1) c1=c4;
    if (c5>c2) c2=c5;
    if (c6>c3) c3=c6;
}

/* Printing the results */
if (tt_type==0)
{
```

```
        printf("The computed solution of DD problem is\n\n");
    }
    if (tt_type==1)
    {
        printf("The computed solution of NN problem is\n\n");
    }
    if (tt_type==2)
    {
        printf("The computed solution of ND problem is\n\n");
    }
    for(k=0;k<=n;k++)
    {
        printf("u[%d]=",k);
        printf("%6.3f\n",f[k]);
    }
    printf("\nError=%6.3e\n\n",c3/c1);
    if (c3/c1>0.05)
    {
        printf("The computed solution seems to be inaccurate. ");
        goto FAILURE;
    }
    /* End of the loop over the different kind of transforms and problems */
}

/* Success message to print if everything is OK */
printf("This example has successfully PASSED through all steps of computation!\n");
/* Jumping over failure message */
goto SUCCESS;
```

```
        /* Failure message to print if something went wrong */  
FAILURE: printf("Failed to compute the solution(s)...\n");  
  
SUCCESS: return 0;  
        /* End of the example code */  
}
```

Example C-35 shows Fortran 90 code of the computations.

Example C-35. Fortran Example to Solve a Set of 1D Helmholtz Problems

```

!*****
!
!                               INTEL CONFIDENTIAL
!
!   Copyright(C) 2005-2008 Intel Corporation. All Rights Reserved.
!
!   The source code contained or described herein and all documents related to
!   the source code ("Material") are owned by Intel Corporation or its suppliers
!   or licensors. Title to the Material remains with Intel Corporation or its
!   suppliers and licensors. The Material contains trade secrets and proprietary
!   and confidential information of Intel or its suppliers and licensors. The
!   Material is protected by worldwide copyright and trade secret laws and
!   treaty provisions. No part of the Material may be used, copied, reproduced,
!   modified, published, uploaded, posted, transmitted, distributed or disclosed
!   in any way without Intel's prior express written permission.
!
!   No license under any patent, copyright, trade secret or other intellectual
!   property right is granted to or conferred upon you by disclosure or delivery
!   of the Materials, either expressly, by implication, inducement, estoppel or
!   otherwise. Any license under such intellectual property rights must be
!   express and approved by Intel in writing.
!
!*****
!
!   Content:
!
!   Double precision Fortran90 test example for trigonometric transforms
!*****
!
!   This example gives the solution of the 1D differential problems
!   with the equation  $-u''+u=f(x)$ ,  $0<x<1$ , and with 3 types of boundary conditions:
!    $u(0)=u(1)=0$  (DD case), or  $u'(0)=u'(1)=0$  (NN case), or  $u'(0)=u(1)=0$  (ND case)

```

```
program d_tt_example_bvp
  use mkl_dfti
  use mkl_trig_transforms
  implicit none

  integer n, i, k,j, tt_type
  integer ir, ipar(128)
  parameter (n=8)
  double precision pi, xi
  double precision c1, c2, c3, c4, c5, c6
  ! NOTE: This example uses shorter dpar array of size 3n/2+2 instead of 5n/2+2

  ! as only sine, cosine, and staggered cosine transforms are used. More details
```

```

! can be found in Chapter 13 of the Intel MKL Reference Manual. */
double precision u(n+1), f(n+1), dpar(3*n/2+2), lambda(n+1)
parameter (pi=3.14159265358979324D0)
type(dfti_descriptor), pointer :: handle

! Printing the header for the example
print *, ''
print *, ' Example of use of MKL Trigonometric Transforms'
print *, ' *****'
print *, ''
print *, ' This example gives the solution of the 1D differential problems'
print *, ' with the equation -u''+u=f(x), 0<x<1, '
print *, ' and with 3 types of boundary conditions:'
print *, ' DD case: u(0)=u(1)=0,'
print *, ' NN case: u''(0)=u''(1)=0,'
print *, ' ND case: u''(0)=u(1)=0.'
print *, ' -----'
print *, ' In general, the error should be of order O(1.0/n**2)'
print *, ' For this example, the value of n is', n
print *, ' The approximation error should be of order 0.5E-01, if everything is OK'
print *, ' -----'
print *, '
                                DOUBLE PRECISION COMPUTATIONS
                                '
print *, ' ====='
print *, ''

do i=0,2
! Varying the type of the transform
    tt_type=i

```

```
! Computing test solution u(x)
  do k=1,n+1
    xi=1.0D0*(k-1)/n
    u(k)=dsin(pi*xi)**2
  end do
! Computing the right-hand side f(x)
  do k=1,n+1
    f(k)=(4.0D0*(pi**2)+1.0D0)*u(k)-2.0D0*(pi**2)
  end do
! Computing the right-hand side for the algebraic system
  do k=1,n+1
    f(k)=f(k)/(n**2)
  end do
  if (tt_type.eq.0) then
! The Dirichlet boundary conditions
    f(1)=0.0D0
    f(n+1)=0.0D0
  end if
  if (tt_type.eq.2) then
! The mixed Neumann-Dirichlet boundary conditions
    f(n+1)=0.0D0
  end if

! Computing the eigenvalues for the three-point finite-difference problem
  if (tt_type.eq.0.or.tt_type.eq.1) then
    do k=1,n+1
      lambda(k)=(2.0D0*dsin(0.5D0*pi*(k-1)/n))**2+1.0D0/(n**2)
    end do
```

```
end if
if (tt_type.eq.2) then
  do k=1,n+1
    lambda(k)=(2.0D0*dsin(0.25D0*pi*(2*k-1)/n))**2+1.0D0/(n**2)
  end do
end if

! Computing the solution of 1D problem using trigonometric transforms
! First we initialize the transform
  CALL D_INIT_TRIG_TRANSFORM(n,tt_type,ipar,dpar,ir)
  if (ir.ne.0) goto 99
! Then we commit the transform. Note that the data in f will be changed at this stage !
! If you want to keep them, save them in some other array before the call to the routine
  CALL D_COMMIT_TRIG_TRANSFORM(f,handle,ipar,dpar,ir)
  if (ir.ne.0) goto 99
! Now we can apply trigonometric transform
  CALL D_FORWARD_TRIG_TRANSFORM(f,handle,ipar,dpar,ir)
  if (ir.ne.0) goto 99

! Scaling the solution by the eigenvalues
  do k=1,n+1
    f(k)=f(k)/lambda(k)
  end do
! Now we can apply trigonometric transform once again as ONLY input vector f has changed
  CALL D_BACKWARD_TRIG_TRANSFORM(f,handle,ipar,dpar,ir)
  if (ir.ne.0) goto 99
! Cleaning the memory used by handle
! Now we can use handle for other KIND of trigonometric transform
```

```
CALL FREE_TRIG_TRANSFORM(handle,ipar,ir)
if (ir.ne.0) goto 99

! Performing the error analysis
c1=0.0D0
c2=0.0D0
c3=0.0D0
do k=1,n+1
! Computing the absolute value of the exact solution
c4=dabs(u(k))
! Computing the absolute value of the computed solution
! Note that the solution is now in place of the former right-hand side !
c5=dabs(f(k))
! Computing the
absolute error
c6=dabs(f(k)-u(k))
! Computing the maximum among the above 3 values c4-c6
if (c4.gt.c1) c1=c4
if (c5.gt.c2) c2=c5
if (c6.gt.c3) c3=c6
end
do

! Printing the results
if (tt_type.eq.0) then
print *, 'The computed solution of DD problem is'
end if
if (tt_type.eq.1) then
print *, 'The computed solution of NN problem is'
```

```
end if
if (tt_type.eq.2) then
  print *, 'The computed solution of ND problem is'
endif
print *, ''
do k=1,n+1
  write(*,11) k,f(k)
end do
print *, ''
write(*,12) c3/c1
print *, ''
if (c3/c1.ge.5.0D-2) then
  print *, 'The computed solution seems to be inaccurate.'
  goto 99
endif
! End of the loop over the different kind of transforms and problems
end do

! Success message to print if everything is OK
print *, 'This example has successfully PASSED through all steps of computation!'
! Jumping over failure message
goto 1
! Failure message to print if something went wrong
99  continue
   print *, 'FAILED to compute the solution(s)...'

1   continue
```

```
! Print formats
11     format(1x,'u(',I1,')=' ,F6.3)
12     format(1x,'Relative error =' ,E10.3)

! End of the example code
end
```

Poisson Library Code Examples

Cartesian case

The code below computes an approximate solution of a 2D Poisson problem

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = 8\pi^2 \sin 2\pi x \cdot \sin 2\pi y$$

in the rectangle $0 < x < 1, 0 < y < 1$.

The following boundary conditions are imposed for the problem:

- Dirichlet boundary condition

$$u(0, y) = u(1, y) = 1$$

for $0 \leq y \leq 1$.

- Neumann boundary condition

$$\frac{\partial u}{\partial n}(x, 0) = -2\pi \sin 2\pi x, \frac{\partial u}{\partial n}(x, 1) = 2\pi \sin 2\pi x$$

for $0 < x < 1$.

The exact solution is known to be

$$u(x, y) = \sin 2\pi x \cdot \sin 2\pi y + 1$$

Example [C-36](#) implements the computations in C and Example [C-37](#) provides Fortran 90 code. Mind that PL interface cannot be invoked from Fortran 77 due to restrictions imposed by the use of Intel MKL FFT interface.

The algorithm of computing the approximate solution of a Poisson Problem uses [Poisson Library routines](#), described in chapter 13. Details of the Poisson problem being solved and the errors are printed out between the computed solution and the exact one.

Upon successful execution of Example C-36 the following text is printed out (Example C-37 produces similar output):

Example of use of MKL Poisson Library

```
This example gives the solution of 2D Poisson problem
with the equation -u_xx-u_yy=f(x,y), 0<x<1, 0<y<1,
f(x,y)=(8*pi*pi)*sin(2*pi*x)*sin(2*pi*y),
and with the following boundary conditions:
  u(0,y)=u(1,y)=1 (Dirichlet boundary conditions),
-u_y(x,0)=-2.0*pi*sin(2*pi*x) (Neumann boundary condition),
  u_y(x,1)= 2.0*pi*sin(2*pi*x) (Neumann boundary condition).
```

```
-----
In general, the error should be of order O(1.0/nx^2+1.0/ny^2)
For this example, the value of nx=ny is 6
The approximation error should be of order 1.0e-01, if everything is OK
-----
```

DOUBLE PRECISION COMPUTATIONS

```
=====
The number of mesh intervals in x-direction is nx=6
The number of mesh intervals in y-direction is ny=6
```

```
In the mesh point (0.167,0.000) the error between the computed and the true solution
is equal to -7.505e-02
In the mesh point (0.167,0.167) the error between the computed and the true solution
is equal to  4.432e-02
In the mesh point (0.167,0.333) the error between the computed and the true solution
is equal to  6.309e-02
```

In the mesh point (0.167,0.500) the error between the computed and the true solution is equal to -5.551e-16

In the mesh point (0.167,0.667) the error between the computed and the true solution is equal to -6.309e-02

In the mesh point (0.167,0.833) the error between the computed and the true solution is equal to -4.432e-02

In the mesh point (0.167,1.000) the error between the computed and the true solution is equal to 7.505e-02

Double precision 2D Poisson example has successfully PASSED
through all steps of computation!

Note that actual figures in the error may slightly differ from the figures printed above depending on the architecture and operating system used to run the example.

Example [C-36](#) shows C code of the computations.

Example C-36. C Example to Solve 2D Poisson Problem

```

/*****
/*
/*                               INTEL CONFIDENTIAL
/*
/*   Copyright(C) 2006-2008 Intel Corporation. All Rights Reserved.
/*
/*   The source code contained or described herein and all documents related to
/*   the source code ("Material") are owned by Intel Corporation or its suppliers
/*   or licensors. Title to the Material remains with Intel Corporation or its
/*   suppliers and licensors. The Material contains trade secrets and proprietary
/*   and confidential information of Intel or its suppliers and licensors. The
/*   Material is protected by worldwide copyright and trade secret laws and
/*   treaty provisions. No part of the Material may be used, copied, reproduced,
/*   modified, published, uploaded, posted, transmitted, distributed or disclosed
/*   in any way without Intel");s prior express written permission.
/*
/*   No license under any patent, copyright, trade secret or other intellectual
/*   property right is granted to or conferred upon you by disclosure or delivery
/*   of the Materials, either expressly, by implication, inducement, estoppel or
/*   otherwise. Any license under such intellectual property rights must be
/*   express and approved by Intel in writing.
/*
/*
/*****
/* Content:
/* C double precision example of solving 2D Poisson problem in a
/* rectangular domain using MKL Poisson Library
/*
/*****/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

```

```

/* Include Poisson Library header files */
#include "mkl_poisson.h"
int main(void)
{
    MKL_INT nx=6, ny=6;
    double pi=3.14159265358979324;
    MKL_INT ix, iy, i, stat;
    MKL_INT ipar[128];
    double ax, bx, ay, by, lx, ly, hx, hy, xi, yi, cx, cy, cl;
    double *dpar, *f, *u, *bd_ax, *bd_bx, *bd_ay, *bd_by;
    double q;
    DFTI_DESCRIPTOR_HANDLE xhandle = 0;
    char *BCTYPE;

    /* Printing the header for the example */
    printf("\n Example of use of MKL Poisson Library\n");
    printf(" *****\n\n");
    printf(" This example gives the solution of 2D Poisson problem\n");
    printf(" with the equation -u_xx-u_yy=f(x,y), 0<x<1, 0<y<1,\n");
    printf(" f(x,y)=(8*pi*pi)*sin(2*pi*x)*sin(2*pi*y),\n");
    printf(" and with the following boundary conditions:\n");
    printf("  u(0,y)=u(1,y)=1 (Dirichlet boundary conditions),\n");
    printf(" -u_y(x,0)=-2.0*pi*sin(2*pi*x) (Neumann boundary condition),\n");
    printf("  u_y(x,1)= 2.0*pi*sin(2*pi*x) (Neumann boundary condition).\n");
    printf(" ----- \n");
    printf(" In general, the error should be of order O(1.0/nx^2+1.0/ny^2)\n");
    printf(" For this example, the value of nx=ny is %d\n", nx);
    printf(" The approximation error should be of order 1.0e-01, if everything is OK\n");
    printf(" ----- \n");

```

```

printf("                                DOUBLE PRECISION COMPUTATIONS                                \n");
printf(" =====\n\n");

dpar=(double*)malloc((13*nx/2+7)*sizeof(double));
f=(double*)malloc((nx+1)*(ny+1)*sizeof(double));
u=(double*)malloc((nx+1)*(ny+1)*sizeof(double));
bd_ax=(double*)malloc((ny+1)*sizeof(double));
bd_bx=(double*)malloc((ny+1)*sizeof(double));
bd_ay=(double*)malloc((nx+1)*sizeof(double));
bd_by=(double*)malloc((nx+1)*sizeof(double));

/* Defining the rectangular domain 0<x<1, 0<y<1 for 2D Poisson Solver */
ax=0.0E0;
bx=1.0E0;
ay=0.0E0;
by=1.0E0;

/*****

Setting the coefficient q to 0.

Note that this is the way to use Helmholtz Solver to solve Poisson problem!
*****/

q=0.0E0;

/* Computing the mesh size hx in x-direction */
lx=bx-ax;
hx=lx/nx;

/* Computing the mesh size hy in y-direction */
ly=by-ay;
hy=ly/ny;

/* Filling in the values of the TRUE solution u(x,y)=sin(2*pi*x)*sin(2*pi*y)+1
in the mesh points into the array u
Filling in the right-hand side f(x,y)=(8*pi*pi+q)*sin(2*pi*x)*sin(2*pi*y)+q

```

in the mesh points into the array f.

We choose the right-hand side to correspond to the TRUE solution of Poisson equation.

Here we are using the mesh sizes hx and hy computed before to compute

the coordinates (xi,yi) of the mesh points */

```
for(iy=0;iy<=ny;iy++)
```

```
{
```

```
    for(ix=0;ix<=nx;ix++)
```

```
    {
```

```
        xi=hx*ix/lx;
```

```
        yi=hy*iy/ly;
```

```
        cx=sin(2*pi*xi);
```

```
        cy=sin(2*pi*yi);
```

```
        u[ix+iy*(nx+1)]=1.0E0*cx*cy;
```

```
        f[ix+iy*(nx+1)]=(8.0E0*pi*pi)*u[ix+iy*(nx+1)];
```

```
        u[ix+iy*(nx+1)]=u[ix+iy*(nx+1)]+1.0E0;
```

```
    }
```

```
}
```

```
/* Setting the type of the boundary conditions on each side of the rectangular domain:
```

```
On the boundary laying on the line x=0(=ax) Dirichlet boundary condition will be used
```

```
On the boundary laying on the line x=1(=bx) Dirichlet boundary condition will be used
```

```
On the boundary laying on the line y=0(=ay) Neumann boundary condition will be used
```

```
On the boundary laying on the line y=1(=by) Neumann boundary condition will be used */
```

```
BCtype = "DDNN";
```

```
/* Setting the values of the boundary function G(x,y) that is equal to the TRUE solution
```

```
in the mesh points laying on Dirichlet boundaries */
```

```
for(iy=0;iy<=ny;iy++)
```

```
{
```

```
    bd_ax[iy]=1.0E0;
```

```

        bd_bx[iy]=1.0E0;
    }
/* Setting the values of the boundary function g(x,y) that is equal to the normal derivative
of the TRUE solution in the mesh points laying on Neumann boundaries */
for(ix=0;ix<=nx;ix++)
{
    bd_ay[ix]=-2.0*pi*sin(2*pi*ix/nx);
    bd_by[ix]= 2.0*pi*sin(2*pi*ix/nx);
}
/* Initializing ipar array to make it free from garbage */
for(i=0;i<128;i++)
{
    ipar[i]=0;
}
/* Initializing simple data structures of Poisson Library for 2D Poisson Solver */
d_init_Helmholtz_2D(&ax, &bx, &ay, &by, &nx, &ny, Bctype, &q, ipar, dpar, &stat);
if (stat!=0) goto FAILURE;
/* Initializing complex data structures of Poisson Library for 2D Poisson Solver
NOTE: Right-hand side f may be altered after the Commit step. If you want to keep it,
you should save it in another memory location! */
d_commit_Helmholtz_2D(f, bd_ax, bd_bx, bd_ay, bd_by, &xhandle, ipar, dpar, &stat);
if (stat!=0) goto FAILURE;
/* Computing the approximate solution of 2D Poisson problem
NOTE: Boundary data stored in the arrays bd_ax, bd_bx, bd_ay, bd_by should not be changed
between the Commit step and the subsequent call to the Solver routine/*
Otherwise the results may be wrong. */
d_Helmholtz_2D(f, bd_ax, bd_bx, bd_ay, bd_by, &xhandle, ipar, dpar, &stat);
if (stat!=0) goto FAILURE;

```



```
/* Cleaning the memory used by xhandle */
free_Helmholtz_2D(&xhandle, ipar, &stat);
if (stat!=0) goto FAILURE;
/* Now we can use xhandle to solve another 2D Poisson problem*/
/* Printing the results */
printf("The number of mesh intervals in x-direction is nx=%d\n", nx);
printf("The number of mesh intervals in y-direction is ny=%d\n\n", ny);
/* Watching the error along the line x=hx */
ix=1;
c1=0.0;
for(iy=0;iy<=ny;iy++)
{
    printf("In the mesh point (%5.3f,%5.3f) the error between the computed and the true
solution is equal to %10.3e\n", ix*hx, iy*hy, f[ix+iy*(nx+1)]-u[ix+iy*(nx+1)]);
    if (c1<fabs(f[ix+iy*(nx+1)]-u[ix+iy*(nx+1)])) c1 =
fabs(f[ix+iy*(nx+1)]-u[ix+iy*(nx+1)]);
}
if (c1>1.0e-01)
{
    printf("The computed solution seems to be inaccurate. ");
    goto FAILURE;
}
/* Success message to print if everything is OK */
printf("\n Double precision 2D Poisson example has successfully PASSED\n");
printf(" through all steps of computation!\n");
/* Jumping over failure message */
goto SUCCESS;
/* Failure message to print if something went wrong */
FAILURE: printf("\nDouble precision 2D Poisson example FAILED to compute the solution...\n");
```

```
SUCCESS: return 0;  
    /* End of the example code */  
}
```

Example [C-37](#) shows Fortran 90 code for the problem.

Example C-37. Fortran Example to Solve 2D Poisson Problem

```

!*****
!
!                               INTEL CONFIDENTIAL
!
!   Copyright(C) 2006-2008 Intel Corporation. All Rights Reserved.
!
!   The source code contained or described herein and all documents related to
!   the source code ("Material") are owned by Intel Corporation or its suppliers
!   or licensors. Title to the Material remains with Intel Corporation or its
!   suppliers and licensors. The Material contains trade secrets and proprietary
!   and confidential information of Intel or its suppliers and licensors. The
!   Material is protected by worldwide copyright and trade secret laws and
!   treaty provisions. No part of the Material may be used, copied, reproduced,
!   modified, published, uploaded, posted, transmitted, distributed or disclosed
!   in any way without Intel's prior express written permission.
!
!   No license under any patent, copyright, trade secret or other intellectual
!   property right is granted to or conferred upon you by disclosure or delivery
!   of the Materials, either expressly, by implication, inducement, estoppel or
!   otherwise. Any license under such intellectual property rights must be
!   express and approved by Intel in writing.
!
!*****
!
!   Content:
!
!   Fortran-90 double precision example of solving 2D Poisson problem in a
!   rectangular domain using MKL Poisson Library
!
!*****
program Poisson_2D_double_precision
! Include modules defined by mkl_poisson.f90 and mkl_dfti.f90 header files
use mkl_poisson

```

```

implicit none
integer nx,ny
parameter(nx=6, ny=6)
double precision pi
parameter(pi=3.14159265358979324D0)
integer ix, iy, i, stat
integer ipar(128)
double precision ax, bx, ay, by, lx, ly, hx, hy, xi, yi, cx, cy, cl
double precision dpar(13*nx/2+7)

! Note that proper packing of data in right-hand side array f is
! automatically provided by the following declarations of the array
double precision f(nx+1,ny+1), u(nx+1,ny+1)
double precision bd_ax(ny+1), bd_bx(ny+1), bd_ay(nx+1), bd_by(nx+1)
double precision q
type(DFTI_DESCRIPTOR), pointer :: xhandle
character(4) Bctype

! Printing the header for the example
      print *, ''
      print *, ' Example of use of MKL Poisson Library'
      print *, ' *****'
      print *, ''
      print *, ' This example gives the solution of 2D Poisson problem'
      print *, ' with the equation -u_xx-u_yy=f(x,y), 0<x<1, 0<y<1,'
      print *, ' f(x,y)=(8*pi*pi)*sin(2*pi*x)*sin(2*pi*y),'
      print *, ' and with the following boundary conditions:'
      print *, ' u(0,y)=u(1,y)=1 (Dirichlet boundary conditions),'
      print *, ' -u_y(x,0)=-2.0*pi*sin(2*pi*x) (Neumann boundary condition),'
      print *, ' u_y(x,1)= 2.0*pi*sin(2*pi*x) (Neumann boundary condition).'
```

```

    print *, ' -----'
    print *, ' In general, the error should be of order  $O(1.0/nx^2+1.0/ny^2)$ '
    print '(lx,a,11)', ' For this example, the value of nx=ny is ', nx
    print *, ' The approximation error should be of order 0.1D+0, if everything is OK'
    print *, ' -----'
    print *, '
                                DOUBLE PRECISION COMPUTATIONS
                                '
    print *, ' ====='
    print *, ''

! Defining the rectangular domain 0<x<1, 0<y<1 for 2D Poisson Solver
ax=0.0D0
bx=1.0D0
ay=0.0D0
by=1.0D0

!*****
! Setting the coefficient q to 0.
! Note that this is the way to use Helmholtz Solver to solve Poisson problem!
!*****
q=0.0D0
! Computing the mesh size hx in x-direction
lx=bx-ax
hx=lx/nx
! Computing the mesh size hy in y-direction
ly=by-ay
hy=ly/ny
! Filling in the values of the TRUE solution  $u(x,y)=\sin(2\pi x)\sin(2\pi y)+1$ 
! in the mesh points into the array u

```

```

! Filling in the right-hand side  $f(x,y)=(8\pi^2+q)\sin(2\pi x)\sin(2\pi y)+q$ 
! in the mesh points into the array f.
! We choose the right-hand side to correspond to the TRUE solution of Poisson equation.
! Here we are using the mesh sizes  $h_x$  and  $h_y$  computed before to compute
! the coordinates  $(x_i,y_i)$  of the mesh points
do iy=1,ny+1
    do ix=1,nx+1
        xi=hx*(ix-1)/lx
        yi=hy*(iy-1)/ly
        cx=dsin(2*pi*xi)
        cy=dsin(2*pi*yi)
        u(ix,iy)=1.0D0*cx*cy
        f(ix,iy)=(8.0D0*pi**2)*u(ix,iy)
        u(ix,iy)=u(ix,iy)+1.0D0
    enddo
enddo

! Setting the type of the boundary conditions on each side of the rectangular domain:
! On the boundary laying on the line  $x=0(=a_x)$  Dirichlet boundary condition will be used
! On the boundary laying on the line  $x=1(=b_x)$  Dirichlet boundary condition will be used
! On the boundary laying on the line  $y=0(=a_y)$  Neumann boundary condition will be used
! On the boundary laying on the line  $y=1(=b_y)$  Neumann boundary condition will be used
Bctype = 'DDNN'

! Setting the values of the boundary function  $G(x,y)$  that is equal to the TRUE solution
! in the mesh points laying on Dirichlet boundaries
do iy = 1,ny+1
    bd_ax(iy) = 1.0D0
    bd_bx(iy) = 1.0D0
enddo

```

```
! Setting the values of the boundary function g(x,y) that is equal to the normal derivative
! of the TRUE solution in the mesh points laying on Neumann boundaries
do ix = 1,nx+1
    bd_ay(ix) = -2.0*pi*dsin(2*pi*(ix-1)/nx)
    bd_by(ix) =  2.0*pi*dsin(2*pi*(ix-1)/nx)
enddo

! Initializing ipar array to make it free from garbage
do i=1,128
    ipar(i)=0
enddo

! Initializing simple data structures of Poisson Library for 2D Poisson Solver
call d_init_Helmholtz_2D(ax, bx, ay, by, nx, ny, Bctype, q, ipar, dpar, stat)
if (stat.ne.0) goto 999

! Initializing complex data structures of Poisson Library for 2D Poisson Solver
! NOTE: Right-hand side f may be altered after the Commit step. If you want to keep it,
! you should save it in another memory location!
call d_commit_Helmholtz_2D(f, bd_ax, bd_bx, bd_ay, bd_by, xhandle, ipar, dpar, stat)
if (stat.ne.0) goto 999

! Computing the approximate solution of 2D Poisson problem
! NOTE: Boundary data stored in the arrays bd_ax, bd_bx, bd_ay, bd_by should not be changed
! between the Commit step and the subsequent call to the Solver routine!
! Otherwise the results may be wrong.
call d_Helmholtz_2D(f, bd_ax, bd_bx, bd_ay, bd_by, xhandle, ipar, dpar, stat)
if (stat.ne.0) goto 999

! Cleaning the memory used by xhandle
call free_Helmholtz_2D(xhandle, ipar, stat)
if (stat.ne.0) goto 999

! Now we can use xhandle to solve another 2D Poisson problem
```

```
! Printing the results
write(*,10) nx
write(*,11) ny
print *, ''
! Watching the error along the line x=hx
ix=2
c1 = 0.0
do iy=1,ny+1
    write(*,12) (ix-1)*hx, (iy-1)*hy, f(ix,iy)-u(ix,iy)
    if (dabs(f(ix,iy)-u(ix,iy)).ge.c1) c1 = dabs(f(ix,iy)-u(ix,iy))
enddo
print *, ''
if (c1.ge.0.1D+0) then
    print *, 'The computed solution seems to be inaccurate.'
    goto 999
endif
! Success message to print if everything is OK
    print *, ' Double precision 2D Poisson example has successfully PASSED'
    print *, ' through all steps of computation!'
! Jumping over failure message
    go to 1
! Failure message to print if something went wrong
999 print *, 'Double precision 2D Poisson example FAILED to compute the solution...'
1 continue
10    format(1x,'The number of mesh intervals in x-direction is nx=',I1)
11    format(1x,'The number of mesh intervals in y-direction is ny=',I1)
12    format(1x,'In the mesh point (' ,F5.3,' ,',F5.3,') the error between the computed
```



```

1 and the true solution is equal to ', E10.3)

! End of the example code
end

```

Spherical case

The code below computes an approximate solution of a Helmholtz problem on the entire sphere:

$$-\Delta_s u + u = 3 \cos \theta, \quad \Delta_s = \frac{1}{\sin^2 \theta} \frac{\partial^2}{\partial \varphi^2} + \frac{1}{\sin \theta} \frac{\partial}{\partial \theta} \left(\sin \theta \frac{\partial}{\partial \theta} \right)$$

with $a_\phi = 0$, $b_\phi = 2\pi$, $a_\theta = 0$, $b_\theta = \pi$, and a boundary condition

$$\left(\sin \theta \frac{\partial u}{\partial \theta} \right)_{\substack{\theta \rightarrow 0 \\ \theta \rightarrow \pi}} = 0$$

at the poles (*periodic* case).

The exact solution is known to be $u(\phi, \theta) = \cos \theta$.

Example [C-38](#) implements the computations in C and Example [C-39](#) provides Fortran 90 code. Mind that PL interface cannot be invoked from Fortran 77 due to restrictions imposed by the use of Intel MKL FFT interface.

The algorithm of computing the approximate solution of a Poisson Problem uses [Poisson Library routines](#), described in chapter 13. Details of the Poisson problem being solved and the errors are printed out between the computed solution and the exact one. Upon successful execution of Example C-38 the following text is printed out (Example C-39 produces similar output):

Example of use of MKL Poisson Library

This example gives the solution of Helmholtz problem on a whole sphere
 $0 < p < 2\pi$, $0 < t < \pi$, with Helmholtz coefficient $q=1$ and right-hand side
 $f(p,t)=3\cos(t)$

 In general, the error should be of order $O(1.0/np^2+1.0/nt^2)$

For this example, the value of $np=nt$ is 8

The approximation error should be of order $1.5e-01$, if everything is OK

 Note that np should be even to solve the PERIODIC problem!

 DOUBLE PRECISION COMPUTATIONS

=====

The number of mesh intervals in phi-direction is $np=8$

The number of mesh intervals in theta-direction is $nt=8$

In the mesh point (0.785,0.000) the error between the computed and the true solution
 is equal to $-1.402e-001$

In the mesh point (0.785,0.393) the error between the computed and the true solution
 is equal to $-3.097e-002$

In the mesh point (0.785,0.785) the error between the computed and the true solution
 is equal to $-6.036e-003$

In the mesh point (0.785,1.178) the error between the computed and the true solution is equal to 2.964e-004

In the mesh point (0.785,1.571) the error between the computed and the true solution is equal to 4.979e-017

In the mesh point (0.785,1.963) the error between the computed and the true solution is equal to -2.964e-004

In the mesh point (0.785,2.356) the error between the computed and the true solution is equal to 6.036e-003

In the mesh point (0.785,2.749) the error between the computed and the true solution is equal to 3.097e-002

In the mesh point (0.785,3.142) the error between the computed and the true solution is equal to 1.402e-001

Double precision Helmholtz example on a whole sphere has successfully PASSED through all steps of computation!

Note that actual figures in the error may slightly differ from the figures printed above depending on the architecture and operating system used to run the example.

Example [C-38](#) shows C code for the problem.

Example C-38. C Example to Solve Helmholtz Problem on a Sphere

```

/*****
/*
/*                               INTEL CONFIDENTIAL
/*
/*   Copyright(C) 2006-2008 Intel Corporation. All Rights Reserved.
/*
/*   The source code contained or described herein and all documents related to
/*   the source code ("Material") are owned by Intel Corporation or its suppliers
/*   or licensors. Title to the Material remains with Intel Corporation or its
/*   suppliers and licensors. The Material contains trade secrets and proprietary
/*   and confidential information of Intel or its suppliers and licensors. The
/*   Material is protected by worldwide copyright and trade secret laws and
/*   treaty provisions. No part of the Material may be used, copied, reproduced,
/*   modified, published, uploaded, posted, transmitted, distributed or disclosed
/*   in any way without Intel's prior express written permission.
/*
/*   No license under any patent, copyright, trade secret or other intellectual
/*   property right is granted to or conferred upon you by disclosure or delivery
/*   of the Materials, either expressly, by implication, inducement, estoppel or
/*   otherwise. Any license under such intellectual property rights must be
/*   express and approved by Intel in writing.
/*
/*
/*****
/* Content:
/* C double precision example of solving Helmholtz problem on a whole sphere
/* using MKL Poisson Library
/*
/*****/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

```

```

/* Include Poisson Library header files */
#include "mkl_poisson.h"
int main(void)
{
    /* Note that the size of the transform np must be even !!! */
    MKL_INT np=8, nt=8;
    double pi=3.14159265358979324;
    MKL_INT ip, it, i, stat;
    MKL_INT ipar[128];
    double ap, bp, at, bt, lp, lt, hp, ht, theta_i, ct, cl;
    double *dpar, *f, *u;
    double q;
    DFTI_DESCRIPTOR_HANDLE handle_s = 0;
    DFTI_DESCRIPTOR_HANDLE handle_c = 0;
    /* Printing the header for the example */
    printf("\n Example of use of MKL Poisson Library\n");
    printf(" *****\n\n");
    printf(" This example gives the solution of Helmholtz problem on a whole sphere\n");
    printf(" 0<p<2*pi, 0<t<pi, with Helmholtz coefficient q=1 and right-hand side\n");
    printf(" f(p,t)=3*cos(t)\n");
    printf(" -----\n");
    printf(" In general, the error should be of order O(1.0/np^2+1.0/nt^2)\n");
    printf(" For this example, the value of np=nt is %d\n", np);
    printf(" The approximation error should be of order 1.5e-01, if everything is OK\n");
    printf(" -----\n");
    printf("   Note that np should be even to solve the PERIODIC problem!\n");
    printf(" -----\n");
    printf("                                DOUBLE PRECISION COMPUTATIONS                                \n");

```

```

printf(" =====\n\n");
dpar=(double*)malloc((5*np/2+nt+10)*sizeof(double));
f=(double*)malloc((np+1)*(nt+1)*sizeof(double));
u=(double*)malloc((np+1)*(nt+1)*sizeof(double));
/* Defining the rectangular domain on a sphere 0<p<2*pi, 0<t<pi
for Helmholtz Solver on a sphere */
/* Poisson Library will automatically detect that this problem is on a whole sphere! */
ap=0.0E0;
bp=2*pi;
at=0.0E0;
bt=pi;
/* Setting the coefficient q to 1.0E0 for Helmholtz problem */
/* If you like to solve Poisson problem, please set q to 0.0E0 */
q=1.0E0;
/* Computing the mesh size hp in phi-direction */
lp=bp-ap;
hp=lp/np;
/* Computing the mesh size ht in theta-direction */
lt=bt-at;
ht=lt/nt;
/* Filling in the values of the TRUE solution u(p,t)=cos(t)
in the mesh points into the array u
Filling in the right-hand side f(p,t)=(2+q)*cos(t)
in the mesh points into the array f.
We choose the right-hand side to correspond to the TRUE solution
of Helmholtz equation on a sphere.
Here we are using the mesh sizes hp and ht computed before to compute
the coordinates (phi_i,theta_i) of the mesh points */

```

```
for(it=0;it<=nt;it++)
{
    for(ip=0;ip<=np;ip++)
    {
        theta_i=ht*it;
        ct=cos(theta_i);
        u[ip+it*(np+1)]=ct;
        f[ip+it*(np+1)]=ct*(2.+q);
    }
}

/* Initializing ipar array to make it free from garbage */
for(i=0;i<128;i++)
{
    ipar[i]=0;
}

/* Initializing simple data structures of Poisson Library
for Helmholtz Solver on a sphere */
/* As we are looking for the solution on a whole sphere, this is a PERIDOC problem */
/* Therefore, the routines ending with "_p" are used to find the solution */
d_init_sph_p(&ap,&bp,&at,&bt,&np,&nt,&q,ipar,dpar,&stat);
if (stat!=0) goto FAILURE;

/* Initializing complex data structures of Poisson Library for Helmholtz Solver on a sphere
NOTE: Right-hand side f may be altered after the Commit step. If you want to keep it,
you should save it in another memory location! */
d_commit_sph_p(f,&handle_s,&handle_c,ipar,dpar,&stat);
if (stat!=0) goto FAILURE;

/* Computing the approximate solution of Helmholtz problem on a whole sphere */
d_sph_p(f,&handle_s,&handle_c,ipar,dpar,&stat);
```

```

    if (stat!=0) goto FAILURE;

    /* Cleaning the memory used by handle_s and handle_c */
    free_sph_p(&handle_s,&handle_c,ipar,&stat);
    if (stat!=0) goto FAILURE;

    /* Now we can use handle_s and handle_c to solve another Helmholtz problem */
    /* after a proper initialization */
    /* Printing the results */
    printf("The number of mesh intervals in phi-direction is np=%d\n", np);
    printf("The number of mesh intervals in theta-direction is nt=%d\n\n", nt);

    /* Watching the error along the line phi=hp */
    ip=1;
    c1=0.0;
    for(it=0;it<=nt;it++)
    {
        printf("In the mesh point (%5.3f,%5.3f) the error between the computed and the true
        solution is equal to %10.3e\n", ip*hp, it*ht, f[ip+it*(np+1)]-u[ip+it*(np+1)]);

        if (c1<fabs(f[ip+it*(np+1)]-u[ip+it*(np+1)])) c1 =
        fabs(f[ip+it*(np+1)]-u[ip+it*(np+1)]);
    }
    if (c1>1.5e-01)
    {
        printf("The computed solution seems to be inaccurate. ");

        goto FAILURE;
    }

    /* Success message to print if everything is OK */
    printf("\n Double precision Helmholtz example on a whole sphere has successfully PASSED\n");
    printf(" through all steps of computation!\n");

    /* Jumping over failure message */
    goto SUCCESS;

```

```
/* Failure message to print if something went wrong */  
FAILURE: printf("\nDouble precision Helmholtz example on a whole sphere has FAILED to compute  
the solution...\n");  
SUCCESS: return 0;  
/* End of the example code */  
}
```

Example [C-39](#) shows Fortran 90 code for the problem.

Example C-39. Fortran Example to Solve Helmholtz Problem on a Sphere

```
!*****
!
!                               INTEL CONFIDENTIAL
!
!   Copyright(C) 2006-2008 Intel Corporation. All Rights Reserved.
!
!   The source code contained or described herein and all documents related to
!   the source code ("Material") are owned by Intel Corporation or its suppliers
!   or licensors. Title to the Material remains with Intel Corporation or its
!   suppliers and licensors. The Material contains trade secrets and proprietary
!   and confidential information of Intel or its suppliers and licensors. The
!   Material is protected by worldwide copyright and trade secret laws and
!   treaty provisions. No part of the Material may be used, copied, reproduced,
!   modified, published, uploaded, posted, transmitted, distributed or disclosed
!   in any way without Intel's prior express written permission.
!
!   No license under any patent, copyright, trade secret or other intellectual
!   property right is granted to or conferred upon you by disclosure or delivery
!   of the Materials, either expressly, by implication, inducement, estoppel or
!   otherwise. Any license under such intellectual property rights must be
!   express and approved by Intel in writing.
!
!*****
!   Content:
!
!   Fortran-90 precision example of solving Helmholtz problem on a whole sphere
!   using MKL Poisson Library
!
!*****
!
program d_sph_with_poles
! Include modules defined by mkl_poisson.f90 and mkl_dfti.f90 header files
```

```

use mkl_poisson
implicit none

integer np,nt
! Note that the size of the transform np must be even !!!
parameter(np=8,nt=8)
double precision pi
parameter(pi=3.14159265358979324D0)
double precision ap,bp,hp,at,bt,ht,q,lp,lt,theta_i,ct,cl
double precision u(np+1,nt+1),f(np+1,nt+1)
type(DFTI_DESCRIPTOR), pointer :: handle_s, handle_c
integer stat
double precision dpar(5*np/2+nt+10)
integer ip,it,i
integer ipar(128)

! Printing the header for the example
      print *, ''
      print *, ' Example of use of MKL Poisson Library'
      print *, ' *****'
      print *, ''
      print *, ' This example gives the solution of Helmholtz problem on a whole sphere'
      print *, ' 0<p<2*pi, 0<t<pi, with Helmholtz coefficient q=1 and right-hand side'
      print *, ' f(p,t)=3*cos(t)'
      print *, ' -----'
      print *, ' In general, the error should be of order O(1.0/np^2+1.0/nt^2)'
      print '(1x,a,I1)', ' For this example, the value of np=nt is ', np
      print *, ' The approximation error should be of order 0.15D+0, if everything is OK'

```

```

print *, ' -----'
print *, ' Note that np should be even to solve the PERIODIC problem!'
print *, ' -----'
print *, '
                                DOUBLE PRECISION COMPUTATIONS
                                '
print *, ' ====='
print *, ''

```

```

! Defining the rectangular domain on a sphere 0<p<2*pi, 0<t<pi for Helmholtz Solver on a sphere
! Poisson Library will automatically detect that this problem is on a whole sphere!

ap=0.0D0
bp=2*pi
at=0.0D0
bt=pi

! Setting the coefficient q to 1.0D0 for Helmholtz problem
! If you like to solve Poisson problem, please set q to 0.0D0
q=1.0D0

! Computing the mesh size hp in phi-direction
lp=bp-ap
hp=lp/np

! Computing the mesh size ht in theta-direction
lt=bt-at
ht=lt/nt

! Filling in the values of the TRUE solution u(p,t)=cos(t)
! in the mesh points into the array u
! Filling in the right-hand side f(p,t)=3*cos(t)
! in the mesh points into the array f.

! We choose the right-hand side to correspond to the TRUE solution of Helmholtz equation.
! Here we are using the mesh sizes hp and ht computed before to compute

```

```
! the coordinates (phi_i,theta_i) of the mesh points
do it=1,nt+1
  do ip=1,np+1
    theta_i=ht*(it-1)
    ct=dcos(theta_i)
    u(ip,it)=ct
    f(ip,it)=(2.0D0+q)*ct
  enddo
enddo

! Initializing ipar array to make it free from garbage
do i=1,128
  ipar(i)=0
enddo

! Initializing simple data structures of Poisson Library for Helmholtz Solver on a sphere
! As we are looking for the solution on a whole sphere, this is a PERIDOC problem
! Therefore, the routines ending with "_P" are used to find the solution
  call D_INIT_SPH_P(ap,bp,at,bt,np,nt,q,ipar,dpar,stat)
  if (stat.ne.0) goto 999

! Initializing complex data structures of Poisson Library for Helmholtz Solver on a sphere
! NOTE: Right-hand side f may be altered after the Commit step. If you want to keep it,
! you should save it in another memory location!
  call D_COMMIT_SPH_P(f,handle_s,handle_c,ipar,dpar,stat)
  if (stat.ne.0) goto 999

! Computing the approximate solution of Helmholtz problem on a whole sphere
  call D_SPH_P(f,handle_s,handle_c,ipar,dpar,stat)
  if (stat.ne.0) goto 999

! Cleaning the memory used by handle_s and handle_c
```

```
call FREE_SPH_P(handle_s,handle_c,ipar,stat)
if (stat.ne.0) goto 999

! Now we can use handle_s and handle_c to solve another Helmholtz problem
! after a proper initialization

! Printing the results
write(*,10) np
write(*,11) nt
print *, ''

! Watching the error along the line phi=hp
ip=2
c1 = 0.0
do it=1,nt+1
    write(*,12) (ip-1)*hp, (it-1)*ht, f(ip,it)-u(ip,it)
    if (dabs(f(ip,it)-u(ip,it)).ge.c1) c1 = dabs(f(ip,it)-u(ip,it))
enddo
print *, ''
if (c1.ge.0.15D+0) then
    print *, 'The computed solution seems to be inaccurate.'
    goto 999
endif

! Success message to print if everything is OK
    print *, ' Double precision Helmholtz example on a whole sphere has successfully'
    print *, ' PASSED through all steps of computation!'

! Jumping over failure message
    go to 1

! Failure message to print if something went wrong
999 print *, 'Double precision Helmholtz example on a whole sphere FAILED
```

```
      1 to compute the solution...'
1 continue
10    format(1x,'The number of mesh intervals in phi-direction is np=',I1)
11    format(1x,'The number of mesh intervals in theta-direction is nt=',I1)
12    format(1x,'In the mesh point (',F5.3,',',F5.3,') the error between

      1 the computed and the true solution is equal to ', E10.3)
! End of the example code
end
```

Preconditioner Code Examples

This section contains example codes in FORTRAN 77 and C. For description of the preconditioner routines used in these codes, refer to "Preconditioners (Accelerators) based on Incomplete LU Factorization Techniques" in Chapter 8 of the manual.

Fortran Example of Using ILU0 Preconditioner with RCI FGMRES Solver

Example results for non-degenerate system. Upon successful execution of the solver, the output contains the following information:

```
-----
The FULLY ADVANCED example RCI FGMRES with ILU0 preconditioner
to solve the non-degenerate algebraic system of linear equations
-----

Some info about the current run of RCI FGMRES method:

As IPAR(8)=0, the automatic test for the maximal number of iterations will be skipped
+++

As IPAR(9)=0, the automatic residual test will be skipped
+++

As IPAR(10)=1, the user-defined stopping test will be requested via RCI_REQUEST=2
+++

As IPAR(11)=1, the Preconditioned FGMRES iterations will be performed, thus,
the preconditioner action will be requested via RCI_REQUEST=3
+++

As IPAR(12)=0, the automatic test for the norm of the next generated vector is
not equal to zero up to rounding and computational errors will be skipped,
thus, the user-defined test will be requested via RCI_REQUEST=4
+++

The system has been solved

The following solution has been obtained
COMPUTED_SOLUTION(1)= 0.100E+01
COMPUTED_SOLUTION(2)= 0.100E+01
COMPUTED_SOLUTION(3)= 0.100E+01
```



```
COMPUTED_SOLUTION(4)= 0.100E+01
```

The expected solution is:

```
EXPECTED_SOLUTION(1)= 0.100E+01
```

```
EXPECTED_SOLUTION(2)= 0.100E+01
```

```
EXPECTED_SOLUTION(3)= 0.100E+01
```

```
EXPECTED_SOLUTION(4)= 0.100E+01
```

Number of iterations: 2

```
-----  
Fortran example of FGMRES with ILU0 preconditioner  
has successfully PASSED all stages of computations  
-----
```



NOTE. If the ILU0 preconditioner is not used, then the same code requires 7 iterations.

Example C-40. FORTRAN 77 Example to Solve Non-symmetric Non-degenerate Linear System

```
!*****
!
!                               INTEL CONFIDENTIAL
!
!   Copyright(C) 2006-2008 Intel Corporation. All Rights Reserved.
!
!   The source code contained or described herein and all documents related to
!   the source code ("Material") are owned by Intel Corporation or its suppliers
!   or licensors. Title to the Material remains with Intel Corporation or its
!   suppliers and licensors. The Material contains trade secrets and proprietary
!   and confidential information of Intel or its suppliers and licensors. The
!   Material is protected by worldwide copyright and trade secret laws and
!   treaty provisions. No part of the Material may be used, copied, reproduced,
!   modified, published, uploaded, posted, transmitted, distributed or disclosed
!   in any way without Intel's prior express written permission.
!
!   No license under any patent, copyright, trade secret or other intellectual
!   property right is granted to or conferred upon you by disclosure or delivery
!   of the Materials, either expressly, by implication, inducement, estoppel or
!   otherwise. Any license under such intellectual property rights must be
!   express and approved by Intel in writing.
C*****
C
C   Content:
C   Intel MKL example of RCI Flexible Generalized Minimal RESidual method with
C   ILU0 Preconditioner
C*****

C-----
C   Example program for solving non-degenerate system of equations.
C   Full functionality of RCI FGMRES solver is exploited. Example shows how
C   ILU0 preconditioner accelerates the solver by reducing the number of
```

```

C iterations.
C-----

      PROGRAM FGMRES_FULL_FUNCT_F

      IMPLICIT NONE

      INCLUDE "mkl_rci.fi"

      INTEGER N

      PARAMETER(N=4)

      INTEGER SIZE

      PARAMETER (SIZE=128)

C-----
C Define arrays for the upper triangle of the coefficient matrix
C Compressed sparse row storage is used for sparse representation
C-----

      INTEGER IA(5)
      DATA IA /1,4,7,10,13/
      INTEGER JA(12)
      DATA JA /1,2,3,1,2,4,1,3,4,2,3,4/
      DOUBLE PRECISION A(12),BILU0(12),TRVEC(N)
      DATA A / 4.,-1.,-1.,-1.,4.,-1.,-1.,4.,-1.,-1.,-1.,4./

C-----
C Allocate storage for the ?par parameters and the solution/rhs/residual vectors
C-----

      INTEGER IPAR(SIZE),IERR

      DOUBLE PRECISION DPAR(SIZE), TMP(N*(2*N+1)+(N*(N+9))/2+1)

      DOUBLE PRECISION EXPECTED_SOLUTION(N)

```

```
DATA EXPECTED_SOLUTION /1.0,1.0,1.0,1.0/
```

```
DOUBLE PRECISION RHS(N), B(N)
```

```
DOUBLE PRECISION COMPUTED_SOLUTION(N)
```

```
DOUBLE PRECISION RESIDUAL(N)
```

```
INTEGER MATSIZE, INCX, REF_NIT
```

```
DOUBLE PRECISION REF_NORM2, NRM2
```

```
PARAMETER (MATSIZE=12, INCX=1, REF_NIT=2, REF_NORM2=7.772387d0)
```

```
C-----
```

```
C Some additional variables to use with the RCI (P)FGMRES solver
```

```
C-----
```

```
INTEGER ITERCOUNT
```

```
INTEGER RCI_REQUEST, I
```

```
DOUBLE PRECISION DVAR
```

```
C-----
```

```
C An external BLAS function is taken from MKL BLAS to use
```

```
C with the RCI (P)FGMRES solver
```

```
C-----
```

```
DOUBLE PRECISION DNRM2
```

```
EXTERNAL DNRM2
```

```
WRITE( *, '(A,A)' ) '-----',
```

```
1 '-----'
```

```
WRITE(*, '(A,A)' ) 'The FULLY ADVANCED example RCI FGMRES with',
```

```
1 ' ILU0 preconditioner'
```

```
WRITE(*, '(A,A)' ) 'to solve the non-degenerate algebraic system',
```

```
1 ' of linear equations'
```

```
WRITE( *, '(A,A)' ) '-----',
```

```

1 '-----'
C-----
C Initialize variables and the right hand side through matrix-vector product
C-----
      CALL MKL_DCSRGMV('N', N, A, IA, JA, EXPECTED_SOLUTION, RHS)
C-----
C Save the right-hand side in vector B for future use
C-----
      CALL DCOPY(N, RHS, 1, B, 1)
C-----
C Initialize the initial guess
C-----
      DO I=1,N
          COMPUTED_SOLUTION(I)=0.d0
      ENDDO
      COMPUTED_SOLUTION(1)=100.d0
C-----
C Initialize the solver
C-----
      CALL DFGMRES_INIT(N, COMPUTED_SOLUTION, RHS, RCI_REQUEST, IPAR,
1 DPAR, TMP)
      IF (RCI_REQUEST.NE.0) GOTO 999
C-----
C Calculate ILU0 preconditioner.
C
      !ATTENTION!
C DCSRILU0 routine uses some IPAR, DPAR set by DFGMRES_INIT routine.
C Important for DCSRILU0 default entries set by DFGMRES_INIT are

```

```

C ipar(2) = 6 - output of error messages to the screen,
C ipar(6) = 1 - allow output of error messages,
C ipar(31)= 0 - abort DCSRILU0 calculations if routine meets zero diagonal element.
C
C If ILU0 is going to be used out of MKL FGMRES context, than the values
C of ipar(2), ipar(6), ipar(31), dpar(31), and dpar(32) should be user
C provided before the DCSRILU0 routine call.
C
C In this example, specific for DCSRILU0 entries are set in turn:
C ipar(31)= 1 - change small diagonal value to that given by dpar(32),
C dpar(31)= 1.D-20 instead of the default value set by DFGMRES_INIT.
C           It is a small value to compare a diagonal entry with it.
C dpar(32)= 1.D-16 instead of the default value set by DFGMRES_INIT.
C           It is the target value of the diagonal value if it is
C           small as compared to dpar(31) and the routine should change
C           it rather than abort DCSRILU0 calculations.
C-----

```

```

      IPAR(31)=1
      DPAR(31)=1.D-20
      DPAR(32)=1.D-16

      CALL DCSRILU0(N, A, IA, JA, BILU0, IPAR, DPAR, IERR)
      NRM2=DNRM2(MATSIZE, BILU0, INCX)

      IF(IERR.ne.0) THEN
        WRITE(*,'(A,A,I1)') ' Error after calculation of the',
1 ' preconditioner DCSRILU0',IERR
        GOTO 998
      
```

```
ENDIF

C-----
C Set the desired parameters:
C do the restart after 2 iterations
C LOGICAL parameters:
C do not do the stopping test for the maximal number of iterations
C do the Preconditioned iterations of FGMRES method
C Set parameter IPAR(11) for preconditioner call. For this example,
C it reduces the number of iterations.
C DOUBLE PRECISION parameters
C set the relative tolerance to 1.0D-3 instead of default value 1.0D-6
C NOTE. Preconditioner may increase the number of iterations for an
C arbitrary case of the system and initial guess and even ruin the
C convergence. It is user's responsibility to use a suitable preconditioner
C and to apply it skillfully.
C-----

      IPAR(15)=2
      IPAR(8)=0
      IPAR(11)=1
      DPAR(1)=1.0D-3

C-----
C Check the correctness and consistency of the newly set parameters
C-----

      CALL DFGMRES_CHECK(N, COMPUTED_SOLUTION, RHS, RCI_REQUEST,
1 IPAR, DPAR, TMP)
      IF (RCI_REQUEST.NE.0) GOTO 999

C-----
C Print the info about the RCI FGMRES method
```

C-----

```

WRITE( *, ' (A) ' ) ' '
WRITE( *, ' (A,A) ' ) 'Some info about the current run of RCI_FGMRES',
1 ' method:'
WRITE( *, ' (A) ' ) ' '
IF (IPAR(8).NE.0) THEN
    WRITE(*, ' (A,I1,A,A) ' ) 'As IPAR(8)=',IPAR(8),', the automatic',
1 ' test for the maximal number of iterations will be performed'
ELSE
    WRITE(*, ' (A,I1,A,A) ' ) 'As IPAR(8)=',IPAR(8),', the automatic',
1 ' test for the maximal number of iterations will be skipped'
ENDIF
WRITE( *, ' (A) ' ) '+++'
IF (IPAR(9).NE.0) THEN
    WRITE(*, ' (A,I1,A,A) ' ) 'As IPAR(9)=',IPAR(9),', the automatic',
1 ' residual test will be performed'
ELSE
    WRITE(*, ' (A,I1,A,A) ' ) 'As IPAR(9)=',IPAR(9),', the automatic',
1 ' residual test will be skipped'
ENDIF
WRITE( *, ' (A) ' ) '+++'
IF (IPAR(10).NE.0) THEN
    WRITE(*, ' (A,I1,A,A) ' ) 'As IPAR(10)=',IPAR(10),', the',
1 ' user-defined stopping test will be requested via RCI_REQUEST=2'
ELSE
    WRITE(*, ' (A,I1,A,A,A) ' ) 'As IPAR(10)=',IPAR(10),', the',
1 ' user-defined stopping test will not be requested, thus,',
1 ' RCI_REQUEST will not take the value 2'

```



```

ENDIF
WRITE( *, '(A)') '+++
IF (IPAR(11).NE.0) THEN
    WRITE(*, '(A,I1,A,A)') 'As IPAR(11)=',IPAR(11),', the',
1 ' Preconditioned FGMRES iterations will be performed, thus,'
    WRITE(*, '(A,A)') 'the preconditioner action will be requested',
1 ' via RCI_REQUEST=3'
ELSE
    WRITE(*, '(A,I1,A,A)') 'As IPAR(11)=',IPAR(11),', the',
1 ' Preconditioned FGMRES iterations will not be performed,'
    WRITE( *, '(A)') 'thus, RCI_REQUEST will not take the value 3'
ENDIF
WRITE( *, '(A)') '+++
IF (IPAR(12).NE.0) THEN
    WRITE(*, '(A,I1,A,A)') 'As IPAR(12)=',IPAR(12),', the automatic',
1 ' test for the norm of the next generated vector is not'
    WRITE( *, '(A,A)') ' equal to zero up to rounding and',
1 ' computational errors will be performed,'
    WRITE( *, '(A)') 'thus, RCI_REQUEST will not take the value 4'
ELSE
    WRITE(*, '(A,I1,A,A)') 'As IPAR(12)=',IPAR(12),', the automatic',
1 ' test for the norm of the next generated vector is'
    WRITE(*, '(A,A)') 'not equal to zero up to rounding and',
1 ' computational errors will be skipped,'
    WRITE(*, '(A,A)') 'thus, the user-defined test will be requested',
1 ' via RCI_REQUEST=4'
ENDIF
WRITE( *, '(A)') '+++

```

```

C-----
C Compute the solution by RCI (P)FGMRES solver with preconditioning
C Reverse Communication starts here
C-----
1      CALL DFGMRES(N, COMPUTED_SOLUTION, RHS, RCI_REQUEST, IPAR,
          1 DPAR, TMP)
C-----
C If RCI_REQUEST=0, then the solution was found with the required precision
C-----
          IF (RCI_REQUEST.EQ.0) GOTO 3
C-----
C If RCI_REQUEST=1, then compute the vector A*TMP(IPAR(22))
C and put the result in vector TMP(IPAR(23))
C-----
          IF (RCI_REQUEST.EQ.1) THEN
              CALL MKL_DCSRGEMV('N',N, A, IA, JA, TMP(IPAR(22)), TMP(IPAR(23)))
              GOTO 1
          ENDIF
C-----
C If RCI_request=2, then do the user-defined stopping test
C The residual stopping test for the computed solution is performed here
C-----
C NOTE: from this point vector B(N) is no longer containing the right-hand
C side of the problem! It contains the current FGMRES approximation to the
C solution. If you need to keep the right-hand side, save it in some other
C vector before the call to DFGMRES routine. Here we saved it in vector
C RHS(N). The vector B is used instead of RHS to preserve the original
C right-hand side of the problem and guarantee the proper restart of FGMRES

```

```

C method. Vector B will be altered when computing the residual stopping
C criterion!
C-----
      IF (RCI_REQUEST.EQ.2) THEN
C Request to the DFGMRES_GET routine to put the solution into B(N) via IPAR(13)
      IPAR(13)=1
C Get the current FGMRES solution in the vector B(N)
      CALL DFGMRES_GET(N, COMPUTED_SOLUTION, B, RCI_REQUEST, IPAR,
1 DPAR, TMP, ITERCOUNT)
C Compute the current true residual via MKL (Sparse) BLAS routines
      CALL MKL_DCSRGMV('N', N, A, IA, JA, B, RESIDUAL)
      CALL DAXPY(N, -1.0D0, RHS, 1, RESIDUAL, 1)
      DVAR=DNRM2(N, RESIDUAL, 1)
      IF (DVAR.LT.1.0E-3) THEN
          GOTO 3
      ELSE
          GOTO 1
      ENDIF
      ENDIF
C-----
C If RCI_REQUEST=3, then apply the preconditioner on the vector
C TMP(IPAR(22)) and put the result in vector TMP(IPAR(23))
C Here is the recommended usage of the result produced by ILU0 routine
C via standard MKL Sparse Blas solver routine mkl_dcsrtrsv.
C-----
      IF (RCI_REQUEST.EQ.3) THEN
          CALL MKL_DCSRTRSV('L', 'N', 'U', N, BILU0, IA, JA, TMP(IPAR(22)), TRVEC)
          CALL MKL_DCSRTRSV('U', 'N', 'N', N, BILU0, IA, JA, TRVEC, TMP(IPAR(23)))

```

```

        GOTO 1
    ENDIF

C-----
C If RCI_REQUEST=4, then check if the norm of the next generated vector is
C not zero up to rounding and computational errors. The norm is contained
C in DPAR(7) parameter
C-----
        IF (RCI_REQUEST.EQ.4) THEN
            IF (DPAR(7).LT.1.0D-12) THEN
                GOTO 3
            ELSE
                GOTO 1
            ENDIF
        ENDIF

C-----
C If RCI_REQUEST=anything else, then DFGMRES subroutine failed
C to compute the solution vector: COMPUTED_SOLUTION(N)
C-----
        ELSE
            GOTO 999
        ENDIF

C-----
C Reverse Communication ends here
C Get the current iteration number and the FGMRES solution. (DO NOT FORGET to
C call DFGMRES_GET routine as computed_solution is still containing
C the initial guess!). Request to DFGMRES_GET to put the solution into
C vector COMPUTED_SOLUTION(N) via IPAR(13)
C-----
3      IPAR(13)=0

```

```

      CALL DFGMRES_GET(N, COMPUTED_SOLUTION, RHS, RCI_REQUEST, IPAR,
1 DPAR, TMP, ITERCOUNT)

C-----
C Print solution vector: COMPUTED_SOLUTION(N) and
C the number of iterations: ITERCOUNT
C-----

      WRITE( *, '(A)' ) ' '
      WRITE( *, '(A)' ) 'The system has been solved'
      WRITE( *, '(A)' ) ' '
      WRITE( *, '(A)' ) 'The following solution has been obtained:'
      DO I=1,N
         WRITE(*, '(A18,I1,A2,E10.3)') 'COMPUTED_SOLUTION(', I, ')=',
1 COMPUTED_SOLUTION(I)
      ENDDO

      WRITE( *, '(A)' ) ' '
      WRITE( *, '(A)' ) 'The expected solution is:'
      DO I=1,N
         WRITE(*, '(A18,I1,A2,E10.3)') 'EXPECTED_SOLUTION(', I, ')=',
1 EXPECTED_SOLUTION(I)
      ENDDO

      WRITE( *, '(A)' ) ' '
      WRITE( *, '(A,I2)') 'Number of iterations: ', ITERCOUNT
      WRITE( *, '(A)' ) ' '

      IF(ITERCOUNT.EQ.REF_NIT.AND.DABS(REF_NORM2-NRM2).LT.1.D-6) THEN
      WRITE( *, '(A)' ) ' '
      WRITE( *, '(A)' ) '-----'
      WRITE( *, '(A,A)') 'Fortran example of FGMRES with ILU0',

```

```

1 ' preconditioner '
  WRITE( *,'(A,A)') 'has successfully PASSED all stages of',
1 ' computations'
  WRITE( *,'(A)') '-----'
  WRITE( *,'(A)') ' '
  GOTO 1000
  ELSE
  WRITE( *,'(A,A)') 'Probably, the preconditioner was computed',
1 ' incorrectly:'
  WRITE( *,'(A,F9.6,A,F9.6)')
1 'Either the preconditioner norm',NRM2,
2 ' differs from the expected norm',REF_NORM2
  WRITE( *,'(A,I2,A,I2)'),
1 'and/or the number of iterations',ITERCOUNT,
2 ' differs from the expected number ', REF_NIT
  WRITE( *,'(A)') ' '
  WRITE( *,'(A,A)') '-----',
1 '-----'
  WRITE( *,'(A,A)') 'Unfortunately, FGMRES+ILU0 Fortran example',
1 ' has FAILED'
  WRITE( *,'(A,A)') '-----',
1 '-----'
  WRITE( *,'(A)') ' '
  GOTO 1000
  END IF
999 WRITE( *,'(A,I2)') 'The solver has returned the ERROR code ',
1 RCI_REQUEST
998 WRITE( *,'(A)') ' '

```

```

WRITE( *, '(A,A)') '-----',
1 '-----'

WRITE( *, '(A,A)') 'Unfortunately, FGMRES+ILU0 Fortran example',
1 ' has FAILED'

WRITE( *, '(A,A)') '-----',
1 '-----'

WRITE( *, '(A)') ' '

1000 CONTINUE

END

```

C Example of Using ILU0 Preconditioner with RCI FGMRES Solver

C example has the same non-degenerate system as in the previous FORTRAN 77 example. The results are the same up to the notational convention between C and Fortran. Please pay special attention to how it is recommended to handle the differences between the Fortran and C arrays. Specifically, in this example we adjust the addresses for input/result for user-defined operations from `IPAR(22)` to `ipar[21]-1` and from `IPAR(23)` to `ipar[22]-1`, respectively. Specific to ILU0 preconditioner parameter entries are referred to as `ipar[30]`, `dpar[30]`, `dpar[31]`. Upon successful execution of the solver, the following information is printed (up to rounding errors that depend on the computer system):

```

-----
The FULLY ADVANCED example RCI FGMRES with ILU0 preconditioner
to solve the non-degenerate algebraic system of linear equations
-----

```

Some info about the current run of RCI FGMRES method:

As ipar[7]=0, the automatic test for the maximal number of iterations will be skipped

+++

As ipar[8]=0, the automatic residual test will be skipped

+++

As ipar[9]=1 the user-defined stopping test will be requested via

RCI_request=2

+++

As ipar[10]=1, the Preconditioned FGMRES iterations will be performed, thus, the preconditioner action will be requested via RCI_request=3

+++

As ipar[11]=0, the automatic test for the norm of the next generated vector is not equal to zero up to rounding and computational errors will be skipped, thus, the user-defined test will be requested via RCI_request=4

+++

The system has been solved

The following solution has been obtained

computed_solution[0]=1.000000e+00

computed_solution[1]=1.000000e+00

computed_solution[2]=1.000000e+00

computed_solution[3]=1.000000e+00

The expected solution is:

expected_solution[0]=1.000000e+00

expected_solution[1]=1.000000e+00

expected_solution[2]=1.000000e+00

expected_solution[3]=1.000000e+00

Number of iterations: 2

C example of FGMRES with ILU0 preconditioner
has successfully PASSED all stages of computations



NOTE. If the ILU0 preconditioner is not used, then the same code requires 7 iterations.

Example C-41. C Example to Solve Non-symmetric Non-degenerate Linear System

```

/*****
/*
/*          INTEL CONFIDENTIAL
/*
/*   Copyright(C) 2006-2008 Intel Corporation. All Rights Reserved.
/*
/*   The source code contained or described herein and all documents related to
/*   the source code ("Material") are owned by Intel Corporation or its suppliers
/*   or licensors. Title to the Material remains with Intel Corporation or its
/*   suppliers and licensors. The Material contains trade secrets and proprietary
/*   and confidential information of Intel or its suppliers and licensors. The
/*   Material is protected by worldwide copyright and trade secret laws and
/*   treaty provisions. No part of the Material may be used, copied, reproduced,
/*   modified, published, uploaded, posted, transmitted, distributed or disclosed
/*   in any way without Intel's prior express written permission.
/*
/*   No license under any patent, copyright, trade secret or other intellectual
/*   property right is granted to or conferred upon you by disclosure or delivery
/*   of the Materials, either expressly, by implication, inducement, estoppel or
/*   otherwise. Any license under such intellectual property rights must be
/*   express and approved by Intel in writing.
/*
/*
/*****
/*   Content:
/*
/*   Intel MKL example of RCI Flexible Generalized Minimal RESidual method with
/*   ILU0 Preconditioner
/*****

/*-----
/*
/*   Example program for solving non-degenerate system of equations.
/*
/*   Full functionality of RCI FGMRES solver is exploited. Example shows how

```

```

/* ILU0 preconditioner accelerates the solver by reducing the number of
/* iterations.
/*-----*/

#include <stdio.h>
#include "math.h"
#include "mkl_blas.h"
#include "mkl_spblas.h"
#include "mkl_rci.h"
#define N 4
#define size 128
int main(void)
{
/*-----
/* Define arrays for the upper triangle of the coefficient matrix
/* Compressed sparse row storage is used for sparse representation
/*-----*/
    MKL_INT ia[5]={1,4,7,10,13};
    MKL_INT ja[12]={1,2,3,1,2,4,1,3,4,2,3,4};
    double A[12]={4.,-1.,-1.,-1.,4.,-1.,-1.,4.,-1.,-1.,-1.,4.};
/*-----
/* Allocate storage for the ?par parameters and the solution/rhs/residual vectors
/*-----*/
    MKL_INT ipar[size];
    double dpar[size], tmp[N*(2*N+1)+(N*(N+9))/2+1];
    double trvec[N], bilu0[12];
    double expected_solution[N]={1.0,1.0,1.0,1.0};
    double rhs[N], b[N];

```

```

        double computed_solution[N];
        double residual[N];

        MKL_INT matsize=12, incx=1, ref_nit=2;
        double ref_norm2=7.772387E+0, nrm2;

/*-----
/* Some additional variables to use with the RCI (P)FGMRES solver
/*-----*/

        MKL_INT itercount,ierr=0;
        MKL_INT RCI_request, i, ivar;
        double dvar;
        char cvar,cvar1,cvar2;

        printf("-----\n");
        printf("The FULLY ADVANCED example RCI FGMRES with ILU0 preconditioner\n");
        printf("to solve the non-degenerate algebraic system of linear equations\n");
        printf("-----\n\n");

/*-----
/* Initialize variables and the right hand side through matrix-vector product
/*-----*/

        ivar=N;
        cvar='N';
        mkl_dcsrgemv(&cvar, &ivar, A, ia, ja, expected_solution, rhs);

/*-----
/* Save the right-hand side in vector b for future use
/*-----*/

        i=1;
        dcopy(&ivar, rhs, &i, b, &i);

```

```

/*-----
/* Initialize the initial guess
/*-----*/

    for(i=0;i<N;i++)
    {
        computed_solution[i]=0.0;
    }
    computed_solution[0]=100.0;

/*-----
/* Initialize the solver
/*-----*/

    dfgmres_init(&ivar, computed_solution, rhs, &RCI_request, ipar, dpar, tmp);
    if (RCI_request!=0) goto FAILED;

/*-----
/* Calculate ILU0 preconditioner.
/*
/*          !ATTENTION!
/* DCSRILU0 routine uses some IPAR, DPAR set by DFGMRES_INIT routine.
/* Important for DCSRILU0 default entries set by DFGMRES_INIT are
/* ipar[1] = 6 - output of error messages to the screen,
/* ipar[5] = 1 - allow output of errors,
/* ipar[30]= 0 - abort DCSRILU0 calculations if routine meets zero diagonal element.
/*
/* If ILU0 is going to be used out of MKL FGMRES context, than the values
/* of ipar[1], ipar[5], ipar[30], dpar[30], and dpar[31] should be user
/* provided before the DCSRILU0 routine call.
/*

```

```

/* In this example, specific for DCSRILU0 entries are set in turn:
/* ipar[30]= 1 - change small diagonal value to that given by dpar[31],
/* dpar[30]= 1.E-20 instead of the default value set by DFGMRES_INIT.
/*          It is a small value to compare a diagonal entry with it.
/* dpar[31]= 1.E-16 instead of the default value set by DFGMRES_INIT.
/*          It is the target value of the diagonal value if it is
/*          small as compared to dpar[30] and the routine should change
/*          it rather than abort DCSRILU0 calculations.
/*-----*/

        ipar[30]=1;
        dpar[30]=1.E-20;
        dpar[31]=1.E-16;

dcsrilu0(&ivar, A, ia, ja, bilu0, ipar, dpar, &ierr);
nrm2=dhnm2(&matsize, bilu0, &incx );

        if (ierr!=0)
        {
            printf("Preconditioner dcsrilu0 has returned the ERROR code %d", ierr);
            goto FAILED1;
        }

/*-----
/* Set the desired parameters:
/* do the restart after 2 iterations
/* LOGICAL parameters:
/* do not do the stopping test for the maximal number of iterations

```

```

/* do the Preconditioned iterations of FGMRES method
/* Set parameter ipar[10] for preconditioner call. For this example,
/* it reduces the number of iterations.
/* DOUBLE PRECISION parameters
/* set the relative tolerance to 1.0D-3 instead of default value 1.0D-6
/* NOTE. Preconditioner may increase the number of iterations for an
/* arbitrary case of the system and initial guess and even ruin the
/* convergence. It is user's responsibility to use a suitable preconditioner
/* and to apply it skillfully.
/*-----*/

    ipar[14]=2;
    ipar[7]=0;
    ipar[10]=1;
    dpar[0]=1.0E-3;

/*-----
/* Check the correctness and consistency of the newly set parameters
/*-----*/

dfgmres_check(&ivar, computed_solution, rhs, &RCI_request, ipar, dpar, tmp);
if (RCI_request!=0) goto FAILED;

/*-----
/* Print the info about the RCI FGMRES method
/*-----*/

printf("Some info about the current run of RCI FGMRES method:\n\n");
if (ipar[7])
{
    printf("As ipar[7]=%d, the automatic test for the maximal number of iterations
will be\n", ipar[7]);
    printf("performed\n");
}

```

```
        else
        {
            printf("As ipar[7]=%d, the automatic test for the maximal number of iterations
will be\n", ipar[7]);
            printf("skipped\n");
        }

printf("+++\\n");

        if (ipar[8])
        {
            printf("As ipar[8]=%d, the automatic residual test will be performed\\n", ipar[8]);
        }
        else
        {
            printf("As ipar[8]=%d, the automatic residual test will be skipped\\n", ipar[8]);
        }
        printf("+++\\n");
        if (ipar[9])
        {
            printf("As ipar[9]=%d the user-defined stopping test will be requested via\\n",
ipar[9]);
            printf("RCI_request=2\\n");
        }
        else
        {
            printf("As ipar[9]=%d, the user-defined stopping test will not be requested,
thus,\\n", ipar[9]);
            printf("RCI_request will not take the value 2\\n");
        }
        printf("+++\\n");
```

```

        if (ipar[10])
        {
            printf("As ipar[10]=%d, the Preconditioned FGMRES iterations will be performed,
thus,\n", ipar[10]);
            printf("the preconditioner action will be requested via RCI_request=3\n");
        }
        else
        {
            printf("As ipar[10]=%d, the Preconditioned FGMRES iterations will not be
performed,\n", ipar[10]);
            printf("thus, RCI_request will not take the value 3\n");
        }
        printf("+++\\n");
        if (ipar[11])
        {
            printf("As ipar[11]=%d, the automatic test for the norm of the next generated
vector is\\n", ipar[11]);
            printf("not equal to zero up to rounding and computational errors will be
performed,\\n");
            printf("thus, RCI_request will not take the value 4\\n");
        }
        else
        {
            printf("As ipar[11]=%d, the automatic test for the norm of the next generated
vector is\\n", ipar[11]);
            printf("not equal to zero up to rounding and computational errors will be
skipped,\\n");
            printf("thus, the user-defined test will be requested via RCI_request=4\\n");
        }
        printf("+++\\n\\n");
        /*-----

```

```

        /* Compute the solution by RCI (P)FGMRES solver with preconditioning
        /* Reverse Communication starts here
        /*-----*/
ONE:  dfgmres(&ivar, computed_solution, rhs, &RCI_request, ipar, dpar, tmp);
        /*-----
        /* If RCI_request=0, then the solution was found with the required precision
        /*-----*/
        if (RCI_request==0) goto COMPLETE;
        /*-----
        /* If RCI_request=1, then compute the vector A*tmp[ipar[21]-1]
        /* and put the result in vector tmp[ipar[22]-1]
        /*-----
        /* NOTE that ipar[21] and ipar[22] contain FORTRAN style addresses,
        /* therefore, in C code it is required to subtract 1 from them to get C style
        /* addresses.
        /*-----*/
        if (RCI_request==1)
        {
            mkl_dcsrgemv(&cvar, &ivar, A, ia, ja, &tmp[ipar[21]-1], &tmp[ipar[22]-1]);
            goto ONE;
        }
        /*-----
        /* If RCI_request=2, then do the user-defined stopping test
        /* The residual stopping test for the computed solution is performed here
        /*-----
        /* NOTE: from this point vector b[N] is no longer containing the right-hand
        /* side of the problem! It contains the current FGMRES approximation to the
        /* solution. If you need to keep the right-hand side, save it in some other

```

```
/* vector before the call to dfgmres routine. Here we saved it in vector
/* rhs[N]. The vector b is used instead of rhs to preserve the
/* original right-hand side of the problem and guarantee the proper
/* restart of FGMRES method. Vector b will be altered when computing the
/* residual stopping criterion!
/*-----*/
if (RCI_request==2)
{
    /* Request to the dfgmres_get routine to put the solution into b[N]
    /* via ipar[12]
    /*-----
    /* WARNING: beware that the call to dfgmres_get routine with ipar[12]=0
    /* at this stage may destroy the convergence of the FGMRES method,
    /* therefore, only advanced users should exploit this option with care */
    ipar[12]=1;
    /* Get the current FGMRES solution in the vector b[N] */
    dfgmres_get(&ivar, computed_solution, b, &RCI_request, ipar, dpar, tmp,
               &itercount);

    /* Compute the current true residual via MKL (Sparse) BLAS routines */
    mkl_dcsrgemv(&cvar, &ivar, A, ia, ja, b, residual);
    dvar=-1.0E0;
    i=1;
    daxpy(&ivar, &dvar, rhs, &i, residual, &i);
    dvar=dnrm2(&ivar,residual,&i);
    if (dvar<1.0E-3) goto COMPLETE;
```

```

        else goto ONE;
    }
    /*-----
    /* If RCI_request=3, then apply the preconditioner on the vector
    /* tmp[ipar[21]-1] and put the result in vector tmp[ipar[22]-1]
    /*-----
    /* NOTE that ipar[21] and ipar[22] contain FORTRAN style addresses,
    /* therefore, in C code it is required to subtract 1 from them to get C style
    /* addresses
    /* Here is the recommended usage of the result produced by ILU0 routine
    /* via standard MKL Sparse Blas solver routine mkl_dcsrtrsv.
    /*-----*/
    if (RCI_request==3)
    {
        cvar1='L';
        cvar='N';
        cvar2='U';
        mkl_dcsrtrsv(&cvar1,&cvar,&cvar2,&ivar,bilu0,ia,ja,&tmp[ipar[21]-1],trvec);
        cvar1='U';
        cvar='N';
        cvar2='N';
        mkl_dcsrtrsv(&cvar1,&cvar,&cvar2,&ivar,bilu0,ia,ja,trvec,&tmp[ipar[22]-1]);
        goto ONE;
    }
    /*-----
    /* If RCI_request=4, then check if the norm of the next generated vector is
    /* not zero up to rounding and computational errors. The norm is contained
    /* in dpar[6] parameter

```

```

/*-----*/
if (RCI_request==4)
{
if (dpar[6]<1.0E-12) goto COMPLETE;
else goto ONE;
}
/*-----
/* If RCI_request=anything else, then dfgmres subroutine failed
/* to compute the solution vector: computed_solution[N]
/*-----*/
else
{
goto FAILED;
}
/*-----
/* Reverse Communication ends here
/* Get the current iteration number and the FGMRES solution (DO NOT FORGET to
/* call dfgmres_get routine as computed_solution is still containing
/* the initial guess!). Request to dfgmres_get to put the solution
/* into vector computed_solution[N] via ipar[12]
/*-----*/
COMPLETE: ipar[12]=0;
dfgmres_get(&ivar, computed_solution, rhs, &RCI_request, ipar, dpar, tmp,
&itercount);
/*-----
/* Print solution vector: computed_solution[N] and the number
/* of iterations: itercount
/*-----*/

```

```
printf("The system has been solved \n");
printf("\nThe following solution has been obtained: \n");
for (i=0;i<N;i++)
{
printf("computed_solution[%d]=%e\n",i,computed_solution[i]);
}
printf("\nThe expected solution is: \n");
for (i=0;i<N;i++)
{
printf("expected_solution[%d]=%e\n",i,expected_solution[i]);
}
printf("\nNumber of iterations: %d\n",itercount);
printf("\n");

if(itercount==ref_nit && fabs(ref_norm2-nrm2)<1.e-6)
{
printf("-----\n");
printf("C example of FGMRES with ILU0 preconditioner \n");
printf("has successfully PASSED all stages of computations\n");
printf("-----\n");

return 0;
}
else
{
printf("Probably, the preconditioner was computed incorrectly:\n");
printf("Either the preconditioner norm %e differs from the expected norm
%e\n",nrm2,ref_norm2);
```

```

        printf("and/or the number of iterations %d differs from the expected number
        %d\n",itercount,ref_nit);
        printf("-----\n");
        printf("Unfortunately, FGMRES+ILU0 C example has FAILED\n");
        printf("-----\n");
        return 0;
    }

FAILED:

        printf("The solver has returned the ERROR code %d \n", RCI_request);

FAILED1:

        printf("-----\n");
        printf("Unfortunately, FGMRES+ILU0 C example has FAILED\n");
        printf("-----\n");
        return 0;
}

```

Fortran Example of Using ILUT Preconditioner with RCI FGMRES Solver

Example results for non-degenerate system. Upon successful execution of the solver, the output contains the following information:

```

-----
The FULLY ADVANCED example RCI FGMRES with ILUT preconditioner
to solve the non-degenerate algebraic system of linear equations
-----

```

Some info about the current run of RCI FGMRES method:

As IPAR(8)=0, the automatic test for the maximal number of iterations will be skipped

+++

As IPAR(9)=0, the automatic residual test will be skipped

+++

As IPAR(10)=1, the user-defined stopping test will be requested via RCI_REQUEST=2

+++

As IPAR(11)=1, the Preconditioned FGMRES iterations will be performed, thus,
the preconditioner action will be requested via RCI_REQUEST=3

+++

As IPAR(12)=0, the automatic test for the norm of the next generated vector is
not equal to zero up to rounding and computational errors will be skipped,
thus, the user-defined test will be requested via RCI_REQUEST=4

+++

The system has been solved

The following solution has been obtained:

COMPUTED_SOLUTION(1)= 0.1000020E+01

COMPUTED_SOLUTION(2)= 0.1000007E+01

COMPUTED_SOLUTION(3)= 0.1000006E+01

COMPUTED_SOLUTION(4)= 0.9999908E+00

The expected solution is:

EXPECTED_SOLUTION(1)= 0.1000000E+01

EXPECTED_SOLUTION(2)= 0.1000000E+01

EXPECTED_SOLUTION(3)= 0.1000000E+01

EXPECTED_SOLUTION(4)= 0.1000000E+01

Number of iterations: 4

Fortran example of FGMRES with ILUT preconditioner
has successfully PASSED all stages of computations



NOTE. If the ILUT preconditioner is not used, then the same code requires 10 iterations.

Example C-42. Fortran Example to Solve Non-symmetric Non-degenerate Linear System

```
!*****
!
!                               INTEL CONFIDENTIAL
!
!   Copyright(C) 2007-2008 Intel Corporation. All Rights Reserved.
!
!   The source code contained or described herein and all documents related to
!   the source code ("Material") are owned by Intel Corporation or its suppliers
!   or licensors. Title to the Material remains with Intel Corporation or its
!   suppliers and licensors. The Material contains trade secrets and proprietary
!   and confidential information of Intel or its suppliers and licensors. The
!   Material is protected by worldwide copyright and trade secret laws and
!   treaty provisions. No part of the Material may be used, copied, reproduced,
!   modified, published, uploaded, posted, transmitted, distributed or disclosed
!   in any way without Intel's prior express written permission.
!
!   No license under any patent, copyright, trade secret or other intellectual
!   property right is granted to or conferred upon you by disclosure or delivery
!   of the Materials, either expressly, by implication, inducement, estoppel or
!   otherwise. Any license under such intellectual property rights must be
!   express and approved by Intel in writing.
!*****
C*****
C Content:
C Intel MKL example of RCI Flexible Generalized Minimal RESidual method with
C ILUT Preconditioner
C*****
```

```

C-----
C  Example program for solving non-degenerate system of equations.
C  Full functionality of RCI FGMRES solver is exploited. Example shows how
C  ILUT preconditioner accelerates the solver by reducing the number of
C  iterations.
C-----

      PROGRAM FGMRES_FULL_FUNC_F

      IMPLICIT NONE

      INCLUDE "mkl_rci.fi"

      INTEGER N
      PARAMETER(N=4)

      INTEGER SIZE
      PARAMETER (SIZE=128)

C-----
C  Define arrays for the upper triangle of the coefficient matrix
C  Compressed sparse row storage is used for sparse representation
C-----

      INTEGER IA(5),IBILUT(5)
      DATA IA /1,4,7,10,13/
      INTEGER JA(12),JBILUT(16)
      DATA JA /1,2,3,1,2,4,1,3,4,2,3,4/
      DOUBLE PRECISION A(12),BILUT(16),TRVEC(N)
      DATA A / 4.,-1.,-1.,-1.,4.,-1.,-1.,4.,-1.,-1.,-1.,4./

C-----

```

C Allocate storage for the ?par parameters and the solution/rhs/residual vectors

C-----

```

    INTEGER IPAR(SIZE), IERR
    DOUBLE PRECISION DPAR(SIZE), TMP(N*(2*N+1)+(N*(N+9))/2+1)
    DOUBLE PRECISION EXPECTED_SOLUTION(N)
    DATA EXPECTED_SOLUTION /1.0,1.0,1.0,1.0/
    DOUBLE PRECISION RHS(N), B(N)
    DOUBLE PRECISION COMPUTED_SOLUTION(N)
    DOUBLE PRECISION RESIDUAL(N)

    INTEGER MATSIZE, INCX, REF_NIT
    DOUBLE PRECISION REF_NORM2, NRM2
    PARAMETER (MATSIZE=10, INCX=1, REF_NIT=4, REF_NORM2=7.836719d0)

```

C-----

C Some additional variables to use with the RCI (P)FGMRES solver

C-----

```

    INTEGER ITERCOUNT
    INTEGER RCI_REQUEST, I
    DOUBLE PRECISION DVAR

```

C-----

C An external BLAS function is taken from MKL BLAS to use

C with the RCI (P)FGMRES solver

C-----

```

    DOUBLE PRECISION DNRM2
    EXTERNAL DNRM2

```

C-----

C Some additional variables to use for ILUT preconditioner call

C-----

```

      INTEGER MAXFIL
      DOUBLE PRECISION TOL

      WRITE( *, '(A,A)') '-----',
1 '-----'
      WRITE(*, '(A,A)') 'The FULLY ADVANCED example RCI FGMRES with',
1 ' ILUT preconditioner'
      WRITE(*, '(A,A)') 'to solve the non-degenerate algebraic system',
1 ' of linear equations'
      WRITE( *, '(A,A)') '-----',
1 '-----'

C-----
C Initialize variables and the right hand side through matrix-vector product
C-----
      CALL MKL_DCSRGMV('N', N, A, IA, JA, EXPECTED_SOLUTION, RHS)
C-----
C Save the right-hand side in vector B for future use
C-----
      CALL DCOPY(N, RHS, 1, B, 1)
C-----
C Initialize the initial guess
C-----
      DO I=1,N
          COMPUTED_SOLUTION(I)=0.d0
      ENDDO
      COMPUTED_SOLUTION(1)=100.d0

```

```

C-----
C Initialize the solver
C-----

      CALL DFGMRES_INIT(N, COMPUTED_SOLUTION, RHS, RCI_REQUEST, IPAR,
1 DPAR, TMP)
      IF (RCI_REQUEST.NE.0) GOTO 999

C-----
C Calculate ILUT preconditioner.
C
C           !ATTENTION!
C DCSRILUT routine uses some IPAR, DPAR set by DFGMRES_INIT routine.
C Important for DCSRILUT default entries set by DFGMRES_INIT are
C ipar(2) = 6 - output of error messages to the screen,
C ipar(6) = 1 - allow output of error messages,
C ipar(31)= 0 - abort DCSRILUT calculations if routine meets zero diagonal element.
C ipar(7) = 1 - output warn messages if any and continue
C
C If ILUT is going to be used out of MKL FGMRES context, than the values
C of ipar(2), ipar(6), ipar(31), and dpar(31), should be user
C provided before the DCSRILUT routine call.
C
C In this example, specific for DCSRILUT entries are set in turn:
C ipar(31)= 1 - change small diagonal value to that given by dpar(31),
C dpar(31)= 1.D-5 instead of the default value set by DFGMRES_INIT.
C
C           It is the target value of the diagonal value if it is
C           small as compared to given tolerance multiplied
C           by the matrix row norm and the routine should
C           change it rather than abort DCSRILUT calculations.

```

```
C-----

      IPAR(31)=1
      DPAR(31)=1.D-5
      TOL=1.d-6
      MAXFIL=1

      CALL DCSRILUT(N, A, IA, JA, BILUT, IBILUT, JBILUT,
&                TOL, MAXFIL, IPAR, DPAR, IERR)
      NRM2=DNRM2(MATSIZE, BILUT, INCX)

      IF(IERR.ne.0) THEN
        WRITE(*,'(A,A,I1)') ' Error after calculation of the',
1' preconditioner DCSRILUT',IERR
        GOTO 998
      ENDIF

C-----

C Set the desired parameters:
C do the restart after 2 iterations
C LOGICAL parameters:
C do not do the stopping test for the maximal number of iterations
C do the Preconditioned iterations of FGMRES method
C Set parameter IPAR(11) for preconditioner call.
C For this example, it reduces the number of iterations.
C DOUBLE PRECISION parameters
C set the relative tolerance to 1.0D-3 instead of default value 1.0D-6
C NOTE. Preconditioner may increase the number of iterations for an
```

C arbitrary case of the system and initial guess and even ruin the
C convergence. It is user's responsibility to use a suitable preconditioner
C and to apply it skillfully.

C-----

```

      IPAR(15)=2
      IPAR(8)=0
      IPAR(11)=1
      DPAR(1)=1.0D-3

```

C-----

C Check the correctness and consistency of the newly set parameters

C-----

```

      CALL DFGMRES_CHECK(N, COMPUTED_SOLUTION, RHS, RCI_REQUEST,
1 IPAR, DPAR, TMP)
      IF (RCI_REQUEST.NE.0) GOTO 999

```

C-----

C Print the info about the RCI FGMRES method

C-----

```

      WRITE( *, '(A)' ) ' '
      WRITE( *, '(A,A)' ) 'Some info about the current run of RCI FGMRES',
1 ' method:'
      WRITE( *, '(A)' ) ' '
      IF (IPAR(8).NE.0) THEN
         WRITE(*, '(A,I1,A,A)' ) 'As IPAR(8)=',IPAR(8),', the automatic',
1 ' test for the maximal number of iterations will be performed'
      ELSE
         WRITE(*, '(A,I1,A,A)' ) 'As IPAR(8)=',IPAR(8),', the automatic',
1 ' test for the maximal number of iterations will be skipped'
      ENDIF

```



```

WRITE( *, '(A)') '+++'
IF (IPAR(9).NE.0) THEN
    WRITE(*, '(A,I1,A,A)') 'As IPAR(9)=',IPAR(9),', the automatic',
1 ' residual test will be performed'
ELSE
    WRITE(*, '(A,I1,A,A)') 'As IPAR(9)=',IPAR(9),', the automatic',
1 ' residual test will be skipped'
ENDIF
WRITE( *, '(A)') '+++'
IF (IPAR(10).NE.0) THEN
    WRITE(*, '(A,I1,A,A)') 'As IPAR(10)=',IPAR(10),', the',
1 ' user-defined stopping test will be requested via RCI_REQUEST=2'
ELSE
    WRITE(*, '(A,I1,A,A,A)') 'As IPAR(10)=',IPAR(10),', the',
1 ' user-defined stopping test will not be requested, thus,',
1 ' RCI_REQUEST will not take the value 2'
ENDIF
WRITE( *, '(A)') '+++'
IF (IPAR(11).NE.0) THEN
    WRITE(*, '(A,I1,A,A)') 'As IPAR(11)=',IPAR(11),', the',
1 ' Preconditioned FGMRES iterations will be performed, thus,'
    WRITE(*, '(A,A)') 'the preconditioner action will be requested',
1 ' via RCI_REQUEST=3'
ELSE
    WRITE(*, '(A,I1,A,A)') 'As IPAR(11)=',IPAR(11),', the',
1 ' Preconditioned FGMRES iterations will not be performed,'
    WRITE( *, '(A)') 'thus, RCI_REQUEST will not take the value 3'
ENDIF

```

```

WRITE( *, ' (A) ' ) '+++'
IF (IPAR(12).NE.0) THEN
    WRITE(*, ' (A,I1,A,A) ') 'As IPAR(12)=',IPAR(12),' , the automatic',
1 ' test for the norm of the next generated vector is not'
    WRITE( *, ' (A,A) ') ' equal to zero up to rounding and',
1 ' computational errors will be performed,'
    WRITE( *, ' (A) ') 'thus, RCI_REQUEST will not take the value 4'
ELSE
    WRITE(*, ' (A,I1,A,A) ') 'As IPAR(12)=',IPAR(12),' , the automatic',
1 ' test for the norm of the next generated vector is'
    WRITE(*, ' (A,A) ') 'not equal to zero up to rounding and',
1 ' computational errors will be skipped,'
    WRITE(*, ' (A,A) ') 'thus, the user-defined test will be requested',
1 ' via RCI_REQUEST=4'
ENDIF
WRITE( *, ' (A) ' ) '+++'

```

C-----

C Compute the solution by RCI (P)FGMRES solver with preconditioning

C Reverse Communication starts here

C-----

```

1    CALL DFGMRES(N, COMPUTED_SOLUTION, RHS, RCI_REQUEST, IPAR,
1    DPAR, TMP)

```

C-----

C If RCI_REQUEST=0, then the solution was found with
the required precision

C-----

```

    IF (RCI_REQUEST.EQ.0) GOTO 3

```

C-----

C If RCI_REQUEST=1, then compute the vector A*TMP(IPAR(22))

```

C and put the result in vector TMP(IPAR(23))
C-----
      IF (RCI_REQUEST.EQ.1) THEN
          CALL MKL_DCSRGMV('N',N, A, IA, JA, TMP(IPAR(22)), TMP(IPAR(23)))
          GOTO 1
      ENDIF
C-----
C If RCI_request=2, then do the user-defined stopping test
C The residual stopping test for the computed solution is performed here
C-----
C NOTE: from this point vector B(N) is no longer containing the right-hand
C side of the problem! It contains the current FGMRES approximation to the
C solution. If you need to keep the right-hand side, save it in some other
C vector before the call to DFGMRES routine. Here we saved it in vector
C RHS(N). The vector B is used instead of RHS to preserve the original
C right-hand side of the problem and guarantee the proper restart of FGMRES
C method. Vector B will be altered when computing the residual stopping
C criterion!
C-----
      IF (RCI_REQUEST.EQ.2) THEN
C Request to the DFGMRES_GET routine to put the solution into B(N) via IPAR(13)
          IPAR(13)=1
C Get the current FGMRES solution in the vector B(N)
          CALL DFGMRES_GET(N, COMPUTED_SOLUTION, B, RCI_REQUEST, IPAR,
              1 DPAR, TMP, ITERCOUNT)
C Compute the current true residual via MKL (Sparse) BLAS routines
          CALL MKL_DCSRGMV('N', N, A, IA, JA, B, RESIDUAL)
          CALL DAXPY(N, -1.0D0, RHS, 1, RESIDUAL, 1)

```

```

        DVAR=DNRM2(N, RESIDUAL, 1)
        IF (DVAR.LT.1.0E-3) THEN
            GOTO 3
        ELSE
            GOTO 1
        ENDIF
    ENDIF

C-----
C If RCI_REQUEST=3, then apply the preconditioner on the vector
C TMP(IPAR(22)) and put the result in vector TMP(IPAR(23))
C Here is the recommended usage of the result produced by ILUT routine
C via standard MKL Sparse Blas solver routine mkl_dcsrtrsv.
C-----
        IF (RCI_REQUEST.EQ.3) THEN
            CALL MKL_DCSRTRSV('L','N','U',N,BILUT,IBILUT,JBILUT,
&
                        TMP(IPAR(22)),TRVEC)
            CALL MKL_DCSRTRSV('U','N','N',N,BILUT,IBILUT,JBILUT,
&
                        TRVEC,TMP(IPAR(23)))
            GOTO 1
        ENDIF

C-----
C If RCI_REQUEST=4, then check if the norm of the next generated vector is
C not zero up to rounding and computational errors. The norm is contained
C in DPAR(7) parameter
C-----
        IF (RCI_REQUEST.EQ.4) THEN
            IF (DPAR(7).LT.1.0D-12) THEN
                GOTO 3
            
```

```

        ELSE
            GOTO 1
        ENDIF
C-----
C If RCI_REQUEST=anything else, then DFGMRES subroutine failed
C to compute the solution vector: COMPUTED_SOLUTION(N)
C-----
        ELSE
            GOTO 999
        ENDIF
C-----
C Reverse Communication ends here
C Get the current iteration number and the FGMRES solution. (DO NOT FORGET to
C call DFGMRES_GET routine as computed_solution is still containing
C the initial guess!). Request to DFGMRES_GET to put the solution into
C vector COMPUTED_SOLUTION(N) via IPAR(13)
C-----
3      IPAR(13)=0
        CALL DFGMRES_GET(N, COMPUTED_SOLUTION, RHS, RCI_REQUEST, IPAR,
1 DPAR, TMP, ITERCOUNT)
C-----
C Print solution vector: COMPUTED_SOLUTION(N) and
C the number of iterations: ITERCOUNT
C-----
        WRITE( *, '(A)' ) ' '
        WRITE( *, '(A)' ) 'The system has been solved'
        WRITE( *, '(A)' ) ' '
        WRITE( *, '(A)' ) 'The following solution has been obtained:'

```

```

DO I=1,N
    WRITE(*,'(A18,I1,A2,E14.7)') 'COMPUTED_SOLUTION(',I,')=',
1  COMPUTED_SOLUTION(I)
ENDDO

WRITE(*,'(A)') ' '
WRITE(*,'(A)') 'The expected solution is:'

DO I=1,N

WRITE(*,'(A18,I1,A2,E14.7)') 'EXPECTED_SOLUTION(',I,')=',
1  EXPECTED_SOLUTION(I)
ENDDO

WRITE(*,'(A)') ' '
WRITE(*,'(A,I2)') 'Number of iterations: ',ITERCOUNT
WRITE(*,'(A)') ' '

IF(ITERCOUNT.EQ.REF_NIT.AND.DABS(REF_NORM2-NRM2).LT.1.D-6) THEN
WRITE(*,'(A)') ' '
WRITE(*,'(A)') '-----'
WRITE(*,'(A,A)') 'Fortran example of FGMRES with ILUT',
1 ' preconditioner '
WRITE(*,'(A,A)') 'has successfully PASSED all stages of',
1 ' computations'
WRITE(*,'(A)') '-----'
WRITE(*,'(A)') ' '
GOTO 1000
ELSE
WRITE(*,'(A,A)') 'Probably, the preconditioner was computed',
1 ' incorrectly:'
WRITE(*,'(A,F9.6,A,F9.6)')

```

```

1 'Either the preconditioner norm',NRM2,
2 ' differs from the expected norm',REF_NORM2
   WRITE( *,'(A,I2,A,I2)'),
1 'and/or the number of iterations ', ITERCOUNT,
2 ' differs from the expected number ', REF_NIT
   WRITE( *,'(A)') ' '
   WRITE( *,'(A,A)') '-----',
1 '-----'
   WRITE( *,'(A,A)') 'Unfortunately, FGMRES+ILUT Fortran example',
1 ' has FAILED'
   WRITE( *,'(A,A)') '-----',
1 '-----'
   WRITE( *,'(A)') ' '
   GOTO 1000
   END IF
999  WRITE( *,'(A,I2)') 'The solver has returned the ERROR code ',
1 RCI_REQUEST
998  WRITE( *,'(A)') ' '
   WRITE( *,'(A,A)') '-----',
1 '-----'
   WRITE( *,'(A,A)') 'Unfortunately, FGMRES+ILUT Fortran example',
1 ' has FAILED'
   WRITE( *,'(A,A)') '-----',
1 '-----'
   WRITE( *,'(A)') ' '

1000 CONTINUE
   END

```

C Example of Using ILUT Preconditioner with RCI FGMRES Solver

C example has the same non-degenerate system as in the previous FORTRAN 77 example. The results are the same up to the notational convention between C and Fortran. Please pay special attention to how it is recommended to handle the differences between the Fortran and C arrays. Specifically, in this example we adjust the addresses for input/result for user-defined operations from `IPAR(22)` to `ipar[21]-1` and from `IPAR(23)` to `ipar[22]-1`, respectively. Specific to

ILUT preconditioner parameter entries are referred to as *ipar*[30], *dpar*[30], *dpar*[31]. Upon successful execution of the solver, the following information is printed (up to rounding errors that depend on the computer system):

```
-----  
The FULLY ADVANCED example RCI FGMRES with ILUT  
preconditioner to solve  
the non-degenerate algebraic system of linear equations  
-----
```

Some info about the current run of RCI FGMRES method:

As *ipar*[7]=0, the automatic test for the maximal number of iterations will be
skipped

+++

As *ipar*[8]=0, the automatic residual test will be skipped

+++

As *ipar*[9]=1, the user-defined stopping test will be requested via

RCI_request=2

+++

As *ipar*[10]=1, the Preconditioned FGMRES iterations will be performed, thus,
the preconditioner action will be requested via *RCI_request*=3

+++

As *ipar*[11]=0, the automatic test for the norm of the next generated vector is
not equal to zero up to rounding and computational errors will be skipped,
thus, the user-defined test will be requested via *RCI_request*=4

+++

The system has been solved

The following solution has been obtained:

```
computed_solution[0]=1.000020e+00
computed_solution[1]=1.000007e+00
computed_solution[2]=1.000006e+00
computed_solution[3]=9.99908e-01
```

The expected solution is:

```
expected_solution[0]=1.000000e+00
expected_solution[1]=1.000000e+00
expected_solution[2]=1.000000e+00
expected_solution[3]=1.000000e+00
```

Number of iterations: 4

```
-----
C example of FGMRES with ILUT preconditioner
has successfully PASSED all stages of computations
-----
```



NOTE. If the ILUT preconditioner is not used, then the same code requires 10 iterations.

Example C-43. C Example to Solve Non-symmetric Non-degenerate Linear System

```

/*****
/*
/*          INTEL CONFIDENTIAL
/*
/*  Copyright(C) 2007-2008 Intel Corporation. All Rights Reserved.
/*
/*  The source code contained or described herein and all documents related to
/*  the source code ("Material") are owned by Intel Corporation or its suppliers
/*  or licensors. Title to the Material remains with Intel Corporation or its
/*  suppliers and licensors. The Material contains trade secrets and proprietary
/*  and confidential information of Intel or its suppliers and licensors. The
/*  Material is protected by worldwide copyright and trade secret laws and
/*  treaty provisions. No part of the Material may be used, copied, reproduced,
/*  modified, published, uploaded, posted, transmitted, distributed or disclosed
/*  in any way without Intel's prior express written permission.
/*
/*  No license under any patent, copyright, trade secret or other intellectual
/*  property right is granted to or conferred upon you by disclosure or delivery
/*  of the Materials, either expressly, by implication, inducement, estoppel or
/*  otherwise. Any license under such intellectual property rights must be
/*  express and approved by Intel in writing.
/*
/*
/*****
/*  Content:
/*
/*  Intel MKL example of RCI Flexible Generalized Minimal RESidual method with
/*  ILUT Preconditioner
/*****

/*-----
/*
/*  Example program for solving non-degenerate system of equations.
/*
/*  Full functionality of RCI FGMRES solver is exploited. Example shows how

```

```

/* ILUT preconditioner accelerates the solver by reducing the number of
/* iterations.
/*-----*/

#include <stdio.h>
#include "math.h"
#include "mkl_blas.h"
#include "mkl_spblas.h"
#include "mkl_rci.h"
#define N 4
#define size 128

int main(void)
{
/*-----
/* Define arrays for the upper triangle of the coefficient matrix
/* Compressed sparse row storage is used for sparse representation
/*-----*/

    MKL_INT ia[5]={1,4,7,10,13}, ibilut[5];
    MKL_INT ja[12]={1,2,3,1,2,4,1,3,4,2,3,4}, jbilut[16];
    double A[12]={4.,-1.,-1.,-1.,4.,-1.,-1.,4.,-1.,-1.,-1.,4.};

/*-----
/* Allocate storage for the ?par parameters and the solution/rhs/residual vectors
/*-----*/

    MKL_INT ipar[size];
    double dpar[size], tmp[N*(2*N+1)+(N*(N+9))/2+1];
    double trvec[N], bilut[12];
    double expected_solution[N]={1.0,1.0,1.0,1.0};

```

```

        double rhs[N], b[N];
        double computed_solution[N];
        double residual[N];

/*-----
/* Some additional variables to use with the RCI (P)FGMRES solver
/*-----*/

        MKL_INT itercount,ierr=0;
        MKL_INT RCI_request, i, ivar;
        double dvar;
        char cvar,cvar1,cvar2;

        MKL_INT matsize=10, incx=1, ref_nit=4;
        double ref_norm2=7.836719E+0, nrm2;

/*-----
/* Some additional variables to use for ILUT preconditioner call
/*-----*/

        MKL_INT maxfil;
        double tol;

        printf("-----\n");
        printf("The FULLY ADVANCED example RCI FGMRES with ILUT\n");
        printf("preconditioner to solve \n");
        printf("the non-degenerate algebraic system of linear equations\n");
        printf("-----\n\n");

/*-----
/* Initialize variables and the right hand side through matrix-vector product
/*-----*/

        ivar=N;
```

```

        cvar='N';
        mkl_dcsrgemv(&cvar, &ivar, A, ia, ja, expected_solution, rhs);
/*-----
/* Save the right-hand side in vector b for future use
/*-----*/

        i=1;
        dcopy(&ivar, rhs, &i, b, &i);
/*-----
/* Initialize the initial guess
/*-----*/

        for(i=0;i<N;i++)
        {
                computed_solution[i]=0.0;
        }
        computed_solution[0]=100.0;

/*-----
/* Initialize the solver
/*-----*/

        dfgmres_init(&ivar, computed_solution, rhs, &RCI_request, ipar, dpar, tmp);
        if (RCI_request!=0) goto FAILED;

/*-----
/* Calculate ILUT preconditioner.
/*
/*          !ATTENTION!
/* DCSRILUT routine uses some IPAR, DPAR set by DFGMRES_INIT routine.
/* Important for DCSRILUT default entries set by DFGMRES_INIT are
/* ipar[1] = 6 - output of error messages to the screen,

```

```

/* ipar[5] = 1 - allow output of errors,
/* ipar[6] = 1 - output warn messages if any and continue
/* ipar[30]= 0 - abort DCSRILUT calculations if routine meets zero diagonal element.
/*
/* If ILUT is going to be used out of MKL FGMRES context, than the values
/* of ipar[1], ipar[5], ipar[6], ipar[30], and dpar[30] should be user
/* provided before the DCSRILUT routine call.
/*
/* In this example, specific for DCSRILUT entries are set in turn:
/* ipar[30]= 1 - change small diagonal value to that given by dpar[31],
/* dpar[30]= 1.E-5 instead of the default value set by DFGMRES_INIT.
/*
/*           It is the target value of the diagonal value if it is
/*           small as compared to the given tolerance multiplied
/*           by the matrix row norm and the routine should
/*           change it rather than abort DCSRILUT calculations.
/*-----*/

    ipar[30]=1;
    dpar[30]=1.E-5;
    tol=1.E-6;
    maxfil=1;

    dcsrilit(&ivar, A, ia, ja, bilut, ibilut, jbilut, &tol, &maxfil, ipar, dpar, &ierr);
    nrm2=dnrm2(&matsize, bilut, &incx );

    if (ierr!=0)
    {
        printf("Preconditioner dcsrilit has returned the ERROR code %d", ierr);
        goto FAILED1;

```

```

    }

    /*-----
    /* Set the desired parameters:
    /* do the restart after 2 iterations
    /* LOGICAL parameters:
    /* do not do the stopping test for the maximal number of iterations
    /* do the Preconditioned iterations of FGMRES method
    /* Set parameter ipar[10] for preconditioner call. For this example,
    /* it reduces the number of iterations.
    /* DOUBLE PRECISION parameters
    /* set the relative tolerance to 1.0D-3 instead of default value 1.0D-6
    /* NOTE. Preconditioner may increase the number of iterations for an
    /* arbitrary case of the system and initial guess and even ruin the
    /* convergence. It is user's responsibility to use a suitable preconditioner
    /* and to apply it skillfully.
    /*-----*/

    ipar[14]=2;
    ipar[7]=0;
    ipar[10]=1;
    dpar[0]=1.0E-3;

    /*-----
    /* Check the correctness and consistency of the newly set parameters
    /*-----*/

    dfgmres_check(&ivar, computed_solution, rhs, &RCI_request, ipar, dpar, tmp);
    if (RCI_request!=0) goto FAILED;

    /*-----
    /* Print the info about the RCI FGMRES method

```



```
/*-----*/
printf("Some info about the current run of RCI FGMRES method:\n\n");
if (ipar[7])
{
    printf("As ipar[7]=%d, the automatic test for the maximal number of
        iterations will be\n", ipar[7]);
    printf("performed\n");
}
else
{
    printf("As ipar[7]=%d, the automatic test for the maximal number of
        iterations will be\n", ipar[7]);
    printf("skipped\n");
}
printf("+++\\n");
if (ipar[8])
{
    printf("As ipar[8]=%d, the automatic residual test will be
        performed\\n", ipar[8]);
}
else
{
    printf("As ipar[8]=%d, the automatic residual test will be skipped\\n",
        ipar[8]);
}
printf("+++\\n");
if (ipar[9])
{
```

```
        printf("As ipar[9]=%d, the user-defined stopping test will be requested
              via\n", ipar[9]);
        printf("RCI_request=2\n");
    }
    else
    {
        printf("As ipar[9]=%d, the user-defined stopping test will not be
              requested, thus,\n", ipar[9]);
        printf("RCI_request will not take the value 2\n");
    }
    printf("+++\\n");
    if (ipar[10])
    {
        printf("As ipar[10]=%d, the Preconditioned FGMRES iterations will
              be performed, thus,\n", ipar[10]);
        printf("the preconditioner action will be requested via
              RCI_request=3\\n");
    }
    else
    {
        printf("As ipar[10]=%d, the Preconditioned FGMRES iterations will
              not be performed,\n", ipar[10]);
        printf("thus, RCI_request will not take the value 3\\n");
    }
    printf("+++\\n");
    if (ipar[11])
    {
        printf("As ipar[11]=%d, the automatic test for the norm of the next
```

```

        generated vector is\n", ipar[11]);
printf("not equal to zero up to rounding and computational errors
        will be performed,\n");
printf("thus, RCI_request will not take the value 4\n");
    }
else
{
    printf("As ipar[11]=%d, the automatic test for the norm of the next
        generated vector is\n", ipar[11]);
    printf("not equal to zero up to rounding and computational errors
        will be skipped,\n");
    printf("thus, the user-defined test will be requested via
        RCI_request=4\n");
}
printf("+++\n\n");
/*-----
/* Compute the solution by RCI (P)FGMRES solver with preconditioning
/* Reverse Communication starts here
/*-----*/
ONE: dfgmres(&ivar, computed_solution, rhs, &RCI_request, ipar, dpar, tmp);
/*-----
/* If RCI_request=0, then the solution was found with the required precision
/*-----*/
if (RCI_request==0) goto COMPLETE;
/*-----
/* If RCI_request=1, then compute the vector A*tmp[ipar[21]-1]
/* and put the result in vector tmp[ipar[22]-1]
/*-----

```

```

/* NOTE that ipar[21] and ipar[22] contain FORTRAN style addresses,
/* therefore, in C code it is required to subtract 1 from them to get C style
/* addresses
/*-----*/
if (RCI_request==1)
{
    mkl_dcsrgemv(&cvar, &ivar, A, ia, ja, &tmp[ipar[21]-1],
                &tmp[ipar[22]-1]);
    goto ONE;
}
/*-----
/* If RCI_request=2, then do the user-defined stopping test
/* The residual stopping test for the computed solution is performed here
/*-----
/* NOTE: from this point vector b[N] is no longer containing the right-hand
/* side of the problem! It contains the current FGMRES approximation to the
/* solution. If you need to keep the right-hand side, save it in some other
/* vector before the call to dfgmres routine. Here we saved it in vector
/* rhs[N]. The vector b is used instead of rhs to preserve the
/* original right-hand side of the problem and guarantee the proper
/* restart of FGMRES method. Vector b will be altered when computing the
/* residual stopping criterion!
/*-----*/
if (RCI_request==2)
{
    /* Request to the dfgmres_get routine to put the solution
    /* into b[N] via ipar[12]

/*-----

```

```

/* WARNING: beware that the call to dfgmres_get routine with ipar[12]=0
/* at this stage may
/* destroy the convergence of the FGMRES method, therefore, only
/* advanced users should
/* exploit this option with care */
ipar[12]=1;
/* Get the current FGMRES solution in the vector b[N] */
dfgmres_get(&ivar, computed_solution, b, &RCI_request, ipar, dpar,
tmp, &itercount);
/* Compute the current true residual via MKL (Sparse)
/* BLAS routines */
mkl_dcsrgemv(&cvar, &ivar, A, ia, ja, b, residual);
dvar=-1.0E0;
i=1;
daxpy(&ivar, &dvar, rhs, &i, residual, &i);
dvar=dnrm2(&ivar,residual,&i);
if (dvar<1.0E-3) goto COMPLETE;
else goto ONE;
}
/*-----
/* If RCI_request=3, then apply the preconditioner on the vector
/* tmp[ipar[21]-1] and put the result in vector tmp[ipar[22]-1]
/*-----
/* NOTE that ipar[21] and ipar[22] contain FORTRAN style addresses,
/* therefore, in C code it is required to subtract 1 from them to get C style
/* addresses
/* Here is the recommended usage of the result produced by ILUT routine
/* via standard MKL Sparse Blas solver routine mkl_dcsrtrsv.

```

```

/*-----*/
    if (RCI_request==3)
    {
        cvar1='L';
        cvar='N';
        cvar2='U';

mkl_dcsrtrsv(&cvar1,&cvar,&cvar2,&ivar,bilut,ibilut,jbilut,&tmp[ipar[21]-1],trvec);
        cvar1='U';
        cvar='N';
        cvar2='N';

mkl_dcsrtrsv(&cvar1,&cvar,&cvar2,&ivar,bilut,ibilut,jbilut,trvec,&tmp[ipar[22]-1]);
        goto ONE;
    }
/*-----*/
/* If RCI_request=4, then check if the norm of the next generated vector is
/* not zero up to rounding and computational errors. The norm is contained
/* in dpar[6] parameter
/*-----*/
    if (RCI_request==4)
    {
        if (dpar[6]<1.0E-12) goto COMPLETE;
        else goto ONE;
    }
/*-----*/
/* If RCI_request=anything else, then dfgmres subroutine failed
/* to compute the solution vector: computed_solution[N]
/*-----*/

```

```

else
{
    goto FAILED;
}

/*-----
/* Reverse Communication ends here
/* Get the current iteration number and the FGMRES solution (DO NOT FORGET to
/* call dfgmres_get routine as computed_solution is still containing
/* the initial guess!). Request to dfgmres_get to put the solution
/* into vector computed_solution[N] via ipar[12]
/*-----*/

COMPLETE: ipar[12]=0;

dfgmres_get(&ivar, computed_solution, rhs, &RCI_request, ipar, dpar, tmp,
            &itercount);

/*-----
/* Print solution vector: computed_solution[N] and the number
/* of iterations: itercount
/*-----*/

printf("The system has been solved \n");
printf("\nThe following solution has been obtained: \n");
for (i=0;i<N;i++)
{
    printf("computed_solution[%d]=",i);
    printf("%e\n",computed_solution[i]);
}

printf("\nThe expected solution is: \n");
for (i=0;i<N;i++)
{

```

```

        printf("expected_solution[%d]=" ,i);
        printf("%e\n",expected_solution[i]);
    }
    printf("\nNumber of iterations: %d\n" ,itercount);
    printf("\n");
    if(itercount==ref_nit && fabs(ref_norm2-nrm2)<1.e-6)
    {
printf("-----\n");
        printf("C example of FGMRES with ILUT preconditioner \n");
        printf("has successfully PASSED all stages of computations\n");
        printf("-----\n");
        return 0;
    }
    else
    {
        printf("Probably, the preconditioner was computed incorrectly:\n");
        printf("Either the preconditioner norm %e differs from the expected norm
                %e\n",nrm2,ref_norm2);
        printf("and/or the number of iterations %d differs from the expected number
                %d\n",itercount,ref_nit);
        printf("-----\n");
        printf("Unfortunately, FGMRES+ILUT C example has FAILED\n");
        printf("-----\n");
        return 0;
    }

FAILED:
        printf("The solver has returned the ERROR code %d", RCI_request);

FAILED1:

```

```
printf("-----\n");  
printf("Unfortunately, FGMRES+ILUT C example has FAILED\n");  
printf("-----\n");  
return 0;  
}
```



CBLAS Interface to the BLAS

This appendix presents CBLAS, the C interface to the Basic Linear Algebra Subprograms (BLAS) implemented in Intel® MKL.

Similar to BLAS, the CBLAS interface includes the following levels of functions:

- “Level 1 CBLAS” (vector-vector operations)
- “Level 2 CBLAS” (matrix-vector operations)
- “Level 3 CBLAS” (matrix-matrix operations).
- “Sparse CBLAS” (operations on sparse vectors).

To obtain the C interface, the Fortran routine names are prefixed with `cblas_` (for example, `dasum` becomes `cblas_dasum`). Names of all CBLAS functions are in lowercase letters.

Complex functions `?dotc` and `?dotu` become CBLAS subroutines (void functions); they return the complex result via a void pointer, added as the last parameter. CBLAS names of these functions are suffixed with `_sub`. For example, the BLAS function `cdotc` corresponds to `cblas_cdotc_sub`.

In the descriptions of CBLAS interfaces, links provided for each function group lead to the descriptions of the respective Fortran-interface BLAS functions.

CBLAS Arguments

The arguments of CBLAS functions obey the following rules:

- Input arguments are declared with the `const` modifier.
- Non-complex scalar input arguments are passed by value.
- Complex scalar input arguments are passed as void pointers.
- Array arguments are passed by address.
- BLAS character arguments are replaced by the appropriate enumerated type.
- Level 2 and Level 3 routines acquire an additional parameter of type `CBLAS_ORDER` as their first argument. This parameter specifies whether two-dimensional arrays are row-major (`CblasRowMajor`) or column-major (`CblasColMajor`).

Enumerated Types

The CBLAS interface uses the following enumerated types:

```
enum CBLAS_ORDER {
    CblasRowMajor=101, /* row-major arrays */
    CblasColMajor=102}; /* column-major arrays */

enum CBLAS_TRANSPOSE {
    CblasNoTrans=111, /* trans='N' */
    CblasTrans=112, /* trans='T' */
    CblasConjTrans=113}; /* trans='C' */

enum CBLAS_UPLO {
    CblasUpper=121, /* uplo ='U' */
    CblasLower=122}; /* uplo ='L' */

enum CBLAS_DIAG {
    CblasNonUnit=131, /* diag ='N' */
    CblasUnit=132}; /* diag ='U' */

enum CBLAS_SIDE {
    CblasLeft=141, /* side ='L' */
    CblasRight=142}; /* side ='R' */
```

Level 1 CBLAS

This is an interface to “[BLAS Level 1 Routines and Functions](#)”, which perform basic vector-vector operations.

?asum

```
float cblas_sasum(const int N, const float *X, const int incX);
double cblas_dasum(const int N, const double *X, const int incX);
float cblas_scasum(const int N, const void *X, const int incX);
double cblas_dzasum(const int N, const void *X, const int incX);
```

?axpy

```
void cblas_saxpy(const int N, const float alpha, const float *X, const int incX, float *Y,
  const int incY);

void cblas_daxpy(const int N, const double alpha, const double *X, const int incX, double
  *Y, const int incY);

void cblas_caxpy(const int N, const void *alpha, const void *X, const int incX, void *Y,
  const int incY);

void cblas_zaxpy(const int N, const void *alpha, const void *X, const int incX, void *Y,
  const int incY);
```

?copy

```
void cblas_scopy(const int N, const float *X, const int incX, float *Y, const int incY);
void cblas_dcopy(const int N, const double *X, const int incX, double *Y, const int incY);
void cblas_ccopy(const int N, const void *X, const int incX, void *Y, const int incY);
void cblas_zcopy(const int N, const void *X, const int incX, void *Y, const int incY);
```

?dot

```
float cblas_sdot(const int N, const float *X, const int incX, const float *Y, const int
  incY);

double cblas_ddot(const int N, const double *X, const int incX, const double *Y, const int
  incY);
```

?sdot

```
float cblas_sdsdot(const int N, const float *SB, const float *SX, const int incX, const
  float *SY, const int incY);

double cblas_dsdot(const int N, const float *SX, const int incX, const float *SY, const int
  incY);
```

?dotc

```
void cblas_cdotc_sub(const int N, const void *X, const int incX, const void *Y, const int
  incY, void *dotc);

void cblas_zdotc_sub(const int N, const void *X, const int incX, const void *Y, const int
  incY, void *dotc);
```

?dotu

```
void cblas_cdotu_sub(const int N, const void *X, const int incX, const void *Y, const int incY, void *dotu);
```

```
void cblas_zdotu_sub(const int N, const void *X, const int incX, const void *Y, const int incY, void *dotu);
```

?nrm2

```
float cblas_snrm2(const int N, const float *X, const int incX);
```

```
double cblas_dnrm2(const int N, const double *X, const int incX);
```

```
float cblas_scnrm2(const int N, const void *X, const int incX);
```

```
double cblas_dznrm2(const int N, const void *X, const int incX);
```

?rot

```
void cblas_srot(const int N, float *X, const int incX, float *Y, const int incY, const float c, const float s);
```

```
void cblas_drot(const int N, double *X, const int incX, double *Y, const int incY, const double c, const double s);
```

?rotg

```
void cblas_srotg(float *a, float *b, float *c, float *s);
```

```
void cblas_drotg(double *a, double *b, double *c, double *s);
```

?rotm

```
void cblas_srotm(const int N, float *X, const int incX, float *Y, const int incY, const float *P);
```

```
void cblas_drotm(const int N, double *X, const int incX, double *Y, const int incY, const double *P);
```

?rotmg

```
void cblas_srotmg(float *d1, float *d2, float *b1, const float b2, float *P);
```

```
void cblas_drotmg(double *d1, double *d2, double *b1, const double b2, double *P);
```

?scal

```
void cblas_sscal(const int N, const float alpha, float *X, const int incX);
```

```
void cblas_dscal(const int N, const double alpha, double *X, const int incX);
```

```
void cblas_cscal(const int N, const void *alpha, void *X, const int incX);
```

```
void cblas_zscal(const int N, const void *alpha, void *X, const int incX);
```

```
void cblas_csscal(const int N, const float alpha, void *X, const int incX);
```

```
void cblas_zdscal(const int N, const double alpha, void *X, const int incX);
```

?swap

```
void cblas_sswap(const int N, float *X, const int incX, float *Y, const int incY);
```

```
void cblas_dswap(const int N, double *X, const int incX, double *Y, const int incY);
```

```
void cblas_cswap(const int N, void *X, const int incX, void *Y, const int incY);
```

```
void cblas_zswap(const int N, void *X, const int incX, void *Y, const int incY);
```

i?amax

```
CBLAS_INDEX cblas_isamax(const int N, const float *X, const int incX);
```

```
CBLAS_INDEX cblas_idamax(const int N, const double *X, const int incX);
```

```
CBLAS_INDEX cblas_icamax(const int N, const void *X, const int incX);
```

```
CBLAS_INDEX cblas_izamax(const int N, const void *X, const int incX);
```

i?amin

```
CBLAS_INDEX cblas_isamin(const int N, const float *X, const int incX);
```

```
CBLAS_INDEX cblas_idamin(const int N, const double *X, const int incX);
```

```
CBLAS_INDEX cblas_icamin(const int N, const void *X, const int incX);
```

```
CBLAS_INDEX cblas_izamin(const int N, const void *X, const int incX);
```

Level 2 CBLAS

This is an interface to “BLAS Level 2 Routines”, which perform basic matrix-vector operations. Each C routine in this group has an additional parameter of type `CBLAS_ORDER` (the first argument) that determines whether the two-dimensional arrays use column-major or row-major storage.

?gbmv

```
void cblas_sgbmv(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE TransA, const int M, const int N, const int KL, const int KU, const float alpha, const float *A, const int lda, const float *X, const int incX, const float beta, float *Y, const int incY);
```

```
void cblas_dgbmv(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE TransA, const int M, const int N, const int KL, const int KU, const double alpha, const double *A, const int lda, const double *X, const int incX, const double beta, double *Y, const int incY);
```

```
void cblas_cgbmv(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE TransA, const int M, const int N, const int KL, const int KU, const void *alpha, const void *A, const int lda, const void *X, const int incX, const void *beta, void *Y, const int incY);
```

```
void cblas_zgbmv(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE TransA, const int M, const int N, const int KL, const int KU, const void *alpha, const void *A, const int lda, const void *X, const int incX, const void *beta, void *Y, const int incY);
```


?gemv

```
void cblas_sgemv(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE TransA, const int
M, const int N, const float alpha, const float *A, const int lda, const float *X, const
int incX, const float beta, float *Y, const int incY);
```

```
void cblas_dgemv(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE TransA, const int
M, const int N, const double alpha, const double *A, const int lda, const double *X, const
int incX, const double beta, double *Y, const int incY);
```

```
void cblas_cgemv(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE TransA, const int
M, const int N, const void *alpha, const void *A, const int lda, const void *X, const int
incX, const void *beta, void *Y, const int incY);
```

```
void cblas_zgemv(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE TransA, const int
M, const int N, const void *alpha, const void *A, const int lda, const void *X, const int
incX, const void *beta, void *Y, const int incY);
```

?ger

```
void cblas_sger(const enum CBLAS_ORDER order, const int M, const int N, const float alpha,
const float *X, const int incX, const float *Y, const int incY, float *A, const int lda);
```

```
void cblas_dger(const enum CBLAS_ORDER order, const int M, const int N, const double alpha,
const double *X, const int incX, const double *Y, const int incY, double *A, const int
lda);
```

?gerc

```
void cblas_cgerc(const enum CBLAS_ORDER order, const int M, const int N, const void *alpha,
const void *X, const int incX, const void *Y, const int incY, void *A, const int lda);
```

```
void cblas_zgerc(const enum CBLAS_ORDER order, const int M, const int N, const void *alpha,
const void *X, const int incX, const void *Y, const int incY, void *A, const int lda);
```

?geru

```
void cblas_cgeru(const enum CBLAS_ORDER order, const int M, const int N, const void *alpha,
const void *X, const int incX, const void *Y, const int incY, void *A, const int lda);
```

```
void cblas_zgeru(const enum CBLAS_ORDER order, const int M, const int N, const void *alpha,
const void *X, const int incX, const void *Y, const int incY, void *A, const int lda);
```

?hbm_v

```
void cblas_chbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N,
const int  $\bar{K}$ , const void *alpha, const void *A, const int lda, const void *X, const int incX,
const void *beta, void *Y, const int incY);
```

```
void cblas_zhbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N,
const int  $\bar{K}$ , const void *alpha, const void *A, const int lda, const void *X, const int incX,
const void *beta, void *Y, const int incY);
```

?hem_v

```
void cblas_chemv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N,
const void *alpha, const void *A, const int lda, const void *X, const int incX, const void
*beta, void *Y, const int incY);
```

```
void cblas_zhemv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N,
const void *alpha, const void *A, const int lda, const void *X, const int incX, const void
*beta, void *Y, const int incY);
```

?her

```
void cblas_cher(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const
float alpha, const void *X, const int incX, void *A, const int lda);
```

```
void cblas_zher(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const
double alpha, const void *X, const int incX, void *A, const int lda);
```

?her₂

```
void cblas_cher2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N,
const void *alpha, const void *X, const int incX, const void *Y, const int incY, void *A,
const int lda);
```

```
void cblas_zher2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N,
const void *alpha, const void *X, const int incX, const void *Y, const int incY, void *A,
const int lda);
```

?hpmv

```
void cblas_chpmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N,
const void *alpha, const void *Ap, const void *X, const int incX, const void *beta, void
*Y, const int incY);
```

```
void cblas_zhpmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N,
const void *alpha, const void *Ap, const void *X, const int incX, const void *beta, void
*Y, const int incY);
```

?hpr

```
void cblas_chpr(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const
float alpha, const void *X, const int incX, void *A);
```

```
void cblas_zhpr(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const
double alpha, const void *X, const int incX, void *A);
```

?hpr2

```
void cblas_chpr2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N,
const void *alpha, const void *X, const int incX, const void *Y, const int incY, void *Ap);
```

```
void cblas_zhpr2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N,
const void *alpha, const void *X, const int incX, const void *Y, const int incY, void *Ap);
```

?sbmv

```
void cblas_ssbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N,
const int K, const float alpha, const float *A, const int lda, const float *X, const int
incX, const float beta, float *Y, const int incY);
```

```
void cblas_dsbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N,
const int K, const double alpha, const double *A, const int lda, const double *X, const int
incX, const double beta, double *Y, const int incY);
```

?spmv

```
void cblas_sspmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N,
const float alpha, const float *Ap, const float *X, const int incX, const float beta, float
 *Y, const int incY);
```

```
void cblas_dspmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N,
const double alpha, const double *Ap, const double *X, const int incX, const double beta,
double *Y, const int incY);
```

?spr

```
void cblas_sspr(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const
float alpha, const float *X, const int incX, float *Ap);
```

```
void cblas_dspr(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const
double alpha, const double *X, const int incX, double *Ap);
```

?spr2

```
void cblas_sspr2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N,
const float alpha, const float *X, const int incX, const float *Y, const int incY, float
 *A);
```

```
void cblas_dspr2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N,
const double alpha, const double *X, const int incX, const double *Y, const int incY, double
 *A);
```

?symv

```
void cblas_ssymv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N,
const float alpha, const float *A, const int lda, const float *X, const int incX, const
float beta, float *Y, const int incY);
```

```
void cblas_dsymv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N,
const double alpha, const double *A, const int lda, const double *X, const int incX, const
double beta, double *Y, const int incY);
```

?syr

```
void cblas_ssyrr(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const
float alpha, const float *X, const int incX, float *A, const int lda);
```

```
void cblas_dsyr(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N, const
double alpha, const double *X, const int incX, double *A, const int lda);
```

?syr2

```
void cblas_ssyrr2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N,
const float alpha, const float *X, const int incX, const float *Y, const int incY, float
*A, const int lda);
```

```
void cblas_dsyr2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N,
const double alpha, const double *X, const int incX, const double *Y, const int incY, double
*A, const int lda);
```

?tbmv

```
void cblas_stbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const int K, const float
*A, const int lda, float *X, const int incX);
```

```
void cblas_dtbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const int K, const double
*A, const int lda, double *X, const int incX);
```

```
void cblas_ctbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const int K, const void
*A, const int lda, void *X, const int incX);
```

```
void cblas_ztbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const int K, const void
*A, const int lda, void *X, const int incX);
```

?tbsv

```
void cblas_stbsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const int K, const float
*A, const int lda, float *X, const int incX);
```

```
void cblas_dtbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const int K, const double
*A, const int lda, double *X, const int incX);
```

```
void cblas_ctbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const int K, const void
*A, const int lda, void *X, const int incX);
```

```
void cblas_ztbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const int K, const void
*A, const int lda, void *X, const int incX);
```

?tpmv

```
void cblas_stpmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const float *Ap, float *X,
const int incX);
```

```
void cblas_dtpmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const double *Ap, double
*X, const int incX);
```

```
void cblas_ctpmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const void *Ap, void *X,
const int incX);
```

```
void cblas_ztpmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const void *Ap, void *X,
const int incX);
```

?tpsv

```
void cblas_stpsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const float *Ap, float *X,
const int incX);
```

```
void cblas_dtpsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const double *Ap, double
*X, const int incX);
```

```
void cblas_ctpsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const void *Ap, void *X,
const int incX);
```

```
void cblas_ztpsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const void *Ap, void *X,
const int incX);
```

?trmv

```
void cblas_strmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const float *A, const int
lda, float *X, const int incX);
```

```
void cblas_dtrmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const double *A, const int
lda, double *X, const int incX);
```

```
void cblas_ctrmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const void *A, const int
lda, void *X, const int incX);
```

```
void cblas_ztrmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const void *A, const int
lda, void *X, const int incX);
```

?trsv

```
void cblas_strsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const float *A, const int
lda, float *X, const int incX);
```

```
void cblas_dtrsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const double *A, const int
lda, double *X, const int incX);
```

```
void cblas_ctrsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const void *A, const int
lda, void *X, const int incX);
```

```
void cblas_ztrsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const void *A, const int
lda, void *X, const int incX);
```

Level 3 CBLAS

This is an interface to “[BLAS Level 3 Routines](#)”, which perform basic matrix-matrix operations. Each C routine in this group has an additional parameter of type `CBLAS_ORDER` (the first argument) that determines whether the two-dimensional arrays use column-major or row-major storage.

?gemm

```
void cblas_sgemm(const enum CBLAS_ORDER Order, const enum CBLAS_TRANSPOSE TransA, const
enum CBLAS_TRANSPOSE TransB, const int M, const int N, const int K, const float alpha, const
float *A, const int lda, const float *B, const int ldb, const float beta, float *C, const
int ldc);
```

```
void cblas_dgemm(const enum CBLAS_ORDER Order, const enum CBLAS_TRANSPOSE TransA, const
enum CBLAS_TRANSPOSE TransB, const int M, const int N, const int K, const double alpha,
const double *A, const int lda, const double *B, const int ldb, const double beta, double
*C, const int ldc);
```

```
void cblas_cgemm(const enum CBLAS_ORDER Order, const enum CBLAS_TRANSPOSE TransA, const
enum CBLAS_TRANSPOSE TransB, const int M, const int N, const int K, const void *alpha, const
void *A, const int lda, const void *B, const int ldb, const void *beta, void *C, const int
ldc);
```

```
void cblas_zgemm(const enum CBLAS_ORDER Order, const enum CBLAS_TRANSPOSE TransA, const
enum CBLAS_TRANSPOSE TransB, const int M, const int N, const int K, const void *alpha, const
void *A, const int lda, const void *B, const int ldb, const void *beta, void *C, const int
ldc);
```

?hemm

```
void cblas_chemm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const enum
CBLAS_UPLO Uplo, const int M, const int N, const void *alpha, const void *A, const int lda,
const void *B, const int ldb, const void *beta, void *C, const int ldc);
```

```
void cblas_zhemm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const enum
CBLAS_UPLO Uplo, const int M, const int N, const void *alpha, const void *A, const int lda,
const void *B, const int ldb, const void *beta, void *C, const int ldc);
```


?herk

```
void cblas_cherk(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE Trans, const int N, const int K, const float alpha, const void *A, const
int lda, const float beta, void *C, const int ldc);
```

```
void cblas_zherk(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE Trans, const int N, const int K, const double alpha, const void *A, const
int lda, const double beta, void *C, const int ldc);
```

?her2k

```
void cblas_cher2k(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE Trans, const int N, const int K, const void *alpha, const void *A, const
int lda, const void *B, const int ldb, const float beta, void *C, const int ldc);
```

```
void cblas_zher2k(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE Trans, const int N, const int K, const void *alpha, const void *A, const
int lda, const void *B, const int ldb, const double beta, void *C, const int ldc);
```

?symm

```
void cblas_ssymm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const enum
CBLAS_UPLO Uplo, const int M, const int N, const float alpha, const float *A, const int
lda, const float *B, const int ldb, const float beta, float *C, const int ldc);
```

```
void cblas_dsymm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const enum
CBLAS_UPLO Uplo, const int M, const int N, const double alpha, const double *A, const int
lda, const double *B, const int ldb, const double beta, double *C, const int ldc);
```

```
void cblas_csymm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const enum
CBLAS_UPLO Uplo, const int M, const int N, const void *alpha, const void *A, const int lda,
const void *B, const int ldb, const void *beta, void *C, const int ldc);
```

```
void cblas_zsymm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const enum
CBLAS_UPLO Uplo, const int M, const int N, const void *alpha, const void *A, const int lda,
const void *B, const int ldb, const void *beta, void *C, const int ldc);
```

?syrk

```
void cblas_ssyrrk(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE Trans, const int N, const int K, const float alpha, const float *A, const
int lda, const float beta, float *C, const int ldc);
```

```
void cblas_dsyrrk(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE Trans, const int N, const int K, const double alpha, const double *A, const
int lda, const double beta, double *C, const int ldc);
```

```
void cblas_csyrrk(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE Trans, const int N, const int K, const void *alpha, const void *A, const
int lda, const void *beta, void *C, const int ldc);
```

```
void cblas_zsyrrk(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE Trans, const int N, const int K, const void *alpha, const void *A, const
int lda, const void *beta, void *C, const int ldc);
```

?syr2k

```
void cblas_ssyrr2k(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE Trans, const int N, const int K, const float alpha, const float *A, const
int lda, const float *B, const int ldb, const float beta, float *C, const int ldc);
```

```
void cblas_dsyrr2k(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE Trans, const int N, const int K, const double alpha, const double *A, const
int lda, const double *B, const int ldb, const double beta, double *C, const int ldc);
```

```
void cblas_csyrr2k(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE Trans, const int N, const int K, const void *alpha, const void *A, const int
lda, const void *B, const int ldb, const void *beta, void *C, const int ldc);
```

```
void cblas_zsyrr2k(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const enum
CBLAS_TRANSPOSE Trans, const int N, const int K, const void *alpha, const void *A, const
int lda, const void *B, const int ldb, const void *beta, void *C, const int ldc);
```

?trmm

```
void cblas_strmm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
M, const int N, const float alpha, const float *A, const int lda, float *B, const int ldb);
```

```
void cblas_dtrmm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
M, const int N, const double alpha, const double *A, const int lda, double *B, const int
ldb);
```

```
void cblas_ctrmm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
M, const int N, const void *alpha, const void *A, const int lda, void *B, const int ldb);
```

```
void cblas_ztrmm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
M, const int N, const void *alpha, const void *A, const int lda, void *B, const int ldb);
```

?trsm

```
void cblas_strsm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
M, const int N, const float alpha, const float *A, const int lda, float *B, const int ldb);
```

```
void cblas_dtrsm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
M, const int N, const double alpha, const double *A, const int lda, double *B, const int
ldb);
```

```
void cblas_ctrsm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
M, const int N, const void *alpha, const void *A, const int lda, void *B, const int ldb);
```

```
void cblas_ztrsm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const enum
CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
M, const int N, const void *alpha, const void *A, const int lda, void *B, const int ldb);
```

Sparse CBLAS

This is an interface to “[Sparse BLAS Level 1 Routines and Functions](#)”, which perform a number of common vector operations on sparse vectors stored in compressed form.

Note that all index parameters, *indx*, are in C-type notation and vary in the range $[0..N-1]$.

?axpyi

```
void cblas_saxpy(const int N, const float alpha, const float *X, const int *indx, float *Y);
```

```
void cblas_daxpy(const int N, const double alpha, const double *X, const int *indx, double *Y);
```

```
void cblas_caxpy(const int N, const void *alpha, const void *X, const int *indx, void *Y);
```

```
void cblas_zaxpy(const int N, const void *alpha, const void *X, const int *indx, void *Y);
```

?doti

```
float cblas_sdoti(const int N, const float *X, const int *indx, const float *Y);
```

```
double cblas_ddoti(const int N, const double *X, const int *indx, const double *Y);
```

?dotci

```
void cblas_cdotci_sub(const int N, const void *X, const int *indx, const void *Y, void *dotui);
```

```
void cblas_zdotci_sub(const int N, const void *X, const int *indx, const void *Y, void *dotui);
```

?dotui

```
void cblas_cdotui_sub(const int N, const void *X, const int *indx, const void *Y, void *dotui);
```

```
void cblas_zdotui_sub(const int N, const void *X, const int *indx, const void *Y, void *dotui);
```

?gthr

```
void cblas_sgthr(const int N, const float *Y, float *X, const int *indx);
```

```
void cblas_dgthr(const int N, const double *Y, double *X, const int *indx);
```

```
void cblas_cgthr(const int N, const void *Y, void *X, const int *indx);
```

```
void cblas_zgthr(const int N, const void *Y, void *X, const int *indx);
```

?gthrz

```
void cblas_sgthrz(const int N, float *Y, float *X, const int *indx);
void cblas_dgthrz(const int N, double *Y, double *X, const int *indx);
void cblas_cgthrz(const int N, void *Y, void *X, const int *indx);
void cblas_zgthrz(const int N, void *Y, void *X, const int *indx);
```

?roti

```
void cblas_sroti(const int N, float *X, const int *indx, float *Y, const float c, const
float s);
void cblas_droti(const int N, double *X, const int *indx, double *Y, const double c, const
double s);
```

?sctr

```
void cblas_ssctr(const int N, const float *X, const int *indx, float *Y);
void cblas_dsctr(const int N, const double *X, const int *indx, double *Y);
void cblas_csctr(const int N, const void *X, const int *indx, void *Y);
void cblas_zsctr(const int N, const void *X, const int *indx, void *Y);
```

Specific Features of Fortran 95 Interfaces for LAPACK Routines



Intel® MKL implements Fortran 95 interface for LAPACK package, further referred to as MKL LAPACK95, to provide full capacity of MKL FORTRAN 77 LAPACK routines. This is the principal difference of Intel MKL from the Netlib Fortran 95 implementation for LAPACK.

A new feature of MKL LAPACK95 by comparison with Intel MKL LAPACK77 implementation is presenting a package of source interfaces along with wrappers that make the implementation compiler-independent. As a result, the MKL LAPACK package can be used in all programming environments intended for Fortran 95.

Depending on the degree and type of difference from Netlib implementation, the MKL LAPACK95 interfaces fall into several groups that require different transformations (see [“MKL Fortran 95 Interfaces for LAPACK Routines vs. Netlib Implementation”](#)). The groups are given in full with the calling sequences of the routines and appropriate differences from Netlib analogs.

The following conventions are used:

```
<interface> ::= <name of interface> '(' <arguments list> ')'  
<arguments list> ::= <first argument> {<argument>}*  
<first argument> ::= < identifier >  
<argument> ::= <required argument>|<optional argument>  
<required argument> ::= ',' <identifier>  
<optional argument> ::= '[,' <identifier> ']'  
<name of interface> ::= <identifier>
```

where defined notions are separated from definitions by `::=`, notion names are marked by angle brackets, terminals are given in quotes, and `{...}*` denotes repetition zero, one, or more times.

<first argument> and each *<required argument>* should be present in all calls of denoted interface, *<optional argument>* may be omitted. Comments to interface definitions are provided where necessary. Comment lines begin with character `!`.

Two interfaces with one name are presented when two variants of subroutine calls (separated by types of arguments) exist.

Interfaces Identical to Netlib

```

GETRI(A, IPIV [, INFO])
GEEQU(A,R,C[,ROWCND][,COLCND][,AMAX][,INFO])
GESV(A,B[,IPIV][,INFO])
GESVX(A,B,X[,AF][,IPIV][,FACT][,TRANS][,EQUED][,R][,C][,FERR][,BERR]
[,RCOND][,RPVGRW][,INFO])
GBSV(A,B[,KL][,IPIV][,INFO])
GTSV(DL,D,DU,B[,INFO])
GTSVX(DL,D,DU,B,X[,DLF][,DF][,DUF][,DU2][,IPIV][,FACT][,TRANS][,FERR]
[,BERR][,RCOND][,INFO])
POSV(A,B[,UPLO][,INFO])
POSVX(A,B,X[,UPLO][,AF][,FACT][,EQUED][,S][,FERR][,BERR][,RCOND][,INFO])
PPSV(A,B[,UPLO][,INFO])
PPSVX(A,B,X[,UPLO][,AF][,FACT][,EQUED][,S][,FERR][,BERR][,RCOND][,INFO])
PBSV(A,B[,UPLO][,INFO])
PBSVX(A,B,X[,UPLO][,AF][,FACT][,EQUED][,S][,FERR][,BERR][,RCOND][,INFO])
PTSV(D,E,B[,INFO])
PTSVX(D,E,B,X[,DF][,EF][,FACT][,FERR][,BERR][,RCOND][,INFO])
SYSV(A,B[,UPLO][,IPIV][,INFO])
SYSVX(A,B,X[,UPLO][,AF][,IPIV][,FACT][,FERR][,BERR][,RCOND][,INFO])
HESVX(A,B,X[,UPLO][,AF][,IPIV][,FACT][,FERR][,BERR][,RCOND][,INFO])
SYTRD(A,TAU[,UPLO][,INFO])
ORGTR(A,TAU[,UPLO][,INFO])
HETRD(A,TAU[,UPLO][,INFO])
UNGTR(A,TAU[,UPLO][,INFO])
SYGST(A,B[,ITYPE][,UPLO][,INFO])
HEGST(A,B[,ITYPE][,UPLO][,INFO])

```



```

GELS(A,B[,TRANS][,INFO])
GELSY(A,B[,RANK][,JPVT][,RCOND][,INFO])
GELSS(A,B[,RANK][,S][,RCOND][,INFO])
GELSD(A,B[,RANK][,S][,RCOND][,INFO])
GGLSE(A,B,C,D,X[,INFO])
GGGLM(A,B,D,X,Y[,INFO])
SYEV(A,W[,JOBZ][,UPLO][,INFO])
HEEV(A,W[,JOBZ][,UPLO][,INFO])
SYEVD(A,W[,JOBZ][,UPLO][,INFO])
SPEV(A,W[,UPLO][,Z][,INFO])
HPEV(A,W[,UPLO][,Z][,INFO])
SPEVD(A,W[,UPLO][,Z][,INFO])
HPEVD(A,W[,UPLO][,Z][,INFO])
SPEVX(A,W[,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,ABSTOL][,INFO])
HPEVX(A,W[,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,ABSTOL][,INFO])
SBEV(A,W[,UPLO][,Z][,INFO])
HBEV(A,W[,UPLO][,Z][,INFO])
SBEVD(A,W[,UPLO][,Z][,INFO])
HBEVD(A,W[,UPLO][,Z][,INFO])
SBEVX(A,W[,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,Q][,ABSTOL]
[,INFO])
HBEVX(A,W[,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,Q][,ABSTOL]
[,INFO])
HPGV(A,B,W[,ITYPE][,UPLO][,Z][,INFO])
STEV(D,E[,Z][,INFO])
STEVD(D,E[,Z][,INFO])
STEVX(D,E,W[,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,ABSTOL][,INFO])
STEVR(D,E,W[,Z][,VL][,VU][,IL][,IU][,M][,ISUPPZ][,ABSTOL][,INFO])

```

GEES(A,WR,WI[,VS][,SELECT][,SDIM][,INFO])

GEES(A,W[,VS][,SELECT][,SDIM][,INFO])

GEESEX(A,WR,WI[,VS][,SELECT][,SDIM][,RCONDE][,RCONDV][,INFO])

GEESEX(A,W[,VS][,SELECT][,SDIM][,RCONDE][,RCONDV][,INFO])

GEEV(A,WR,WI[,VL][,VR][,INFO])

GEEV(A,W[,VL][,VR][,INFO])

GEEVX(A,WR,WI[,VL][,VR][,BALANC][,ILO][,IHI][,SCALE][,ABNRM][,RCONDE][,RCONDV][,INFO])

GEEVX(A,W[,VL][,VR][,BALANC][,ILO][,IHI][,SCALE][,ABNRM][,RCONDE][,RCONDV][,INFO])

GESVD(A,S[,U][,VT][,WW][,JOB][,INFO])

GGSDV(A,B,ALPHA,BETA[,K][,L][,U][,V][,Q][,IWORK][,INFO])

SYGV(A,B,W[,ITYPE][,JOBZ][,UPLO][,INFO])

HEGV(A,B,W[,ITYPE][,JOBZ][,UPLO][,INFO])

SYGVD(A,B,W[,ITYPE][,JOBZ][,UPLO][,INFO])

HEGVD(A,B,W[,ITYPE][,JOBZ][,UPLO][,INFO])

SPGV(A,B,W[,ITYPE][,UPLO][,Z][,INFO])

SBGV(A,B,W[,UPLO][,Z][,INFO])

HBGV(A,B,W[,UPLO][,Z][,INFO])

GGES(A,B,ALPHAR,ALPHAI,BETA[,VSL][,VSR][,SELECT][,SDIM][,INFO])

GGES(A,B,ALPHA,BETA[,VSL][,VSR][,SELECT][,SDIM][,INFO])

GGESX(A,B,ALPHAR,ALPHAI,BETA[,VSL][,VSR][,SELECT][,SDIM][,RCONDE][,RCONDV][,INFO])

GGESX(A,B,ALPHA,BETA[,VSL][,VSR][,SELECT][,SDIM][,RCONDE][,RCONDV][,INFO])

GGEV(A,B,ALPHAR,ALPHAI,BETA[,VL][,VR][,INFO])

GGEV(A,B,ALPHA,BETA[,VL][,VR][,INFO])

GGEVX(A,B,ALPHAR,ALPHAI,BETA[,VL][,VR][,BALANC][,ILO][,IHI][,LSCALE][,RSCALE][,ABNRM][,BBNRM][,RCONDE][,RCONDV][,INFO])

GGEVX(A,B,ALPHA,BETA[,VL][,VR][,BALANC][,ILO][,IHI][,LSCALE][,RSCALE][,ABNRM]

```
[,BBNRM][,RCONDE][,RCONDV][,INFO])
GERFS(A,AF,IPIV,B,X[,TRANS][,FERR][,BERR][,INFO])
```

Interfaces with Replaced Argument Names

Argument names in the routines of this group are replaced as follows:

Netlib Argument Name	MKL Argument Name
AP	A
AB	A
AFP	AF
BP	B
BB	B

```
SPSV(A,B[,UPLO][,IPIV][,INFO])

! netlib: (AP,B,UPLO,IPIV,INFO)

SPSVX(A,B,X[,UPLO][,AF][,IPIV][,FACT][,FERR][,BERR][,RCOND][,INFO])

! netlib: (A,B,X,UPLO,AFP,IPIV,FACT,FERR,BERR,RCOND,INFO)

HPSVX(A,B,X[,UPLO][,AF][,IPIV][,FACT][,FERR][,BERR][,RCOND][,INFO])

! netlib: (A,B,X,UPLO,AFP,IPIV,FACT,FERR,BERR,RCOND,INFO)

HEEVD(A,W[,JOB][,UPLO][,INFO])

! netlib: (A,W,JOBZ,UPLO,INFO)

SPGVD(A,B,W[,ITYPE][,UPLO][,Z][,INFO])

! netlib: (AP,BP,W,ITYPE,UPLO,Z,INFO)

HPGVD(A,B,W[,ITYPE][,UPLO][,Z][,INFO])

! netlib: (AP,BP,W,ITYPE,UPLO,Z,INFO)

SPGVX(A,B,W[,ITYPE][,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,ABSTOL][,INFO])

! netlib: (AP,BP,W,ITYPE,UPLO,Z,VL,VU,IL,IU,M,IFAIL,ABSTOL,INFO)

HPGVX(A,B,W[,ITYPE][,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,ABSTOL][,INFO])

! netlib: (AP,BP,W,ITYPE,UPLO,Z,VL,VU,IL,IU,M,IFAIL,ABSTOL,INFO)

SBGVD(A,B,W[,UPLO][,Z][,INFO])
```

```

!    netlib: (AB,BB,W,UPLO,Z,INFO)
HBGVD(A,B,W[,UPLO][,Z][,INFO])

!    netlib: (AB,BB,W,UPLO,Z,INFO)
SBGVX(A,B,W[,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,Q][,ABSTOL][,INFO])

!    netlib: (AB,BB,W,UPLO,Z,VL,VU,IL,IU,M,IFAIL,Q,ABSTOL,INFO)
HBGVX(A,B,W[,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,Q][,ABSTOL][,INFO])

!    netlib: (AB,BB,W,UPLO,Z,VL,VU,IL,IU,M,IFAIL,Q,ABSTOL,INFO)
GBSVX(A,B,X[,KL][,AF][,IPIV][,FACT][,TRANS][,EQUED][,R][,C][,FERR]
[,BERR][,RCOND][,RPVGRW][,INFO])

!    netlib: (A,B,X,KL,AFB,IPIV,FACT,TRANS,EQUED,R,C,FERR,
! BERR,RCOND,RPVGRW,INFO)

```

Modified Netlib Interfaces

```

SYEVX(A,W[,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,ABSTOL][,INFO])
!   Interface netlib95 exists, parameters:
!   netlib: (A,W,JOBZ,UPLO,VL,VU,IL,IU,M,IFAIL,ABSTOL,INFO)
!   Different order for parameter UPLO, netlib: 4, mkl: 3
!   Absent mkl parameter: JOBZ
!   Extra mkl parameter: Z
HEEVX(A,W[,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,ABSTOL][,INFO])
!   Interface netlib95 exists, parameters:
!   netlib: (A,W,JOBZ,UPLO,VL,VU,IL,IU,M,IFAIL,ABSTOL,INFO)
!   Different order for parameter UPLO, netlib: 4, mkl: 3
!   Absent mkl parameter: JOBZ
!   Extra mkl parameter: Z
SYEVR(A,W[,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,ISUPPZ][,ABSTOL][,INFO])
!   Interface netlib95 exists, parameters:
!   netlib: (A,W,JOBZ,UPLO,VL,VU,IL,IU,M,ISUPPZ,ABSTOL,INFO)
!   Different order for parameter UPLO, netlib: 4, mkl: 3
!   Absent mkl parameter: JOBZ
!   Extra mkl parameter: Z
HEEVR(A,W[,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,ISUPPZ][,ABSTOL][,INFO])
!   Interface netlib95 exists, parameters:
!   netlib: (A,W,JOBZ,UPLO,VL,VU,IL,IU,M,ISUPPZ,ABSTOL,INFO)
!   Different order for parameter UPLO, netlib: 4, mkl: 3
!   Absent mkl parameter: JOBZ
!   Extra mkl parameter: Z
GESDD(A,S[,U][,VT][,JOBZ][,INFO])
!   Interface netlib95 exists, parameters:

```

```

!   netlib: (A,S,U,VT,WW,JOB,INFO)
!   Different number for parameter, netlib: 7, mkl: 6
!   Absent mkl parameter: WW
!   Absent mkl parameter: JOB
!   Different order for parameter INFO, netlib: 7, mkl: 6
!   Extra mkl parameter: JOBZ
SYGVX(A,B,W[,ITYPE][,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,ABSTOL][,INFO])
!   Interface netlib95 exists, parameters:
!   netlib: (A,B,W,ITYPE,JOBZ,UPLO,VL,VU,IL,IU,M,IFAIL,ABSTOL,INFO)
!   Different order for parameter UPLO, netlib: 6, mkl: 5
!   Absent mkl parameter: JOBZ
!   Extra mkl parameter: Z
HEGVX(A,B,W[,ITYPE][,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,ABSTOL][,INFO])
!   Interface netlib95 exists, parameters:
!   netlib: (A,B,W,ITYPE,JOBZ,UPLO,VL,VU,IL,IU,M,IFAIL,ABSTOL,INFO)
!   Different order for parameter UPLO, netlib: 6, mkl: 5
!   Absent mkl parameter: JOBZ
!   Extra mkl parameter: Z
GETRS(A,IPIV,B[,TRANS][,INFO])
!   Interface netlib95 exists:
!   Different intents for parameter A, netlib: INOUT, mkl: IN

```

Interfaces Absent From Netlib

```

GTTRF(DL,D,DU,DU2[,IPIV][,INFO])
PPTRF(A[,UPLO][,INFO])
PBTRF(A[,UPLO][,INFO])
PTTRF(D,E[,INFO])
SYTRF(A[,UPLO][,IPIV][,INFO])
HETRF(A[,UPLO][,IPIV][,INFO])
SPTRF(A[,UPLO][,IPIV][,INFO])
HPTRF(A[,UPLO][,IPIV][,INFO])
GBTRS(A,B,IPIV[,KL][,TRANS][,INFO])
GTTRS(DL,D,DU,DU2,B,IPIV[,TRANS][,INFO])
POTRS(A,B[,UPLO][,INFO])
PPTRS(A,B[,UPLO][,INFO])
PBTRS(A,B[,UPLO][,INFO])
PTTRS(D,E,B[,INFO])
PTTRS(D,E,B[,UPLO][,INFO])
SYTRS(A,B,IPIV[,UPLO][,INFO])
HETRS(A,B,IPIV[,UPLO][,INFO])
SPTRS(A,B,IPIV[,UPLO][,INFO])
HPTRS(A,B,IPIV[,UPLO][,INFO])
TRTRS(A,B[,UPLO][,TRANS][,DIAG][,INFO])
TPTRS(A,B[,UPLO][,TRANS][,DIAG][,INFO])
TBTRS(A,B[,UPLO][,TRANS][,DIAG][,INFO])
GECON(A,ANORM,RCOND[,NORM][,INFO])
GBCON(A,IPIV,ANORM,RCOND[,KL][,NORM][,INFO])
GTCON(DL,D,DU,DU2,IPIV,ANORM,RCOND[,NORM][,INFO])
POCON(A,ANORM,RCOND[,UPLO][,INFO])

```

PPCON (A, ANORM, RCOND[, UPLO] [, INFO])
 PBCON (A, ANORM, RCOND[, UPLO] [, INFO])
 PTCON (D, E, ANORM, RCOND[, INFO])
 SYCON (A, IPIV, ANORM, RCOND[, UPLO] [, INFO])
 HECON (A, IPIV, ANORM, RCOND[, UPLO] [, INFO])
 SPCON (A, IPIV, ANORM, RCOND[, UPLO] [, INFO])
 HPCON (A, IPIV, ANORM, RCOND[, UPLO] [, INFO])
 TRCON (A, RCOND[, UPLO] [, DIAG] [, NORM] [, INFO])
 TPCON (A, RCOND[, UPLO] [, DIAG] [, NORM] [, INFO])
 TBCON (A, RCOND[, UPLO] [, DIAG] [, NORM] [, INFO])
 GBRFS (A, AF, IPIV, B, X[, KL] [, TRANS] [, FERR] [, BERR] [, INFO])
 GTRFS (DL, D, DU, DLF, DF, DUF, DU2, IPIV, B, X[, TRANS] [, FERR] [, BERR] [, INFO])
 PORFS (A, AF, B, X[, UPLO] [, FERR] [, BERR] [, INFO])
 PPRFS (A, AF, B, X[, UPLO] [, FERR] [, BERR] [, INFO])
 PBRFS (A, AF, B, X[, UPLO] [, FERR] [, BERR] [, INFO])
 PTRFS (D, DF, E, EF, B, X[, FERR] [, BERR] [, INFO])
 PTRFS (D, DF, E, EF, B, X[, UPLO] [, FERR] [, BERR] [, INFO])
 SYRFS (A, AF, IPIV, B, X[, UPLO] [, FERR] [, BERR] [, INFO])
 HERFS (A, AF, IPIV, B, X[, UPLO] [, FERR] [, BERR] [, INFO])
 SPRFS (A, AF, IPIV, B, X[, UPLO] [, FERR] [, BERR] [, INFO])
 HPRFS (A, AF, IPIV, B, X[, UPLO] [, FERR] [, BERR] [, INFO])
 TRRFS (A, B, X[, UPLO] [, TRANS] [, DIAG] [, FERR] [, BERR] [, INFO])
 TPRFS (A, B, X[, UPLO] [, TRANS] [, DIAG] [, FERR] [, BERR] [, INFO])
 TBRFS (A, B, X[, UPLO] [, TRANS] [, DIAG] [, FERR] [, BERR] [, INFO])
 POTRI (A[, UPLO] [, INFO])
 PPTRI (A[, UPLO] [, INFO])
 SYTRI (A, IPIV[, UPLO] [, INFO])
 HETRI (A, IPIV[, UPLO] [, INFO])


```

SPTRI (A, IPIV[,UPLO] [, INFO])
HPTRI (A, IPIV[,UPLO] [, INFO])
TRTRI (A[,UPLO] [, DIAG] [, INFO])
TPTRI (A[,UPLO] [, DIAG] [, INFO])
GBEQU (A,R,C[,KL] [, ROWCND] [, COLCND] [, AMAX] [, INFO])
POEQU (A,S[,SCOND] [, AMAX] [, INFO])
PPEQU (A,S[,SCOND] [, AMAX] [, UPLO] [, INFO])
PBEQU (A,S[,SCOND] [, AMAX] [, UPLO] [, INFO])
HESV (A,B[,UPLO] [, IPIV] [, INFO])
HPSV (A,B[,UPLO] [, IPIV] [, INFO])
GEQRF (A[,TAU] [, INFO])
GEQPF (A,JPVT[,TAU] [, INFO])
GEQP3 (A,JPVT[,TAU] [, INFO])
ORGQR (A,TAU[, INFO])
ORMQR (A,TAU,C[,SIDE] [, TRANS] [, INFO])
UNGQR (A,TAU[, INFO])
UNMQR (A,TAU,C[,SIDE] [, TRANS] [, INFO])
GELQF (A[,TAU] [, INFO])
ORGLQ (A,TAU[, INFO])
ORMLQ (A,TAU,C[,SIDE] [, TRANS] [, INFO])
UNGLQ (A,TAU[, INFO])
UNMLQ (A,TAU,C[,SIDE] [, TRANS] [, INFO])
GEQLF (A[,TAU] [, INFO])
ORGQL (A,TAU[, INFO])
UNGQL (A,TAU[, INFO])
ORMQL (A,TAU,C[,SIDE] [, TRANS] [, INFO])
UNMQL (A,TAU,C[,SIDE] [, TRANS] [, INFO])
GERQF (A[,TAU] [, INFO])

```

ORGRQ (A, TAU[, INFO])
 UNGRQ (A, TAU[, INFO])
 ORMRQ (A, TAU, C[, SIDE] [, TRANS] [, INFO])
 UNMRQ (A, TAU, C[, SIDE] [, TRANS] [, INFO])
 TZRF (A[, TAU] [, INFO])
 ORMRZ (A, TAU, C, L[, SIDE] [, TRANS] [, INFO])
 UNMRZ (A, TAU, C, L[, SIDE] [, TRANS] [, INFO])
 GGQRF (A, B[, TAUA] [, TAUB] [, INFO])
 GGRQF (A, B[, TAUA] [, TAUB] [, INFO])
 GEBRD (A[, D] [, E] [, TAUQ] [, TAUP] [, INFO])
 GGBRD (A[, C] [, D] [, E] [, Q] [, PT] [, KL] [, M] [, INFO])
 ORGBR (A, TAU[, VECT] [, INFO])
 ORMBR (A, TAU, C[, VECT] [, SIDE] [, TRANS] [, INFO])
 ORMTR (A, TAU, C[, SIDE] [, UPLO] [, TRANS] [, INFO])
 UNGBR (A, TAU[, VECT] [, INFO])
 UNMBR (A, TAU, C[, VECT] [, SIDE] [, TRANS] [, INFO])
 BDSQR (D, E[, VT] [, U] [, C] [, UPLO] [, INFO])
 BDSDC (D, E[, U] [, VT] [, Q] [, IQ] [, UPLO] [, INFO])
 UNMTR (A, TAU, C[, SIDE] [, UPLO] [, TRANS] [, INFO])
 SPTRD (A, TAU[, UPLO] [, INFO])
 OPGTR (A, TAU, Q[, UPLO] [, INFO])
 OPMTR (A, TAU, C[, SIDE] [, UPLO] [, TRANS] [, INFO])
 HPTRD (A, TAU[, UPLO] [, INFO])
 UPGTR (A, TAU, Q[, UPLO] [, INFO])
 UPMTR (A, TAU, C[, SIDE] [, UPLO] [, TRANS] [, INFO])
 SBTRD (A[, Q] [, VECT] [, UPLO] [, INFO])
 HBTRD (A[, Q] [, VECT] [, UPLO] [, INFO])
 STERF (D, E[, INFO])

```

STEQR(D,E[,Z][,COMPZ][,INFO])
STEDC(D,E[,Z][,COMPZ][,INFO])
STEGR(D,E,W[,Z][,VL][,VU][,IL][,IU][,M][,ISUPPZ][,ABSTOL][,INFO])
PTEQR(D,E[,Z][,COMPZ][,INFO])
STEBZ(D,E,M,NSPLIT,W,IBLOCK,ISPLIT[,ORDER][,VL][,VU][,IL][,IU][,ABSTOL][,INFO])
STEIN(D,E,W,IBLOCK,ISPLIT,Z[,IFAILV][,INFO])
DISNA(D,SEP[,JOB][,MINMN][,INFO])
SPGST(A,B[,ITYPE][,UPLO][,INFO])
HPGST(A,B[,ITYPE][,UPLO][,INFO])
SBGST(A,B[,X][,UPLO][,INFO])
HBGST(A,B[,X][,UPLO][,INFO])
PBSTF(B[,UPLO][,INFO])
GEHRD(A[,TAU][,ILO][,IHI][,INFO])
ORGHR(A,TAU[,ILO][,IHI][,INFO])
ORMHR(A,TAU,C[,ILO][,IHI][,SIDE][,TRANS][,INFO])
UNGHR(A,TAU[,ILO][,IHI][,INFO])
UNMHR(A,TAU,C[,ILO][,IHI][,SIDE][,TRANS][,INFO])
GEBAL(A[,SCALE][,ILO][,IHI][,JOB][,INFO])
GEBAK(V,SCALE[,ILO][,IHI][,JOB][,SIDE][,INFO])
HSEQR(H,WR,WI[,ILO][,IHI][,Z][,JOB][,COMPZ][,INFO])
HSEQR(H,W[,ILO][,IHI][,Z][,JOB][,COMPZ][,INFO])
HSEIN(H,WR,WI,SELECT[,VL][,VR][,IFAILL][,IFAILR][,INITV][,EIGSRC][,M][,INFO])
HSEIN(H,W,SELECT[,VL][,VR][,IFAILL][,IFAILR][,INITV][,EIGSRC][,M][,INFO])
TREVC(T[,HOWMNY][,SELECT][,VL][,VR][,M][,INFO])
TRSNA(T[,S][,SEP][,VL][,VR][,SELECT][,M][,INFO])
TREXC(T,IFST,ILST[,Q][,INFO])
TRSEN(T,SELECT[,WR][,WI][,M][,S][,SEP][,Q][,INFO])
TRSEN(T,SELECT[,W][,M][,S][,SEP][,Q][,INFO])

```

TRSYL(A,B,C,SCALE[,TRANA][,TRANB][,ISGN][,INFO])
 GGHRD(A,B[,ILO][,IHI][,Q][,Z][,COMPQ][,COMPZ][,INFO])
 GGBAL(A,B[,ILO][,IHI][,LSCALE][,RSCALE][,JOB][,INFO])
 GGBAK(V[,ILO][,IHI][,LSCALE][,RSCALE][,JOB][,INFO])
 HGEQZ(H,T[,ILO][,IHI][,ALPHAR][,ALPHAI][,BETA][,Q][,Z][,JOB][,COMPQ][,COMPZ][,INFO])
 HGEQZ(H,T[,ILO][,IHI][,ALPHA][,BETA][,Q][,Z][,JOB][,COMPQ][,COMPZ][,INFO])
 TGEVC(S,P[,HOWMNY][,SELECT][,VL][,VR][,M][,INFO])
 TGEXC(A,B[,IFST][,ILST][,Z][,Q][,INFO])
 TGSN(A,B,SELECT[,ALPHAR][,ALPHAI][,BETA][,IJOB][,Q][,Z][,PL][,PR][,DIF][,M][,INFO])
 TGSN(A,B,SELECT[,ALPHA][,BETA][,IJOB][,Q][,Z][,PL][,PR][,DIF][,M][,INFO])
 TGSYL(A,B,C,D,E,F[,IJOB][,TRANS][,SCALE][,DIF][,INFO])
 TGSNA(A,B[,S][,DIF][,VL][,VR][,SELECT][,M][,INFO])
 GGSVP(A,B,TOLA,TOLB[,K][,L][,U][,V][,Q][,INFO])
 TGSJA(A,B,TOLA,TOLB,K,L[,U][,V][,Q][,JOBV][,JOBQ][,ALPHA][,BETA][,NCYCLE][,INFO])

Interfaces of New Functionality

```
GETRF(A[,IPIV][,INFO])
```

```
!   Interface netlib95 exists, parameters:
```

```
!   netlib: (A,IPIV,RCOND,NORM,INFO)
```

```
!   Different number for parameter, netlib: 5, mkl: 3
```

```
!   Different order for parameter INFO, netlib: 5, mkl: 3
```

```
!   Absent mkl parameter: NORM
```

```
!   Absent mkl parameter: RCOND
```

```
GBTRF(A[,KL][,M][,IPIV][,INFO])
```

```
!   Interface netlib95 exists, parameters:
```

```
!   netlib: (A,K,M,IPIV,RCOND,NORM,INFO)
```

```
!   Different number for parameter, netlib: 7, mkl: 5
```

```
!   Different order for parameter INFO, netlib: 7, mkl: 5
```

```
!   Absent mkl parameter: NORM
```

```
!   Replace parameter name: netlib: K: mkl: KL
```

```
!   Absent mkl parameter: RCOND
```

```
POTRF(A[,UPLO][,INFO])
```

```
!   Interface netlib95 exists, parameters:
```

```
!   netlib: (A,UPLO,RCOND,NORM,INFO)
```

```
!   Different number for parameter, netlib: 5, mkl: 3
```

```
!   Different order for parameter INFO, netlib: 5, mkl: 3
```

```
!   Absent mkl parameter: NORM
```

```
!   Absent mkl parameter: RCOND
```

Optimization Solvers Basics

Classical optimization methods are linear methods based on calculation of a direction and search for the best point in the direction of the chosen gradient. The direction is calculated for each iteration. Examples of these methods are Conjugate Gradients and Quickest Descent. The search method usually requires calculation of the objective function approximation in a neighborhood of the current point.

Optimization problems generally have three basic components:

- An *objective function* that needs to be minimized or maximized. For instance, in a manufacturing process, you might want to maximize the profit or minimize the cost. In fitting experimental data to a user-defined model, the total deviation of observed data can be minimized from predictions based on the model.
- A set of *unknowns* or *variables* that affect the value of the objective function. In the manufacturing problem, the variables might include the amount of resources used or the time spent on each activity. In fitting-the-data problem, the unknowns are the parameters that define the model.
- A set of *constraints* that allow unknowns to take on certain values but exclude others. In the manufacturing problem, it does not make sense to spend a negative amount of time on any activity, so all the "time" variables are constrained to be non-negative.

The overall goal is to *find values of the variables that minimize or maximize the objective function while meeting the constraints*.

Intel® MKL provides math optimization tools, namely the optimized Trust-Region (TR) solvers for nonlinear least squares problems with or without linear constraints.

Nonlinear Least Squares Problem

Nonlinear least squares problem can be described as follows:

$$\min_{x \in \mathbb{R}^n} F(x) = \|y - f(x)\|_2^2, y \in \mathbb{R}^m, x \in \mathbb{R}^n, f: \mathbb{R}^n \rightarrow \mathbb{R}^m, m > n,$$

where $F(x) : \mathbb{R}^n \rightarrow \mathbb{R}$ is a twice differentiable function in $\mathbb{R}^n$. Solving of nonlinear least squares problem is searching for the best approximation to vector y with the model function that has nonlinear dependence on variables x . Best approximation means that the sum of squares of residuals is the lowest possible.

If

$$F(x) = \sum_{i=1}^m r_i(x)^2, \quad r_i(x) = y_i - f_i(x),$$

then

$$F(x) = \frac{1}{2} R(x)^T R(x),$$

where $R(x) = \{ r_i(x) \}, i = 1, \dots, m$.

Trust-Region Algorithm

The Trust-Region (TR) algorithms are relatively new iterative methods for solving nonlinear optimization problems. They are widely used in power engineering, finance, applied mathematics, physics, computer science, economics, sociology, biology, medicine, mechanical engineering, chemistry, and other areas. The TR methods have global convergence and local super convergence, which differentiates them from line search methods and Newton methods. The TR methods have better convergence when compared with widely-used Newton-like methods.

The main idea behind a TR algorithm is calculating a trial step and checking if the next values of x_+ belong to the *trust region* [Conn00]. Calculation of the trial step is strongly associated with the approximation model.

If the nonlinear least squares problem is defined as

$$\min_{x \in \mathbb{R}^n} F(x) = \|y - f(x)\|_2^2, \quad y \in \mathbb{R}^m, \quad x \in \mathbb{R}^n, \quad f: \mathbb{R}^n \rightarrow \mathbb{R}^m, \quad m > n,$$

then the trial step is a solution of the following subproblem:

$$\min_{d \in \mathbb{R}^n} \left\| F(x_k) + J(x_k)^T d \right\|_2^2, \text{ where } \|d\|_2^2 < \Delta_k \text{ and } J(x_k) \text{ is Jacobi matrix.}$$

This operation is an approximation to the objective function in a neighborhood of the current point x .

Consider a TR algorithm with **linear (bound) constraints** :

$$\min_{l \leq x \leq u} f(x) = \min_{l \leq x \leq u} \frac{1}{2} \|F(x)\|^2.$$

The first-order necessary condition for the vector to be minimized with this constraint is

$$D(x)^{-2} \nabla f(x) = D(x)^{-2} F'^T(x) F(x) = 0,$$

where D is the diagonal scaling matrix

$$D(x) = \text{diag}(|v_1^{-\frac{1}{2}}(x)|, |v_2^{-\frac{1}{2}}(x)|, \dots, |v_n^{-\frac{1}{2}}(x)|)$$

with

$$v_i(x) = \begin{cases} x_i - u_i & \text{if } \nabla f(x)_i < 0 \text{ and } u_i < \infty \\ x_i - l_i & \text{if } \nabla f(x)_i > 0 \text{ and } l_i < \infty \\ \min(x_i - u_i, u_i - x_i) & \text{if } \nabla f(x)_i = 0 \text{ and } u_i < \infty \text{ or } l_i > -\infty \\ -1 & \text{if } \nabla f(x)_i < 0 \text{ and } u_i = \infty \\ 1 & \text{if } \nabla f(x)_i < 0 \text{ and } u_i = \infty \\ 1 & \text{if } \nabla f(x)_i = 0 \text{ and } u_i = -l_i = \infty \end{cases}$$

for $i = 1, \dots, n$.

Provided $\Omega = \{x \in R^n: l \leq x \leq u\} \subset R^n$, write $\text{int}(\Omega)$ for the strictly nonempty interior of Ω . At each iteration, the basic structure of the method involves solving an elliptical trust-region subproblem and computing a search step to update the current iteration.

Provided $x_i \in \text{int}(\Omega)$ and the trust region size $\Delta_k > 0$, consider the elliptical trust-region subproblem

$$\min_p \left\{ F(x_k) + J(x_k)^T p : \|D_k p\| \leq \Delta_k \right\}.$$

Let $p_{tr}(\Delta_k)$ and $p_c(\Delta_k)$ be a solution and a Cauchy point of the above trust-region subproblem respectively [Conn00]. The following linear combination of two vectors $p_{tr}(\Delta_k)$ and $p_c(\Delta_k)$ defines the search step (trial step) used in the algorithm, that is,

$$p(\Delta_k) = t \bar{p}_c(\Delta_k) + (1-t) \bar{p}_{tr}(\Delta_k).$$

A good agreement between the model function m_k and the objective function f is ensured for the following standard condition:

$$\rho_f(p(\Delta_k)) = \frac{f(x_k) - f(x_k + p(\Delta_k))}{m_k(0) - m_k(p(\Delta_k))} \geq \beta_2$$

with the given constant $\beta_2 \in (0, 1)$. If this condition is not met, reject $p(\Delta_k)$, and the trust-region size Δ_k is successively reduced so that $p(\Delta_k)$ satisfies the accuracy requirement.

Bibliography

For more information about the BLAS, Sparse BLAS, LAPACK, ScaLAPACK, Sparse Solver, Interval Solver, VML, VSL, and FFT functionality, refer to the following publications:

- **BLAS Level 1**

C. Lawson, R. Hanson, D. Kincaid, and F. Krough. *Basic Linear Algebra Subprograms for Fortran Usage*, ACM Transactions on Mathematical Software, Vol.5, No.3 (September 1979) 308-325.

- **BLAS Level 2**

J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. *An Extended Set of Fortran Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, Vol.14, No.1 (March 1988) 1-32.

- **BLAS Level 3**

J. Dongarra, J. DuCroz, I. Duff, and S. Hammarling. *A Set of Level 3 Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software (December 1989).

- **Sparse BLAS**

D. Dodson, R. Grimes, and J. Lewis. *Sparse Extensions to the FORTRAN Basic Linear Algebra Subprograms*, ACM Transactions on Math Software, Vol.17, No.2 (June 1991).

D. Dodson, R. Grimes, and J. Lewis. *Algorithm 692: Model Implementation and Test Package for the Sparse Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, Vol.17, No.2 (June 1991).

[Duff86] I.S.Duff, A.M.Erisman, and J.K.Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford, UK, 1986.

[CXML01] *Compaq Extended Math Library*. Reference Guide, Oct.2001.

[Rem05] K.Remington. *A NIST FORTRAN Sparse Blas User's Guide*. (available on <http://math.nist.gov/~KRemington/fspblas/>)

[Saad94] Y.Saad. *SPARSKIT: A Basic Tool-kit for Sparse Matrix Computation*. Version 2, 1994. (<http://www.cs.umn.edu/~saad>)

[Saad96] Y.Saad. *Iterative Methods for Linear Systems*. PWS Publishing, Boston, 1996.

- **LAPACK**

[LUG] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*, Third Edition, Society for Industrial and Applied Mathematics (SIAM), 1999.

[Golub96] G. Golub and C. Van Loan. *Matrix Computations*, Johns Hopkins University Press, Baltimore, third edition, 1996.

[Bischof92] <http://citeseer.ist.psu.edu/bischof92framework.html>

- [Marques06] O.Marques, E.J.Riedy, and Ch.Voemel. *Benefits of IEEE-754 Features in Modern Symmetric Tridiagonal Eigensolvers*, SIAM Journal on Scientific Computing, Vol.28, No.5, 2006. (Tech report version in LAPACK Working Note 172 with the same title.)
- [Kahan66] W. Kahan. *Accurate Eigenvalues of a Symmetric Tridiagonal Matrix*, Report CS41, Computer Science Dept., Stanford University, July 21, 1966.
- [Dhillon04] I. Dhillon, B. Parlett. *Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices*, Linear Algebra and its Applications, 387(1), pp. 1-28, August 2004.
- [Dhillon04-02] I. Dhillon, B. Parlett. *Orthogonal Eigenvectors and * Relative Gaps*, SIAM Journal on Matrix Analysis and Applications, Vol. 25, 2004. (Also LAPACK Working Note 154.)
- [Dhillon97] I. Dhillon. *A new $O(n^2)$ algorithm for the symmetric tridiagonal eigenvalue/eigenvector problem*, Computer Science Division Technical Report No. UCB/CSD-97-971, UC Berkeley, May 1997.
- **ScaLAPACK**
 - [SLUG] L. Blackford, J. Choi, A.Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K.Stanley, D. Walker, and R. Whaley. *ScaLAPACK Users' Guide*, Society for Industrial and Applied Mathematics (SIAM), 1997.
- **Sparse Solver**
 - [Duff99] I. S. Duff and J. Koster. *The Design and Use of Algorithms for Permuting Large Entries to the Diagonal of Sparse Matrices*. SIAM J. Matrix Analysis and Applications, 20(4):889-901, 1999.
 - [Dong95] J. Dongarra, V.Eijkhout, A.Kalhan. *Reverse Communication Interface for Linear Algebra Templates for Iterative Methods*. UT-CS-95-291, May 1995. <http://www.netlib.org/lapack/lawnspdf/lawn99.pdf>
 - [Karypis98] G. Karypis and V. Kumar. *A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs*. SIAM Journal on Scientific Computing, 20(1):359-392, 1998.
 - [Li99] X.S. Li and J.W. Demmel. *A Scalable Sparse Direct Solver Using Static Pivoting*. In Proceeding of the 9th SIAM conference on Parallel Processing for Scientific Computing, San Antonio, Texas, March 22-34,1999.
 - [Liu85] J.W.H. Liu. *Modification of the Minimum-Degree algorithm by multiple elimination*. ACM Transactions on Mathematical Software, 11(2):141-153, 1985.

- [Menon98] R. Menon L. Dagnum. *OpenMP: An Industry-Standard API for Shared-Memory Programming*. IEEE Computational Science & Engineering, 1:46-55, 1998. <http://www.openmp.org>.
- [Saad03] Y. Saad. *Iterative Methods for Sparse Linear Systems*. 2nd edition, SIAM, Philadelphia, PA, 2003.
- [Schenk00] O. Schenk. *Scalable Parallel Sparse LU Factorization Methods on Shared Memory Multiprocessors*. PhD thesis, ETH Zurich, 2000.
- [Schenk00-2] O. Schenk, K. Gartner, and W. Fichtner. *Efficient Sparse LU Factorization with Left-right Looking Strategy on Shared Memory Multiprocessors*. BIT, 40(1):158-176, 2000.
- [Schenk01] O. Schenk and K. Gartner. *Sparse Factorization with Two-Level Scheduling in PARDISO*. In Proceeding of the 10th SIAM conference on Parallel Processing for Scientific Computing, Portsmouth, Virginia, March 12-14, 2001.
- [Schenk02] O. Schenk and K. Gartner. *Two-level scheduling in PARDISO: Improved Scalability on Shared Memory Multiprocessing Systems*. Parallel Computing, 28:187-197, 2002.
- [Schenk03] O. Schenk and K. Gartner. *Solving Unsymmetric Sparse Systems of Linear Equations with PARDISO*. Journal of Future Generation Computer Systems, 20(3):475-487, 2004.
- [Schenk04] O. Schenk and K. Gartner. *On Fast Factorization Pivoting Methods for Sparse Symmetric Indefinite Systems*. Technical Report, Department of Computer Science, University of Basel, 2004, submitted.
- [Sonn89] P. Sonneveld. *CGS, a Fast Lanczos-Type Solver for Nonsymmetric Linear Systems*. SIAM Journal on Scientific and Statistical Computing, 10:36-52, 1989.
- [Young71] D.M.Young. *Iterative Solution of Large Linear Systems*. New York, Academic Press, Inc., 1971.
- **VSL**
- [VSL Notes] Document included with Intel® MKL product (file name `vslnotes.pdf`).
- [Bratley87] Bratley P., Fox B.L., and Schrage L.E. *A Guide to Simulation*. 2nd edition. Springer-Verlag, New York, 1987.
- [Bratley88] Bratley P. and Fox B.L. *Implementing Sobol's Quasirandom Sequence Generator*, ACM Transactions on Mathematical Software, Vol. 14, No. 1, Pages 88-100, March 1988.
- [Bratley92] Bratley P., Fox B.L., and Niederreiter H. *Implementation and Tests of Low-Discrepancy Sequences*, ACM Transactions on Modeling and Computer Simulation, Vol. 2, No. 3, Pages 195-213, July 1992.

- [Coddington94] Coddington, P. D. *Analysis of Random Number Generators Using Monte Carlo Simulation*. Int. J. Mod. Phys. C-5, 547, 1994.
- [Gentle98] Gentle, James E. *Random Number Generation and Monte Carlo Methods*, Springer-Verlag New York, Inc., 1998.
- [L'Ecuyer94] L'Ecuyer, Pierre. *Uniform Random Number Generation*. Annals of Operations Research, 53, 77-120, 1994.
- [L'Ecuyer99] L'Ecuyer, Pierre. *Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure*. Mathematics of Computation, 68, 225, 249-260, 1999.
- [L'Ecuyer99a] L'Ecuyer, Pierre. *Good Parameter Sets for Combined Multiple Recursive Random Number Generators*. Operations Research, 47, 1, 159-164, 1999.
- [L'Ecuyer01] L'Ecuyer, Pierre. *Software for Uniform Random Number Generation: Distinguishing the Good and the Bad*. Proceedings of the 2001 Winter Simulation Conference, IEEE Press, 95-105, Dec. 2001.
- [Kirkpatrick81] Kirkpatrick, S., and Stoll, E. *A Very Fast Shift-Register Sequence Random Number Generator*. Journal of Computational Physics, V. 40. 517-526, 1981.
- [Knuth81] Knuth, Donald E. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. 2nd edition, Addison-Wesley Publishing Company, Reading, Massachusetts, 1981.
- [Matsumoto98] Matsumoto, M., and Nishimura, T. *Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator*, ACM Transactions on Modeling and Computer Simulation, Vol. 8, No. 1, Pages 3-30, January 1998.
- [Matsumoto00] Matsumoto, M., and Nishimura, T. *Dynamic Creation of Pseudorandom Number Generators*, 56-69, in: Monte Carlo and Quasi-Monte Carlo Methods 1998, Ed. Niederreiter, H. and Spanier, J., Springer 2000, <http://www.math.sci.hiroshima-u.ac.jp/%7Emat/MT/DC/dc.html>.
- [NAG] NAG Numerical Libraries.
http://www.nag.co.uk/numeric/numerical_libraries.asp
- [Sobol76] Sobol, I.M., and Levitan, Yu.L. *The production of points uniformly distributed in a multidimensional cube*. Preprint 40, Institute of Applied Mathematics, USSR Academy of Sciences, 1976 (In Russian).

- **FFT**

- [1] E. Oran Brigham, *The Fast Fourier Transform and Its Applications*, Prentice Hall, New Jersey, 1988.
- [2] Athanasios Papoulis, *The Fourier Integral and its Applications*, 2nd edition, McGraw-Hill, New York, 1984.

[3] Ping Tak Peter Tang, *DFTI, a New API for DFT: Motivation, Design, and Rationale*, July 2002.

[4] Charles Van Loan, *Computational Frameworks for the Fast Fourier Transform*, SIAM, Philadelphia, 1992

- **VML**

J.M.Muller. *Elementary functions: algorithms and implementation*, Birkhauser Boston, 1997.

IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std 754-1985.

- **Interval Solver**

[Alefeld83] G. Alefeld and J. Herzberger, *Introduction to Interval Computations*. - Academic Press, New York, 1983.

[Bentbib02] A.H.Bentbib, *Solving the full rank interval least squares problem // Applied Numerical Mathematics*. - 2002. - Vol. 41. - P. 283-294.

[Bliek92] Ch. Bliek, *Computer methods for design automation, Ph.D. Thesis*. - Dept. of Ocean Engineering, Massachusetts Institute of Technology, 1992.

[Hammer95] R. Hammer, M. Hocks, U. Kulisch, D. Ratz, *C++ Toolbox for Verified Computing I. Basic Numerical Problems*. - Berlin-Heidelberg: Springer, 1995.

[Hansen92] E. Hansen, Bounding the solution of interval linear equations // *SIAM Journal on Numerical Analysis*. - 1992. - Vol. 29, No. 5. - P. 1493-1503.

[Herzberger94] J. Herzberger, Iterative methods for the inclusion of the inverse of a matrix // *Topics in Validated Computations*, J. Herzberger, ed. - Amsterdam: Elsevier, 1994. -

[Jansson91] Ch.Jansson, Interval linear systems with symmetric matrices, skew-symmetric matrices, and dependencies in the right hand side // *Computing*. - 1991. - Vol. 46. - P. 265 - 274.

[Kearfott96] R.B. Kearfott, *Rigorous Global Search: Continuous Problems*. - Dordrecht, Kluwer, 1996.

[Kearfott] R.B. Kearfott, M.T. Nakao, A. Neumaier, S.M. Rump, S.P. Shary, P. van Hentenryck, Standardized notation in interval analysis. - An electronic version of the paper is accessible at <http://www.mat.univie.ac.at/~neum/software/int/>

[Kreinovich97] V. Kreinovich, A. Lakeyev, J. Rohn, P. Kahl, *Computational Complexity and Feasibility of Data Processing and Interval Computations*. - Kluwer, Dordrecht, 1997.

[Neumaier90] A. Neumaier, *Interval Methods for Systems of Equations*. - Cambridge, Cambridge University Press, 1990.

- [Neumaier99] A. Neumaier, A simple derivation of Hansen-Bliek-Rohn-Ning-Kearfott enclosure for linear interval equations // *Reliable Computing*. - 1999. - Vol. 5, No. 2. - P. 131-136.
- [Ning97] S. Ning, R.B. Kearfott, A comparison of some methods for solving linear interval equations // *SIAM Journal on Numerical Analysis*. - 1997. - Vol. 34, No. 4. - P. 1289-1305.
- [Rex99] G. Rex, J. Rohn, Sufficient conditions for regularity and singularity of interval matrices // *SIAM Journal on Numerical Analysis*. - 1999. - Vol. 20. - P. 437-445.
- [Rohn93] J. Rohn, Cheap and tight bounds: the recent result by E. Hansen can be made more efficient // *Interval Computations*. - 1993. - No. 4. - P. 13-21.
- [Rump83] S. M. Rump, Solving algebraic problems with high accuracy // *A New Approach to Scientific Computation*; Kulisch U. W. and Miranker W. L., eds. - New York: Academic Press, 1983. - P. 51-120.
- [Rump84] S. M. Rump, Solution of linear and nonlinear algebraic problems with sharp guaranteed bounds // *Computing Supplement*. - 1984. - Vol. 5. - P. 147-168.
- [Rump80] S. M. Rump, Kaucher E. Small bounds for the solution of systems of linear equations // *Computing Supplement*. - 1980. - Vol. 2. - P. 157-164.
- [Rump] S. Rump, INTLAB -- INTerval LABoratory. - 21 p. - An electronic version of the paper is accessible at <http://www.ti3.tu-harburg.de/rump/intlab/>.
- [Shary92] S.P. Shary, A new class of algorithms for optimal solution of interval linear systems // *Interval Computations*. - 1992, No. 2(4). - P. 18-29.
- [Shary95] S.P. Shary, On optimal solution of interval linear equations // *SIAM Journal on Numerical Analysis*. - 1995. - Vol. 32, No. 2. - P. 610-630.
- [Shary02] S. P. Shary, A new technique in systems analysis under interval uncertainty and ambiguity // *Reliable Computing*. - 2002. - Vol. 8. - 321-419.
- [Shary] S.P. Shary, A new class of methods for optimal enclosing solution sets to interval linear systems // *Journal of Computational Mathematics*. - to be published.

• Optimization Solvers

- [Conn00] A. R. Conn, N. I.M. Gould, P. L. Toint. *Trust-region Methods*. SIAM Society for Industrial & Applied Mathematics, Englewood Cliffs, New Jersey, MPS-SIAM Series on Optimization edition, 2000.
- [Dong95] J. Dongara, V. Eijkhout, A. Kalhan. *Reverse communication interface for linear algebra templates for iterative methods*. 1995.

For a reference implementation of BLAS, sparse BLAS, LAPACK, and ScaLAPACK packages (without platform-specific optimizations) visit www.netlib.org

Glossary

A^H	Denotes the conjugate transpose of a general matrix A . See <i>also</i> conjugate matrix.
A^T	Denotes the transpose of a general matrix A . See <i>also</i> transpose.
band matrix	A general m -by- n matrix A such that $a_{ij} = 0$ for $ i - j > 1$, where $1 < 1 < \min(m, n)$. For example, any tridiagonal matrix is a band matrix.
band storage	A special storage scheme for band matrices. A matrix is stored in a two-dimensional array: columns of the matrix are stored in the corresponding columns of the array, and <i>diagonals</i> of the matrix are stored in rows of the array.
BLAS	Abbreviation for Basic Linear Algebra Subprograms. These subprograms implement vector, matrix-vector, and matrix-matrix operations.
BRNG	Abbreviation for Basic Random Number Generator. Basic random number generators are pseudorandom number generators imitating i.i.d. random number sequences of uniform distribution. Distributions other than uniform are generated by applying different transformation techniques to the sequences of random numbers of uniform distribution.
BRNG registration	Standardized mechanism that allows a user to include a user-designed BRNG into the VSL and use it along with the predefined VSL basic generators.
Bunch-Kaufman factorization	Representation of a real symmetric or complex Hermitian matrix A in the form $A = PUDU^H P^T$ (or $A = PLDL^H P^T$) where P is a permutation matrix, U and L are upper and lower triangular matrices with unit diagonal, and D is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D .
c	When found as the first letter of routine names, c indicates the usage of single-precision complex data type.
CBLAS	C interface to the BLAS. See BLAS.
CDF	Cumulative Distribution Function. The function that determines probability distribution for univariate or multivariate random variable x . For univariate distribution the cumulative distribution function is the function of real argument x , which for every x takes a value equal to probability of the event $A: X \leq x$. For multivariate distribution the cumulative distribution function is the function of a real vector $x = (x_1, x_2, \dots, x_n)$, which, for every x , takes a value equal to probability of the event $A = (X_1 \leq x_1 \ \& \ X_2 \leq x_2, \ \& \ \dots, \ \& \ X_n \leq x_n)$.

Cholesky factorization	Representation of a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A in the form $A = U^H U$ or $A = L L^H$, where L is a lower triangular matrix and U is an upper triangular matrix.
condition number	The number $\kappa(A)$ defined for a given square matrix A as follows: $\kappa(A) = \ A\ \ A^{-1}\ $.
conjugate matrix	The matrix A^H defined for a given general matrix A as follows: $(A^H)_{ij} = (a_{ji})^*$.
conjugate number	The conjugate of a complex number $z = a + bi$ is $z^* = a - bi$.
d	When found as the first letter of routine names, d indicates the usage of double-precision real data type.
FFTs	Abbreviation for Fast Fourier Transforms. See Chapter 11 of this book.
dot product	The number denoted $x \cdot y$ and defined for given vectors x and y as follows: $x \cdot y = \sum_i x_i y_i$. Here x_i and y_i stand for the i -th elements of x and y , respectively.
double precision	A floating-point data type. On Intel® processors, this data type allows you to store real numbers x such that $2.23 \times 10^{-308} < x < 1.79 \times 10^{308}$. For this data type, the machine precision ϵ is approximately 10^{-15} , which means that double-precision numbers usually contain no more than 15 significant decimal digits. For more information, refer to <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture</i> .
eigenvalue	See eigenvalue problem.
eigenvalue problem	A problem of finding non-zero vectors x and numbers λ (for a given square matrix A) such that $Ax = \lambda x$. Here the numbers λ are called the eigenvalues of the matrix A and the vectors x are called the eigenvectors of the matrix A .
eigenvector	See eigenvalue problem.
elementary reflector(Householder matrix)	Matrix of a general form $H = I - \tau v v^T$, where v is a column vector and τ is a scalar. In LAPACK elementary reflectors are used, for example, to represent the matrix Q in the QR factorization (the matrix Q is represented as a product of elementary reflectors).
factorization	Representation of a matrix as a product of matrices. See also Bunch-Kaufman factorization, Cholesky factorization, LU factorization, LQ factorization, QR factorization, Schur factorization.
FFTs	Abbreviation for Fast Fourier Transforms. See Chapter 11 of this book.

full storage	A storage scheme allowing you to store matrices of any kind. A matrix A is stored in a two-dimensional array a , with the matrix element a_{ij} stored in the array element $a(i, j)$.
Hermitian matrix	A square matrix A that is equal to its conjugate matrix A^H . The conjugate A^H is defined as follows: $(A^H)_{ij} = (a_{ji})^*$.
I	See identity matrix.
identity matrix	A square matrix I whose diagonal elements are 1, and off-diagonal elements are 0. For any matrix A , $AI = A$ and $IA = A$.
i.i.d.	Independent Identically Distributed.
in-place	Qualifier of an operation. A function that performs its operation in-place takes its input from an array and returns its output to the same array.
Intel MKL	Abbreviation for Intel® Math Kernel Library.
inverse matrix	The matrix denoted as A^{-1} and defined for a given square matrix A as follows: $AA^{-1} = A^{-1}A = I$. A^{-1} does not exist for singular matrices A .
LQ factorization	Representation of an m -by- n matrix A as $A = LQ$ or $A = (L \ 0)Q$. Here Q is an n -by- n orthogonal (unitary) matrix. For $m \leq n$, L is an m -by- m lower triangular matrix with real diagonal elements; for $m > n$,

$$L = \begin{bmatrix} L_1 \\ L_2 \end{bmatrix}$$

where L_1 is an n -by- n lower triangular matrix, and L_2 is a rectangular matrix.

LU factorization	Representation of a general m -by- n matrix A as $A = PLU$, where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U is upper triangular (upper trapezoidal if $m < n$).
machine precision	The number ϵ determining the precision of the machine representation of real numbers. For Intel® architecture, the machine precision is approximately 10^{-7} for single-precision data, and approximately 10^{-15} for double-precision data. The precision also determines the number of significant decimal digits in the machine representation of real numbers. See <i>also</i> double precision and single precision.

MPI	Message Passing Interface. This standard defines the user interface and functionality for a wide range of message-passing capabilities in parallel computing.
MPICH	A freely available, portable implementation of MPI standard for message-passing libraries.
orthogonal matrix	A real square matrix A whose transpose and inverse are equal, that is, $A^T = A^{-1}$, and therefore $AA^T = A^TA = I$. All eigenvalues of an orthogonal matrix have the absolute value 1.
packed storage	A storage scheme allowing you to store symmetric, Hermitian, or triangular matrices more compactly. The upper or lower triangle of a matrix is packed by columns in a one-dimensional array.
PDF	Probability Density Function. The function that determines probability distribution for univariate or multivariate continuous random variable x . The probability density function $f(x)$ is closely related with the cumulative distribution function $F(x)$. For univariate distribution the relation is

$$F(x) = \int_{-\infty}^x f(t) dt .$$

For multivariate distribution the relation is

$$F(x_1, x_2, \dots, x_n) = \int_{-\infty}^{x_1} \int_{-\infty}^{x_2} \dots \int_{-\infty}^{x_n} f(t_1, t_2, \dots, t_n) dt_1 dt_2 \dots dt_n .$$

positive-definitematrix	A square matrix A such that $Ax \cdot x > 0$ for any non-zero vector x . Here $\cdot$ denotes the dot product.
pseudorandom number generator	A completely deterministic algorithm that imitates truly random sequences.
QR factorization	Representation of an m -by- n matrix A as $A = QR$, where Q is an m -by- m orthogonal (unitary) matrix, and R is n -by- n upper triangular with real diagonal elements (if $m \geq n$) or trapezoidal (if $m < n$) matrix.

random stream	An abstract source of independent identically distributed random numbers of uniform distribution. In this manual a random stream points to a structure that uniquely defines a random number sequence generated by a basic generator associated with a given random stream.
RNG	Abbreviation for Random Number Generator. In this manual the term "random number generators" stands for pseudorandom number generators, that is, generators based on completely deterministic algorithms imitating truly random sequences.
s	When found as the first letter of routine names, <i>s</i> indicates the usage of single-precision real data type.
ScaLAPACK	Stands for Scalable Linear Algebra PACKage.
Schur factorization	Representation of a square matrix A in the form $A = ZTZ^H$. Here T is an upper quasi-triangular matrix (for complex A , triangular matrix) called the Schur form of A ; the matrix Z is orthogonal (for complex A , unitary). Columns of Z are called Schur vectors.
single precision	A floating-point data type. On Intel® processors, this data type allows you to store real numbers x such that $1.18 \times 10^{-38} < x < 3.40 \times 10^{38}$. For this data type, the machine precision (ϵ) is approximately 10^{-7} , which means that single-precision numbers usually contain no more than 7 significant decimal digits. For more information, refer to <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture</i> .
singular matrix	A matrix whose determinant is zero. If A is a singular matrix, the inverse A^{-1} does not exist, and the system of equations $Ax = b$ does not have a unique solution (that is, there exist no solutions or an infinite number of solutions).
singular value	The numbers defined for a given general matrix A as the eigenvalues of the matrix AA^H . See also SVD.
SMP	Abbreviation for Symmetric MultiProcessing. The MKL offers performance gains through parallelism provided by the SMP feature.
sparse BLAS	Routines performing basic vector operations on sparse vectors. Sparse BLAS routines take advantage of vectors' sparsity: they allow you to store only non-zero elements of vectors. See BLAS.
sparse vectors	Vectors in which most of the components are zeros.
storage scheme	The way of storing matrices. See full storage, packed storage, and band storage.
SVD	Abbreviation for Singular Value Decomposition. See also Singular value decomposition section in Chapter 5.
symmetric matrix	A square matrix A such that $a_{ij} = a_{ji}$.

transpose	The transpose of a given matrix A is a matrix A^T such that $(A^T)_{ij} = a_{ji}$ (rows of A become columns of A^T , and columns of A become rows of A^T).
trapezoidal matrix	A matrix A such that $A = (A_1 A_2)$, where A_1 is an upper triangular matrix, A_2 is a rectangular matrix.
triangular matrix	A matrix A is called an upper (lower) triangular matrix if all its subdiagonal elements (superdiagonal elements) are zeros. Thus, for an upper triangular matrix $a_{ij} = 0$ when $i > j$; for a lower triangular matrix $a_{ij} = 0$ when $i < j$.
tridiagonal matrix	A matrix whose non-zero elements are in three diagonals only: the leading diagonal, the first subdiagonal, and the first super-diagonal.
unitary matrix	A complex square matrix A whose conjugate and inverse are equal, that is, that is, $A^H = A^{-1}$, and therefore $AA^H = A^H A = I$. All eigenvalues of a unitary matrix have the absolute value 1.
VML	Abbreviation for Vector Mathematical Library. See Chapter 9 of this book.
VSL	Abbreviation for Vector Statistical Library. See Chapter 10 of this book.
z	When found as the first letter of routine names, z indicates the usage of double-precision complex data type.

Index

?_backward_trig_transform 2646
?_commit_Helmholtz_2D 2673
?_commit_Helmholtz_3D 2673
?_commit_sph_np 2689
?_commit_sph_p 2689
?_commit_trig_transform 2641
?_forward_trig_transform 2644
?_Helmholtz_2D 2679
?_Helmholtz_3D 2679
?_init_Helmholtz_2D 2670
?_init_Helmholtz_3D 2670
?_init_sph_np 2686
?_init_sph_p 2686
?_init_trig_transform 2640
?_sph_np 2691
?_sph_p 2691
?asum 64
?axpy 65
?axpyi 192
?bdsdc 728
?bdsqr 724
?ConvExec 2498
?ConvExec1D 2500
?ConvExecX 2503
?ConvExecX1D 2506
?ConvNewTask 2479
?ConvNewTask1D 2481
?ConvNewTaskX 2483
?copy 67
?CorrExec 2498
?CorrExec1D 2500
?CorrExecX 2503
?CorrExecX1D 2506
?CorrNewTask 2479
?CorrNewTask1D 2481
?CorrNewTaskX 2483
?CorrNewTaskX1D 2486
?dbtf2 2145
?dbtrf 2147
?disna 809
?dot 68
?dotc 71
?dotci 195
?doti 194
?dotu 73
?dotui 197
?dttrf 2149
?dttrsv 2150
?gamn2d 2812
?gamx2d 2810
?gbbrd 706
?gbcon 419
?gbequ 514
?gbmv 91
?gbrfs 454
?gbsv 536
?gbsvx 538
?gbtf2 1255
?gbtrf 354
?gbtrs 382
?gebak 853
?gebal 849
?gebd2 1256
?gebr2d 2825
?gebrd 702
?gebs2d 2823
?gecon 417
?geequ 512

?gees 1072	?getrs 379
?geesx 1077	?ggbak 896
?geev 1084	?ggbal 894
?geevx 1089	?gges 1193
?gegas 2607	?ggesx 1200
?gegss 2613	?ggev 1209
?gehbs 2615	?ggevz 1215
?gehd2 1259	?ggglm 969
?gehrd 834	?gghrd 890
?gehss 2609	?gglse 965
?gekws 2611	?ggqrf 691
?gelq2 1261	?ggrqf 695
?gelqf 640	?ggsvd 1107
?gels 947	?ggsvp 934
?gelsd 960	?gsum2d 2814
?gelss 956	?gtcon 422
?gelsy 951	?gthr 198
?gemip 2630	?gthrz 200
?gemm 156	?gtrfs 457
?gemm3m 160	?gtsv 545
?gemv 95	?gtsvx 547
?gepps 2616	?gttrf 356
?gepss 2619	?gttrs 385
?geql2 1263	?gtts2 1272
?geqlf 655	?hbev 1034
?geqp3 625	?hbevd 1041
?geqpf 622	?hbevz 1050
?geqr2 1264	?hbgst 824
?geqrf 619	?hbgv 1170
?ger 98	?hbgvd 1177
?gerbr 2625	?hbgvx 1187
?gerc 100	?hbtrd 774
?gerfs 451	?hecon 436
?gerq2 1266	?heev 976
?gerqf 668	?heevd 983
?geru 102	?heevr 1003
?gerv2d 2820	?heevz 992
?gesc2 1268	?heft2 1610
?gesd2d 2818	?hegst 814
?gesdd 1102	?hegv 1118
?gesv 525	?hegvd 1125
?gesvd 1096	?hegvz 1136
?gesvr 2627	?hemm 164
?gesvx 529	?hemv 107
?geszi 2624	?her 110
?getc2 1269	?her2 112
?getf2 1271	?her2k 170
?getrf 352	?herdb 742
?getri 493	?herfs 475

?herk 167	?laeda 1315
?hesv 590	?laein 1317
?hesvx 593	?laesy 1243
?hetrd 750	?laev2 1320
?hetrf 370	?laexc 1322
?hetri 502	?lag2 1323
?hetrs 400	?lags2 1326
?hgeqz 899	?lagtf 1328
?hpcon 440	?lagtm 1330
?hpev 1012	?lagts 1332
?hpevd 1018	?lagv2 1334
?hpevx 1027	?lahef 1551
?hpgst 819	?lahqr 1336
?hpgv 1145	?lahr2 1342
?hpgvd 1152	?lahrd 1339
?hpgvx 1162	?laic1 1345
?hpmv 114	?laisnan 1274
?hpr 117	?laln2 1347
?hpr2 119	?lals0 1350
?hprfs 481	?lalsa 1354
?hpsv 605	?lalsd 1358
?hpsvx 607	?lamc1 1635
?hptrd 765	?lamc2 1636
?hptrf 376	?lamc3 1637
?hptri 506	?lamc4 1637
?hptrs 406	?lamc5 1638
?hsein 861	?lamch 1634
?hseqr 855	?lamrg 1361
?isnan 1274	?lamsh 2136
?labrd 1275	?laneg 1362
?lacgv 1240	?langb 1363
?lacn2 1278	?lange 1365
?lacon 1280	?langt 1366
?lacpy 1281	?lanhb 1371
?lacrm 1240	?lanhe 1379
?lact 1242	?lanhp 1375
?ladv 1283	?lanhs 1368
?lae2 1284	?lansb 1369
?laebz 1285	?lansp 1373
?laed0 1290	?lanst/?lanht 1376
?laed1 1293	?lansy 1378
?laed2 1295	?lantb 1381
?laed3 1298	?lantp 1383
?laed4 1300	?lantr 1385
?laed5 1302	?lanv2 1387
?laed6 1303	?lapll 1388
?laed7 1305	?lapmt 1389
?laed8 1309	?lapy2 1390
?laed9 1313	?lapy3 1391

?laqgb	1392	?lasd6	1506
?laqge	1394	?lasd7	1511
?laqhb	1396	?lasd8	1515
?laqp2	1398	?lasd9	1517
?laqps	1400	?lasda	1520
?laqr0	1402	?lasdq	1524
?laqr1	1406	?lasdt	1527
?laqr2	1408	?laset	1528
?laqr3	1412	?lasorte	2140
?laqr4	1416	?lasq1	1529
?laqr5	1420	?lasq2	1530
?laqsb	1424	?lasq3	1532
?laqsp	1426	?lasq4	1533
?laqsy	1428	?lasq5	1534
?laqtr	1429	?lasq6	1536
?lar1v	1432	?lasr	1537
?lar2v	1435	?lasrt	1541
?laref	2137	?lasrt2	2141
?larf	1437	?lassq	1542
?larfb	1439	?lasv2	1544
?larfg	1441	?laswp	1545
?larft	1443	?lasy2	1547
?larfx	1446	?lasyf	1549
?largv	1447	?latbs	1554
?larnv	1449	?latdf	1556
?larra	1450	?latps	1559
?larrb	1452	?latrd	1561
?larrc	1454	?latrs	1565
?larrd	1456	?latrz	1569
?larre	1460	?lauu2	1571
?larrf	1464	?lauum	1573
?larrj	1467	?lazq3	1574
?larrk	1469	?lazq4	1577
?larrl	1470	?nrm2	74
?larrv	1472	?opgtr	760
?lartg	1476	?opmtr	762
?lartv	1478	?org2l/?ung2l	1579
?laruv	1479	?org2r/?ung2r	1580
?larz	1480	?orgbr	709
?larzb	1482	?orghr	836
?larzt	1485	?orgl2/?ungl2	1582
?las2	1488	?orglq	644
?lascl	1489	?orgql	657
?lasd0	1491	?orgqr	629
?lasd1	1493	?org2/?ungr2	1584
?lasd2	1496	?orgrq	671
?lasd3	1500	?orgtr	745
?lasd4	1503	?orm2l/?unm2l	1585
?lasd5	1505	?orm2r/?unm2r	1588

?ormbr 713	?rotm 78
?ormhr 839	?rotmg 81
?orml2/?unml2 1590	?rscl 1603
?ormlq 646	?sbev 1032
?ormql 662	?sbevd 1037
?ormqr 632	?sbevz 1046
?ormr2/?unmr2 1593	?sbgst 821
?ormr3/?unmr3 1595	?sbgv 1167
?ormrq 676	?sbgvd 1173
?ormrz 684	?sbgvx 1182
?ormtr 747	?sbmv 121
?pbcon 429	?sbtrd 771
?pbequ 521	?scal 83
?pbrfs 466	?sctr 203
?pbstf 827	?sdot 70
?pbsv 568	?spcon 438
?pbsvx 570	?spev 1010
?pbt2 1598	?spevd 1014
?pbtrf 363	?spevx 1023
?pbtrs 393	?spgst 816
?pocon 425	?spgv 1142
?poequ 517	?spgvd 1148
?porfs 460	?spgvx 1157
?posv 552	?spm 125, 1246
?posvx 554	?spr 127, 1248
?pot2 1600	?spr2 129
?potrf 358	?sprfs 478
?potri 496	?spsv 598
?potrs 388	?spsvx 601
?ppcon 427	?sptd 758
?ppequ 519	?sptf 374
?pprfs 463	?sptri 504
?ppsv 560	?sptrs 403
?ppsvx 562	?stebz 802
?pptf 360	?stedc 787
?pptri 498	?stegr 793
?pptrs 390	?stein 806
?ptcon 432	?stemr 782
?pteqr 798	?steqr 778
?ptrfs 469	?steqr2 2154
?ptsv 576	?sterf 776
?ptsvx 578	?stev 1055
?pttrf 365	?stevd 1057
?pttrs 396	?stevr 1065
?pttrsv 2152	?stevx 1061
?ptts2 1601	?sum1 1254
?rot 75, 1244	?swap 85
?rotg 77	?sycon 434
?roti 201	?syev 973

?syevd 979	?trrf5 484
?syevr 997	?trrv2d 2821
?syevx 987	?trsd2d 2819
?sygs2/?hegs2 1604	?trsen 880
?sygst 812	?trsm 187
?sygv 1114	?trsna 872
?sygvd 1121	?trsv 153
?sygvx 1130	?trsyl 886
?symm 173	?trti2 1619
?symv 132, 1249	?trtri 2623
?syr 134, 1251	?trtri (LAPACK) 508
?syr2 136	?trtrs 2605
?syr2k 180	?trtrs (LAPACK) 408
?syldb 739	?tzrzf 682
?syarfs 472	?ungbr 716
?syrk 177	?unghr 843
?sysv 582	?unglq 649
?sysvx 585	?ungql 660
?sytd2/?hetd2 1606	?ungqr 635
?sytf2 1608	?ungrq 673
?sytrd 736	?ungtr 753
?sytrf 367	?unmbr 720
?sytri 500	?unmhr 846
?sytrs 398	?unmlq 652
?tbcon 448	?unmql 665
?tbmv 139	?unmqr 637
?tbsv 142	?unmrq 679
?tbtrs 414	?unmrz 687
?tgevc 907	?unmtr 755
?tgex2 1612	?upgtr 767
?tgexc 912	?upmtr 769
?tgsen 916	
?tgsja 939	1-norm value
?tgsna 928	complex Hermitian matrix
?tgsy2 1615	packed storage 1375
?tgsyl 923	complex Hermitian tridiagonal matrix 1376
?tpcon 445	complex symmetric matrix 1378
?tpmv 146	general rectangular matrix 1365, 2005
?tprfs 487	general tridiagonal matrix 1366
?tpsv 148	Hermitian band matrix 1371
?tptri 510	real symmetric matrix 1378, 2010
?tptrs 411	real symmetric tridiagonal matrix 1376
?trbr2d 2826	symmetric band matrix 1369
?trbs2d 2824	symmetric matrix
?trcon 443	packed storage 1373
?trevc 867	trapezoidal matrix 1385
?trexc 877	triangular band matrix 1381
?trmm 184	
?trmv 150	

1-norm value (*continued*)
 triangular matrix
 packed storage 1383
 upper Hessenberg matrix 1368, 2008

A

Abs 2272
absolute value of a vector element
 largest 86
 smallest 87
accuracy modes, in VML 2255
Acos 2307
Acosh 2319
Add 2263
adding magnitudes of the vector elements 64
arguments
 matrix 2895
 sparse vector 190
 vector 2893
array descriptor 1641
Asin 2309
Asinh 2321
Atan 2311
Atan2 2312
Atanh 2323
auxiliary routines
 LAPACK 1225
 ScaLAPACK 1945

B

balancing a matrix 849
band storage scheme 2895
basic quasi-number generator
 Niederreiter 2366
 Sobol 2366
basic random number generators 2357, 2365, 2366
 GFSR 2365
 MCG, 32-bit 2365
 MCG, 59-bit 2365
 Mersenne Twister
 MT19937 2365
 MT2203 2366
 MRG 2365
 Wichmann-Hill 2365
Bernoulli 2446

Beta 2438
bidiagonal matrix
 LAPACK 699
 ScaLAPACK 1840
Binomial 2450
bisection 1452
BLACS 1641, 2808, 2810, 2812, 2814, 2815, 2818,
 2819, 2820, 2821, 2822, 2823, 2824, 2825,
 2826, 2827, 2828, 2830, 2831, 2832, 2834,
 2835, 2836, 2837, 2838, 2839, 2840, 2841,
 2842
 broadcast 2822
 combines 2808
 destruction routines 2835
 informational routines 2838
 initialization routines 2827
 miscellaneous routines 2840
 point to point communication 2815
 ?gamn2d 2812
 ?gamx2d 2810
 ?gebr2d 2825
 ?gebs2d 2823
 ?gerv2d 2820
 ?gesd2d 2818
 ?gsum2d 2814
 ?trbr2d 2826
 ?trbs2d 2824
 ?trrv2d 2821
 ?trsd2d 2819
 blacs_abort 2837
 blacs_barrier 2841
 blacs_exit 2837
 blacs_freebuff 2836
 blacs_get 2830
 blacs_gridexit 2836
 blacs_gridinfo 2838
 blacs_gridinit 2832
 blacs_gridmap 2834
 blacs_pcoord 2840
 blacs_pinfo 2828
 blacs_pnum 2839
 blacs_set 2831
 blacs_setup 2828
 usage examples 2842
BLACS routines
 matrix shapes 2807
blacs_abort 2837
blacs_barrier 2841
blacs_exit 2837

blacs_freebuff 2836
 blacs_get 2830
 blacs_gridexit 2836
 blacs_gridinfo 2838
 blacs_gridinit 2832
 blacs_gridmap 2834
 blacs_pcoord 2840
 blacs_pinfo 2828
 blacs_pnum 2839
 blacs_set 2831
 blacs_setup 2828
 BLAS Code Examples 2901
 BLAS Level 1 functions
 ?asum 63, 64
 ?dot 63, 68
 ?dotc 63, 71
 ?dotu 63, 73
 ?nrm2 63, 74
 ?sdot 63, 70
 code example 2901
 dcabs1 63, 89
 i?amax 63, 86
 i?amin 63, 87
 BLAS Level 1 routines
 ?axpy 63, 65
 ?copy 63, 67
 ?rot 63, 75
 ?rotg 63, 77
 ?rotr 63, 78
 ?rotrm 63, 78
 ?rotrmg 63, 81
 ?rotrmq 63, 83
 ?scal 63, 83
 ?swap 63, 85
 code example 2902
 BLAS Level 2 routines
 ?gbmv 89, 91
 ?gemv 89, 95
 ?ger 89, 98
 ?gerc 89, 100
 ?geru 89, 102
 ?hbm 89, 104
 ?hemv 89, 107
 ?her 89, 110
 ?her2 89, 112
 ?hpmv 89, 114
 ?hpr 89, 117
 ?hpr2 89, 119
 ?srbmv 89, 121
 ?spmv 89, 125

BLAS Level 2 routines (*continued*)
 ?spr 89, 127
 ?spr2 89, 129
 ?symv 89, 132
 ?syr 89, 134
 ?syr2 89, 136
 ?tbrmv 89, 139
 ?tbsv 89, 142
 ?tpmv 89, 146
 ?tpsv 89, 148
 ?trmv 89, 150
 ?trsv 89, 153
 code example 2903
 BLAS Level 3 routines
 ?gemm 155, 156
 ?gemm3m 160
 ?hemm 155, 164
 ?her2k 155, 170
 ?herk 155, 167
 ?symm 155, 173
 ?syr2k 155, 180
 ?syrk 155, 177
 ?trmm 155, 184
 ?trsm 155, 187
 code example 2905
 BLAS routines
 routine groups 59
 block reflector
 general matrix
 LAPACK 1482
 ScaLAPACK 2047
 general rectangular matrix
 LAPACK 1439
 ScaLAPACK 2031
 triangular factor
 LAPACK 1443, 1485
 ScaLAPACK 2040, 2056
 block-cyclic distribution 1641
 block-splitting method 2366
 BRNG 2357, 2365
 Bunch-Kaufman factorization 351, 367, 370, 374, 376, 1645
 Hermitian matrix
 packed storage 376
 symmetric matrix
 packed storage 374

C

- Cauchy 2423
- CBLAS
 - arguments 3159
 - level 1 (vector operations) 3160
 - level 2 (matrix-vector operations) 3164
 - level 3 (matrix-matrix operations) 3172
 - sparse BLAS 3175
- CBLAS to the BLAS 3159
- Cbvt 2280
- Ceil 2331
- Cholesky factorization
 - Hermitian positive-definite matrix
 - band storage 363, 393, 570, 1655, 1670
 - packed storage 360, 562
 - split 827
 - symmetric positive-definite matrix
 - band storage 363, 393, 570, 1655, 1670
 - packed storage 360, 562
- CIS 2304
- clag2z 1620
- ClearErrorCallback 2355
- ClearErrStatus 2351
- code examples
 - BLAS Level 1 function 2901
 - BLAS Level 1 routine 2902
 - BLAS Level 2 routine 2903
 - BLAS Level 3 routine 2905
- CommitDescriptor 2531
- CommitDescriptorDM 2584
- communication subprograms 1641
- complex division in real arithmetic 1283
- complex Hermitian matrix
 - 1-norm value
 - LAPACK 1379
 - ScaLAPACK 2010
 - factorization with diagonal pivoting method 1610
 - Frobenius norm
 - LAPACK 1379
 - ScaLAPACK 2010
 - infinity- norm
 - LAPACK 1379
 - ScaLAPACK 2010
 - largest absolute value of element
 - LAPACK 1379
 - ScaLAPACK 2010
- complex Hermitian matrix in packed form
 - 1-norm value 1375
- complex Hermitian matrix in packed form (*continued*)
 - Frobenius norm 1375
 - infinity- norm 1375
 - largest absolute value of element 1375
- complex Hermitian tridiagonal matrix
 - 1-norm value 1376
 - Frobenius norm 1376
 - infinity- norm 1376
 - largest absolute value of element 1376
- complex matrix
 - complex elementary reflector
 - ScaLAPACK 2052
- complex symmetric matrix
 - 1-norm value 1378
 - Frobenius norm 1378
 - infinity- norm 1378
 - largest absolute value of element 1378
- complex vector
 - 1-norm using true absolute value
 - LAPACK 1254
 - ScaLAPACK 1955
 - conjugation
 - LAPACK 1240
 - ScaLAPACK 1951
- complex vector conjugation
 - LAPACK 1240
 - ScaLAPACK 1951
- compressed sparse vectors 190
- computational node 2359
- Computational Routines 617
- ComputeBackward 2539
- ComputeBackwardDM 2591
- ComputeForward 2536
- ComputeForwardDM 2587
- condition number
 - band matrix 419
 - general matrix
 - LAPACK 417
 - ScaLAPACK 1685, 1688, 1691
 - Hermitian matrix
 - packed storage 440
 - Hermitian positive-definite matrix
 - band storage 429
 - packed storage 427
 - tridiagonal 432
 - symmetric matrix
 - packed storage 438
 - symmetric positive-definite matrix
 - band storage 429

- condition number (*continued*)
 - symmetric positive-definite matrix (*continued*)
 - packed storage 427
 - tridiagonal 432
 - triangular matrix
 - band storage 448
 - packed storage 445
 - tridiagonal matrix 422
- configuration parameters, in FFT interface 2523
- Conj 2271
- Continuous Distribution Generators 2400
- Continuous Distributions 2401
- ConvCopyTask 2510
- ConvDeleteTask 2508
- converting a sparse vector into compressed storage
 - form 198, 200
 - and writing zeros to the original vector 200
- converting compressed sparse vectors into full storage
 - form 203
- ConvInternalPrecision 2492
- Convolution and Correlation 2470
- Convolution Functions
 - ?ConvExec 2498
 - ?ConvExec1D 2500
 - ?ConvExecX 2503
 - ?ConvExecX1D 2506
 - ?ConvNewTask 2479
 - ?ConvNewTask1D 2481
 - ?ConvNewTaskX 2483
 - ?ConvNewTaskX1D 2486
 - ConvCopyTask 2510
 - ConvDeleteTask 2508
 - ConvSetDecimation 2495
 - ConvSetInternalPrecision 2492
 - ConvSetMode 2490
 - ConvSetStart 2494
 - CorrCopyTask 2510
 - CorrDeleteTask 2508
- ConvSetMode 2490
- ConvSetStart 2494
- CopyDescriptor 2532
- copying
 - matrices
 - distributed 1993
 - global parallel 1995
 - local replicated 1995
 - two-dimensional
 - LAPACK 1281
 - ScaLAPACK 1997

- copying (*continued*)
 - vectors 67
- CopyStream 2384
- CopyStreamState 2385
- CorrCopyTask 2510
- CorrDeleteTask 2508
- Correlation Functions
 - ?CorrExec 2498
 - ?CorrExec1D 2500
 - ?CorrExecX 2503
 - ?CorrExecX1D 2506
 - ?CorrNewTask 2479
 - ?CorrNewTask1D 2481
 - ?CorrNewTaskX 2483
 - ?CorrNewTaskX1D 2486
 - CorrSetDecimation 2495
 - CorrSetInternalPrecision 2492
 - CorrSetMode 2490
 - CorrSetStart 2494
- CorrSetInternalDecimation 2495
- CorrSetInternalPrecision 2492
- CorrSetMode 2490
- CorrSetStart 2494
- Cos 2299
- Cosh 2314
- Cray 2156
- CreateDescriptor 2529
- CreateDescriptorDM 2582

D

- data type
 - in VML 2255
 - shorthand 57
- Data Types 2368
- dcabs1 89
- dcg_check 2219
- dcg_get 2223
- dcg_init 2218
- dcgmrhs_check 2225
- dcgmrhs_get 2229
- dcgmrhs_init 2224
- DeleteStream 2383
- descriptor configuration
 - cluster FFT 2582
 - FFT 2524
- descriptor manipulation
 - cluster FFT 2582

descriptor manipulation (*continued*)
 FFT 2524
dfgmres_check 2231
dfgmres_get 2235
dfgmres_init 2230
DFT routines
 descriptor configuration
 SetValue 2542
diagonal elements
 LAPACK 1528
 ScaLAPACK 2062
diagonally dominant-like banded matrix
 solving systems of linear equations 1679
diagonally dominant-like tridiagonal matrix
 solving systems of linear equations 1676
dimension 2893
Direct Sparse Solver (DSS) Interface Routines 2182
Discrete Distribution Generators 2400
Discrete Distributions 2441
Discrete Fourier Transform
 SetValue 2542
distributed-memory computations 1641
Distribution Generators 2399
Distribution Generators Supporting Accurate Mode 2401
djacobi 2768, 2777, 2803
 usage example 2777, 2803
djacobi_delete 2768
djacobi_init 2766
djacobi_solve 2767, 2770
 usage example 2770
dlag2s 1621
dNewAbstractStream 2377
dot product
 complex vectors, conjugated 71
 complex vectors, unconjugated 73
 real vectors 68
 real vectors (extended precision) 70
 sparse complex vectors 197
 sparse complex vectors, conjugated 195
 sparse real vectors 194
driver
 expert 1642
 simple 1642
Driver Routines 524, 946
dss_create 2186
dtrnlspl
 usage example 2730
dtrnlspl_delete 2729
dtrnlspl_get 2727

dtrnlspl_init 2724
dtrnlspl_solve 2725
dtrnlsplbc
 usage example 2750
dtrnlsplbc_delete 2749
dtrnlsplbc_get 2747
dtrnlsplbc_init 2744
dtrnlsplbc_solve 2746

E

eigenpairs, sorting 2140
eigenvalue problems 613, 731, 811, 829, 889, 1826,
 2142, 2154
 general matrix 829, 889, 1826
 generalized form 811
 Hermitian matrix 731
 symmetric matrix 731
 symmetric tridiagonal matrix 2142, 2154
eigenvalues
 eigenvalue problems 731
eigenvectors
 eigenvalue problems 731
elementary reflector
 complex matrix 2052
 general matrix 1480, 2043
 general rectangular matrix
 LAPACK 1437, 1446
 ScaLAPACK 2027, 2035
 LAPACK generation 1441
 ScaLAPACK generation 2038
Erf 2324
Erfc 2326
ErfInv 2328
error diagnostics, in VML 2260
error estimation for linear equations
 distributed tridiagonal coefficient matrix 1703
error handling
 pxerbla 2161, 2793
 xerbla 59, 1640, 2260
ErrorClass 2525
ErrorMessage 2527
errors in solutions of linear equations
 distributed tridiagonal coefficient matrix 1703
 general matrix
 band storage 454
 Hermitian matrix
 packed storage 481

errors in solutions of linear equations (*continued*)

- Hermitian positive-definite matrix
 - band storage 466
 - packed storage 463
- symmetric matrix
 - packed storage 478
- symmetric positive-definite matrix
 - band storage 466
 - packed storage 463
- triangular matrix
 - band storage 490
 - packed storage 487
- tridiagonal matrix 457
- Euclidean norm
 - of a vector 74
- Exp 2291
- expert driver 1642
- Expm1 2293
- Exponential 2413

F

factorization

- Bunch-Kaufman
 - LAPACK 351
 - ScaLAPACK 1645
- Cholesky
 - LAPACK 351, 1598, 1600
 - ScaLAPACK 2123
- diagonal pivoting
 - Hermitian matrix
 - complex 1610
 - packed 607
 - symmetric matrix
 - indefinite 1608
 - packed 601
- LU
 - LAPACK 351
 - ScaLAPACK 1645
- orthogonal
 - LAPACK 617
 - ScaLAPACK 1719
- partial
 - complex Hermitian indefinite matrix 1551
 - real/complex symmetric matrix 1549
- triangular factorization 351
- triangular factorization[factorization
aaa] 1645

factorization (*continued*)

- upper trapezoidal matrix 1569
- fast Fourier transform
 - CommitDescriptor 2531
 - CommitDescriptorDM 2584
 - ComputeBackward 2539
 - ComputeBackwardDM 2591
 - ComputeForwardDM 2587
 - CopyDescriptor 2532
 - CreateDescriptor 2529
 - CreateDescriptorDM 2582
 - ErrorClass 2525
 - ErrorMessage 2527
 - FreeDescriptor 2534
 - FreeDescriptorDM 2586
 - GetValue 2544
 - GetValueDM 2597
 - SetValueDM 2594
- fast Fourier Transform
 - ComputeForward 2536
- FFT computation
 - cluster FFT 2582
 - FFT 2524
- FFT functions
 - descriptor manipulation
 - CommitDescriptor 2531
 - CommitDescriptorDM 2584
 - CopyDescriptor 2532
 - CreateDescriptor 2529
 - CreateDescriptorDM 2582
 - FreeDescriptor 2534
 - FreeDescriptorDM 2586
 - DFT computation
 - ComputeBackward 2539
 - ComputeForward 2536
 - FFT computation
 - ComputeForwardDM 2587
 - status checking
 - ErrorClass 2525
 - ErrorMessage 2527
- FFT Interface 2523
- FFT routines
 - descriptor configuration
 - GetValue 2544
 - GetValueDM 2597
 - SetValueDM 2594
 - FFT computation
 - ComputeBackwardDM 2591
- fill-in, for sparse matrices 2864

finding
 index of the element of a vector with the largest absolute value of the real part 1952
 element of a vector with the largest absolute value 86
 element of a vector with the largest absolute value of the real part and its global index 1954
 element of a vector with the smallest absolute value 87
Floor 2330
font conventions 57
Fortran 95 interface conventions
 BLAS, Sparse BLAS 61
Fortran 95 interface conventions
 LAPACK 343
Fortran 95 Interfaces for LAPACK
 absent from Netlib 3187
 identical to Netlib 3180
 modified Netlib interfaces 3185
 new functionality 3193
 with replaced Netlib argument names 3183
Fortran 95 Interfaces for LAPACK Routines
 specific MKL features 3179
Fortran 95 LAPACK interface vs. Netlib 345
free_Helmholtz_2D 2685
free_Helmholtz_3D 2685
free_sph_np 2694
free_sph_p 2694
free_trig_transform 2648
FreeDescriptor 2534
FreeDescriptorDM 2586
Frobenius norm
 complex Hermitian matrix
 packed storage 1375
 complex Hermitian tridiagonal matrix 1376
 complex symmetric matrix 1378
 general rectangular matrix 1365, 2005
 general tridiagonal matrix 1366
 Hermitian band matrix 1371
 real symmetric matrix 1378, 2010
 real symmetric tridiagonal matrix 1376
 symmetric band matrix 1369
 symmetric matrix
 packed storage 1373
 trapezoidal matrix 1385
 triangular band matrix 1381
 triangular matrix
 packed storage 1383
 upper Hessenberg matrix 1368, 2008

full storage scheme 2895
full-storage vectors 190
function name conventions, in VML 2256

G

Gamma 2435
gathering sparse vector's elements into compressed form 198, 200
 and writing zeros to these elements 200
Gauss method, for interval systems 2607, 2631
Gauss-Seidel iteration, for interval systems 2613, 2631
Gaussian 2405
GaussianMV 2408
general matrix
 block reflector 1482, 2047
 eigenvalue problems 829, 889, 1826
 elementary reflector 1480, 2043
 estimating the condition number
 band storage 419
 inverting matrix
 LAPACK 493
 ScaLAPACK 1707
 LQ factorization 640, 1736
 LU factorization
 band storage 354, 1255, 1648, 1651, 2145, 2147
 matrix-vector product
 band storage 91
 multiplying by orthogonal matrix
 from LQ factorization 1590, 2106
 from QR factorization 1588, 2101
 from RQ factorization 1593, 2110
 from RZ factorization 1595
 multiplying by unitary matrix
 from LQ factorization 1590, 2106
 from QR factorization 1588, 2101
 from RQ factorization 1593, 2110
 from RZ factorization 1595
 QL factorization
 LAPACK 655
 ScaLAPACK 1750
 QR factorization
 with pivoting 622, 625, 1722
 rank-I update 98
 rank-I update, conjugated 100
 rank-I update, unconjugated 102
 reduction to bidiagonal form 1256, 1275, 1964

- general matrix (*continued*)
 - reduction to upper Hessenberg form 1968
 - RQ factorization
 - LAPACK 668
 - ScaLAPACK 1795
 - scalar-matrix-matrix product 156, 160
 - solving systems of linear equations
 - band storage
 - LAPACK 382
 - ScaLAPACK 1665
- general rectangular distributed matrix
 - computing scaling factors 1714
 - equilibration 1714
- general rectangular matrix
 - 1-norm value
 - LAPACK 1365
 - ScaLAPACK 2005
 - block reflector
 - LAPACK 1439
 - ScaLAPACK 2031
 - elementary reflector
 - LAPACK 1437, 2035
 - ScaLAPACK 2027
 - Frobenius norm
 - LAPACK 1365
 - ScaLAPACK 2005
 - infinity- norm
 - LAPACK 1365
 - ScaLAPACK 2005
 - largest absolute value of element
 - LAPACK 1365
 - ScaLAPACK 2005
 - LQ factorization
 - LAPACK 1261
 - ScaLAPACK 1971
 - multiplication
 - LAPACK 1489
 - ScaLAPACK 2060
 - QL factorization
 - LAPACK 1263
 - ScaLAPACK 1974
 - QR factorization
 - LAPACK 1264
 - ScaLAPACK 1977
 - reduction of first columns
 - LAPACK 1339, 1342
 - ScaLAPACK 2001
 - reduction to bidiagonal form 1985

- general rectangular matrix (*continued*)
 - row interchanges
 - LAPACK 1545
 - ScaLAPACK 2068
 - RQ factorization
 - LAPACK 1266
 - ScaLAPACK 1764, 1980
 - scaling 2019
- general square matrix
 - reduction to upper Hessenberg form 1259
 - trace 2070
- general triangular matrix
 - LU factorization
 - band storage 1956
- general tridiagonal matrix
 - 1-norm value 1366
 - Frobenius norm 1366
 - infinity- norm 1366
 - largest absolute value of element 1366
- general tridiagonal triangular matrix
 - LU factorization
 - band storage 1960
- generalized eigenvalue problems 811, 812, 814, 816, 819, 821, 824, 1604, 1606, 1856, 1858, 2127, 2130
 - complex Hermitian-definite problem
 - band storage 824
 - packed storage 819
 - real symmetric-definite problem
 - band storage 821
 - packed storage 816
 - See also LAPACK routines, generalized eigenvalue problems 811
- Generalized LLS Problems 964
- Generalized Nonsymmetric Eigenproblems 1192
- generalized Schur factorization 1334, 1435, 1447, 1449
- Generalized Singular Value Decomposition 933
- generalized Sylvester equation 923
- Generalized SymmetricDefinite Eigenproblems 1113
- generation methods 2359
- Geometric 2448
- GetBrngProperties 2465
- getcpuclocks 2796
- getcpufrequency 2797
- GetErrorCallBack 2354
- GetErrStatus 2350
- GetMode 2347
- GetNumRegBrngs 2398
- GetStreamStateBrng 2397

GetValue 2544
GetValueDM 2597
GFSR 2360
Givens rotation
 modified Givens transformation parameters 81
 of sparse vectors 201
 parameters 77
global array 1641
Gumbel 2432

H

Hansen-Blik-Rohn procedure, for interval systems 2615
Helmholtz problem
 three-dimensional 2663
 two-dimensional 2659
Helmholtz problem on a sphere
 non-periodic 2661
 periodic 2661
Hermitian band matrix
 1-norm value 1371
 Frobenius norm 1371
 infinity- norm 1371
 largest absolute value of element 1371
Hermitian matrix 104, 107, 110, 112, 114, 117, 119,
 164, 167, 170, 370, 376, 400, 406, 436,
 440, 502, 506, 731, 811, 1561, 1604, 1606,
 1911, 2021, 2071, 2127, 2130
 Bunch-Kaufman factorization
 packed storage 376
 eigenvalues and eigenvectors 1911
 estimating the condition number
 packed storage 440
 generalized eigenvalue problems 811
 inverting the matrix
 packed storage 506
 matrix-vector product
 band storage 104
 packed storage 114
 rank-1 update
 packed storage 117
 rank-2 update
 packed storage 119
 rank-2k update 170
 rank-n update 167
 reducing to standard form
 LAPACK 1604
 ScaLAPACK 2127
Hermitian matrix (*continued*)
 reducing to tridiagonal form
 LAPACK 1561, 1606
 ScaLAPACK 2071, 2130
 scalar-matrix-matrix product 164
 scaling 2021
 solving systems of linear equations
 packed storage 406
Hermitian positive definite distributed matrix
 computing scaling factors 1716
 equilibration 1716
Hermitian positive-definite band matrix
 Cholesky factorization 1598
Hermitian positive-definite distributed matrix
 inverting the matrix 1710
Hermitian positive-definite matrix
 Cholesky factorization
 band storage 363, 1655
 packed storage 360
 estimating the condition number
 band storage 429
 packed storage 427
 inverting the matrix
 packed storage 498
 solving systems of linear equations
 band storage 393, 1670
 packed storage 390
Hermitian positive-definite tridiagonal matrix
 solving systems of linear equations 1673
Householder matrix
 LAPACK 1441
 ScaLAPACK 2038
Householder method, for interval systems 2609, 2631
Householder reflector 2137
Hypergeometric 2452
Hypot 2289

I

i?amax 86
i?amin 87
i?max1 1253
IBM ESSL library 2470
IEEE arithmetic 2004
IEEE standard
 implementation 2157
 signbit position 2160
ilaenv 1626

- ilaver 1625
- ILU0 preconditioner 2238
- Incomplete LU Factorization Technique 2238
- increment 2893
- iNewAbstractStream 2375
- infinity-norm
 - complex Hermitian matrix
 - packed storage 1375
 - complex Hermitian tridiagonal matrix 1376
 - complex symmetric matrix 1378
 - general rectangular matrix 1365, 2005
 - general tridiagonal matrix 1366
 - Hermitian band matrix 1371
 - real symmetric matrix 1378, 2010
 - real symmetric tridiagonal matrix 1376
 - symmetric band matrix 1369
 - symmetric matrix
 - packed storage 1373
 - trapezoidal matrix 1385
 - triangular band matrix 1381
 - triangular matrix
 - packed storage 1383
 - upper Hessenberg matrix 1368, 2008
- Interface Consideration 208
- interval solver routines
 - ?gegas 2607
 - ?gegss 2613
 - ?gehbs 2615
 - ?gehss 2609
 - ?gekws 2611
 - ?gemip 2630
 - ?gepps 2616
 - ?gepps 2619
 - ?gerbr 2625
 - ?gesvr 2627
 - ?geszi 2624
 - ?trtri 2623
 - ?trtrs 2605
- Inv 2274
- InvCbrt 2281
- inverse matrix. inverting a matrix 493, 1707, 1710, 1712
- inverting a matrix
 - general matrix
 - LAPACK 493
 - ScaLAPACK 1707
 - Hermitian matrix
 - packed storage 506

- inverting a matrix (*continued*)
 - Hermitian positive-definite matrix
 - LAPACK 496
 - packed storage 498
 - ScaLAPACK 1710
 - symmetric matrix
 - packed storage 504
 - symmetric positive-definite matrix
 - LAPACK 496
 - packed storage 498
 - ScaLAPACK 1710
 - triangular distributed matrix 1712
 - triangular matrix
 - packed storage 510
- InvSqrt 2279
- iparmq 1629
- Iterative Sparse Solvers 2198
- Iterative Sparse Solvers based on Reverse Communication Interface (RCI ISS) 2198

J

- Jacobi matrix calculation routines 2765, 2766, 2767, 2768
 - djacobi 2768
 - djacobi_delete 2768
 - djacobi_init 2766
 - djacobi_solve 2767

K

- Krawczyk iteration method, for interval systems 2611

L

- LAPACK
 - naming conventions 342
- LAPACK routines
 - 2-by-2 generalized eigenvalue problem 1323
 - 2-by-2 Hermitian matrix
 - plane rotation 1435
 - 2-by-2 orthogonal matrices 1326
 - 2-by-2 real matrix
 - generalized Schur factorization 1334
 - 2-by-2 real nonsymmetric matrix
 - Schur factorization 1387

LAPACK routines (*continued*)

2-by-2 symmetric matrix
plane rotation 1435
2-by-2 triangular matrix
singular values 1488
SVD 1544
approximation to smallest eigenvalue 1533, 1577
auxiliary routines
?gbtf2 1255
?gebd2 1256
?gehd2 1259
?gelq2 1261
?geql2 1263
?geqr2 1264
?gerq2 1266
?gesc2 1268
?getc2 1269
?getf2 1271
?gtts2 1272
?hetf2 1610
?isnan 1274
?labrd 1275
?lacgv 1240
?lacn2 1278
?lacon 1280
?lacpy 1281
?lacrm 1240
?lact 1242
?ladiv 1283
?lae2 1284
?laebz 1285
?laed0 1290
?laed1 1293
?laed2 1295
?laed3 1298
?laed4 1300
?laed5 1302
?laed6 1303
?laed7 1305
?laed8 1309
?laed9 1313
?laeda 1315
?laein 1317
?laesy 1243
?laev2 1320
?laexc 1322
?lag2 1323
?lags2 1326
?lagtf 1328

LAPACK routines (*continued*)auxiliary routines (*continued*)

?lagtm 1330
?lagts 1332
?lagv2 1334
?lahef 1551
?lahqr 1336
?lahr2 1342
?lahrd 1339
?laic1 1345
?laisnan 1274
?laln2 1347
?lals0 1350
?lalsa 1354
?lalsd 1358
?lamrg 1361
?laneg 1362
?langb 1363
?lange 1365
?langt 1366
?lanhb 1371
?lanhe 1379
?lanhp 1375
?lanhs 1368
?lansb 1369
?lansp 1373
?lanst/?lanht 1376
?lansy 1378
?lantb 1381
?lantp 1383
?lantr 1385
?lanv2 1387
?lapll 1388
?lapmt 1389
?lapy2 1390
?lapy3 1391
?laqgb 1392
?laqge 1394
?laqhb 1396
?laqp2 1398
?laqps 1400
?laqr0 1402
?laqr1 1406
?laqr2 1408
?laqr3 1412
?laqr4 1416
?laqr5 1420
?laqsb 1424
?laqsp 1426

LAPACK routines (*continued*)

auxiliary routines (*continued*)

?laqsy 1428
 ?laqtr 1429
 ?lar1v 1432
 ?lar2v 1435
 ?larf 1437
 ?larfb 1439
 ?larfg 1441
 ?larft 1443
 ?larfx 1446
 ?largv 1447
 ?larnv 1449
 ?larra 1450
 ?larrrb 1452
 ?larrrc 1454
 ?larrrd 1456
 ?larrrf 1460
 ?larrrg 1464
 ?larrrj 1467
 ?larrrk 1469
 ?larrrr 1470
 ?larrrv 1472
 ?lartg 1476
 ?lartv 1478
 ?laruv 1479
 ?larz 1480
 ?larzb 1482
 ?larzt 1485
 ?las2 1488
 ?lascl 1489
 ?lasd0 1491
 ?lasd1 1493
 ?lasd2 1496
 ?lasd3 1500
 ?lasd4 1503
 ?lasd5 1505
 ?lasd6 1506
 ?lasd7 1511
 ?lasd8 1515
 ?lasd9 1517
 ?lasda 1520
 ?lasdq 1524
 ?lasdt 1527
 ?laset 1528
 ?lasq1 1529
 ?lasq2 1530
 ?lasq3 1532
 ?lasq4 1533

LAPACK routines (*continued*)

auxiliary routines (*continued*)

?lasq5 1534
 ?lasq6 1536
 ?lasr 1537
 ?lasrt 1541
 ?lassq 1542
 ?lasv2 1544
 ?laswp 1545
 ?lasy2 1547
 ?lasyf 1549
 ?latbs 1554
 ?latdf 1556
 ?latps 1559
 ?latrd 1561
 ?latrs 1565
 ?latrz 1569
 ?lauu2 1571
 ?lauum 1573
 ?lazq3 1574
 ?lazq4 1577
 ?org2l/?ung2l 1579
 ?org2r/?ung2r 1580
 ?orgl2l/?ungl2 1582
 ?orgr2/?ungr2 1584
 ?orm2l/?unm2l 1585
 ?orm2r/?unm2r 1588
 ?orml2/?unml2 1590
 ?ormr2/?unmr2 1593
 ?ormr3/?unmr3 1595
 ?pbt2 1598
 ?potf2 1600
 ?ptts2 1601
 ?rot 1244
 ?rscl 1603
 ?spmv 1246
 ?spr 1248
 ?sum1 1254
 ?sygs2/?hegs2 1604
 ?symv 1249
 ?syr 1251
 ?sytd2/?hetd2 1606
 ?sytf2 1608
 ?tgex2 1612
 ?tgsy2 1615
 ?trti2 1619
 clag2z 1620
 dlag2s 1621
 i?max1 1253

LAPACK routines (*continued*)

- auxiliary routines (*continued*)
 - slag2d 1622
 - zlag2c 1623
- bidiagonal divide and conquer 1527
- block reflector
 - triangular factor 1443, 1485
- checking for characters equality 1632
- checking for safe infinity 1631
- checking for strings equality 1632
- complex Hermitian matrix
 - packed storage 1375
- complex Hermitian tridiagonal matrix 1376
- complex matrix multiplication 1240
- complex symmetric matrix
 - computing eigenvalues and eigenvectors 1243
 - matrix-vector product 1249
 - symmetric rank-1 update 1251
- complex symmetric packed matrix
 - symmetric rank-1 update 1248
- complex vector
 - 1-norm using true absolute value 1254
 - index of element with max absolute value 1253
 - linear transformation 1242
 - matrix-vector product 1246
 - plane rotation 1244
- complex vector conjugation 1240
- condition number estimation
 - ?disna 809
 - ?gbcon 419
 - ?gecon 417
 - ?gtcon 422
 - ?hecon 436
 - ?hpcon 440
 - ?pbcon 429
 - ?pocon 425
 - ?ppcon 427
 - ?ptcon 432
 - ?spcon 438
 - ?sycon 434
 - ?tbcon 448
 - ?tpcon 445
 - ?trcon 443
- determining machine parameters 1635, 1636
- dqd transform 1536
- dqds transform 1534
- driver routines
 - generalized LLS problems
 - ?ggglm 969

LAPACK routines (*continued*)

- driver routines (*continued*)
 - generalized LLS problems (*continued*)
 - ?gglse 965
 - generalized nonsymmetric eigenproblems
 - ?gges 1193
 - ?ggesx 1200
 - ?ggeev 1209
 - ?ggevx 1215
 - generalized symmetric definite eigenproblems
 - ?hbgv 1170
 - ?hbgvd 1177
 - ?hbgvx 1187
 - ?hegv 1118
 - ?hegvd 1125
 - ?hegvx 1136
 - ?hpgv 1145
 - ?hpgvd 1152
 - ?hpgvx 1162
 - ?sbgv 1167
 - ?sbgvd 1173
 - ?sbgvx 1182
 - ?spgv 1142
 - ?spgvd 1148
 - ?spgvx 1157
 - ?sygv 1114
 - ?sygvd 1121
 - ?sygvx 1130
 - linear least squares problems
 - ?gels 947
 - ?gelsd 960
 - ?gelss 956
 - ?gelsy 951
 - ?lals0 (auxiliary) 1350
 - ?lalsa (auxiliary) 1354
 - ?lalsd (auxiliary) 1358
 - nonsymmetric eigenproblems
 - ?gees 1072
 - ?geesx 1077
 - ?geev 1084
 - ?geevx 1089
 - singular value decomposition
 - ?gelsd 960
 - ?gesdd 1102
 - ?gesvd 1096
 - ?ggsvd 1107
 - solving linear equations
 - ?gbsv 536
 - ?gbsvx 538

LAPACK routines (*continued*)

driver routines (*continued*)

solving linear equations (*continued*)

?gesv 525

?gesvx 529

?gtsv 545

?gtsvx 547

?hesv 590

?hesvx 593

?hpsv 605

?hpsvx 607

?pbsv 568

?pbsvx 570

?posv 552

?posvx 554

?ppsv 560

?ppsvx 562

?ptsv 576

?ptsvx 578

?spsv 598

?spsvx 601

?sysv 582

?sysvx 585

symmetric eigenproblems

?hbev 1034

?hbevd 1041

?hbevx 1050

?heev 976

?heevd 983

?heevr 1003

?heevx 992

?hpev 1012

?hpevd 1018

?hpevx 1027

?sbev 1032

?sbevd 1037

?sbevx 1046

?spev 1010

?spevd 1014

?spevx 1023

?stev 1055

?stevd 1057

?stevr 1065

?stevx 1061

?syev 973

?syevd 979

?syevr 997

?syevx 987

environmental enquiry 1626, 1629

LAPACK routines (*continued*)

finding a relatively isolated eigenvalue 1464

general band matrix

equilibration 1392

general matrix

block reflector 1482

elementary reflector 1480

reduction to bidiagonal form 1256, 1275

general rectangular matrix

block reflector 1439

elementary reflector 1437, 1446

equilibration 1394

LQ factorization 1261

plane rotation 1537

QL factorization 1263

QR factorization 1264

row interchanges 1545

RQ factorization 1266

general square matrix

reduction to upper Hessenberg form 1259

general tridiagonal matrix 1328, 1330, 1332,

1366, 1460, 1472

generalized eigenvalue problems

?hbgst 824

?hegst 814

?hpgst 819

?pbstf 827

?sbgst 821

?spgst 816

?sygst 812

generalized SVD

?ggsvp 934

?tgsja 939

generalized Sylvester equation

?tgsyl 923

Hermitian band matrix

equilibration 1396, 1428

Hermitian band matrix in packed storage

equilibration 1426

Hermitian matrix

computing eigenvalues and eigenvectors 1320

Householder matrix

elementary reflector 1441

incremental condition estimation 1345

linear dependence of vectors 1388

LQ factorization

?gelq2 1261

?gelqf 640

?orglq 644

LAPACK routines (*continued*)LQ factorization (*continued*)

?ormlq 646

?unglq 649

?unmlq 652

LU factorization

general band matrix 1255

matrix equilibration

?gbequ 514

?geequ 512

?laqgb 1392

?laqge 1394

?laqhb 1396

?laqsb 1424

?laqsp 1426

?laqsy 1428

?pbequ 521

?poequ 517

?ppequ 519

matrix inversion

?getri 493

?hetri 502

?hptri 506

?potri 496

?pptri 498

?sptri 504

?sytri 500

?tptri 510

?trtri 508

matrix-matrix product

?lagtm 1330

merging sets of singular values 1496, 1511

mixed precision iterative refinement subroutines
525, 1620, 1621, 1622, 1623

nonsymmetric eigenvalue problems

?gebak 853

?gebal 849

?gehrd 834

?hsein 861

?hseqr 855

?orghr 836

?ormhr 839

?trevc 867

?trexc 877

?trsen 880

?trsna 872

?unghr 843

?unmhr 846

off-diagonal and diagonal elements 1528

LAPACK routines (*continued*)

permutation list creation 1361

permutation of matrix columns 1389

plane rotation 1476, 1478, 1537

plane rotation vector 1447

QL factorization

?geql2 1263

?geqlf 655

?orgql 657

?ormql 662

?ungql 660

?unmql 665

QR factorization

?geqp3 625

?geqpf 622

?geqr2 1264

?geqrf 619

?ggqrf 691

?ggrqf 695

?laqp2 1398

?laqps 1400

?orgqr 629

?ormqr 632

?ungqr 635

?unmqr 637

p?geqrf 1719

random numbers vector 1449

real lower bidiagonal matrix

SVD 1524

real square bidiagonal matrix

singular values 1529

real symmetric matrix 1378

real symmetric tridiagonal matrix 1285, 1376

real upper bidiagonal matrix

singular values 1491

SVD 1493, 1520, 1524

real upper quasi-triangular matrix

orthogonal similarity transformation 1322

reciprocal condition numbers for eigenvalues

and/or eigenvectors

?tgsna 928

RQ factorization

?geqr2 1266

?gerqf 668

?orgrq 671

?ormrq 676

?ungrq 673

?unmrq 679

LAPACK routines (*continued*)

RZ factorization
 ?ormrz 684
 ?tzrzf 682
 ?unmrz 687
singular value decomposition
 ?bdsdc 728
 ?bdsqr 724
 ?gbbbrd 706
 ?gebrd 702
 ?orgbr 709
 ?ormbr 713
 ?ungbr 716
 ?unmbr 720
solution refinement and error estimation
 ?gbrfs 454
 ?gerfs 451
 ?gtrfs 457
 ?herfs 475
 ?hprfs 481
 ?pbrfs 466
 ?porfs 460
 ?pprfs 463
 ?ptrfs 469
 ?sprfs 478
 ?syrfs 472
 ?tbrfs 490
 ?tprfs 487
 ?trrfs 484
solving linear equations
 ?gbtrs 382
 ?getrs 379
 ?gttrs 385
 ?hetrs 400
 ?hptrs 406
 ?laln2 1347
 ?laqtr 1429
 ?pbtrs 393
 ?potrs 388
 ?pptrs 390
 ?pttrs 396
 ?sptrs 403
 ?sytrs 398
 ?tbtrs 414
 ?tptrs 411
 ?trtrs 408
sorting numbers 1541
square root 1390, 1391
square roots 1500, 1503, 1505, 1515, 1517, 1633

LAPACK routines (*continued*)

Sylvester equation
 ?lasy2 1547
 ?tgsy2 1615
 ?trsyl 886
symmetric band matrix
 equilibration 1424, 1428
symmetric band matrix in packed storage
 equilibration 1426
symmetric eigenvalue problems
 ?disna 809
 ?hbtrd 774
 ?herdb 742
 ?hetrd 750
 ?hptrd 765
 ?opgtr 760
 ?opmtr 762
 ?orgtr 745
 ?ormtr 747
 ?pteqr 798
 ?sbtrd 771
 ?sptrd 758
 ?stebz 802
 ?stedc 787
 ?stegr 793
 ?stein 806
 ?stemr 782
 ?steqr 778
 ?sterf 776
 ?syrdb 739
 ?sytrd 736
 ?ungtr 753
 ?unmtr 755
 ?upgtr 767
 ?upmtr 769
auxiliary
 ?lae2 1284
 ?laebz 1285
 ?laed0 1290
 ?laed1 1293
 ?laed2 1295
 ?laed3 1298
 ?laed4 1300
 ?laed5 1302
 ?laed6 1303
 ?laed7 1305
 ?laed8 1309
 ?laed9 1313
 ?laeda 1315

-
- LAPACK routines (*continued*)
- symmetric matrix
 - computing eigenvalues and eigenvectors 1320
 - packed storage 1373
 - symmetric positive-definite tridiagonal matrix
 - eigenvalues 1530
 - trapezoidal matrix 1385, 1569
 - triangular factorization
 - ?gbtrf 354
 - ?getrf 352
 - ?gttrf 356
 - ?hetrf 370
 - ?hptrf 376
 - ?pbtrf 363
 - ?potrf 358
 - ?pptrf 360
 - ?pttrf 365
 - ?sptfr 374
 - ?sytrf 367
 - p?dbtrf 1651
 - triangular matrix
 - packed storage 1383
 - triangular system of equations 1559, 1565
 - tridiagonal band matrix 1381
 - uniform distribution 1479
 - unreduced symmetric tridiagonal matrix 1290
 - updated upper bidiagonal matrix
 - SVD 1506
 - updating sum of squares 1542
 - upper Hessenberg matrix
 - computing a specified eigenvector 1317
 - eigenvalues 1336
 - Schur factorization 1336
 - utility functions and routines
 - ?labad 1633
 - ?lamc1 1635
 - ?lamc2 1636
 - ?lamc3 1637
 - ?lamc4 1637
 - ?lamc5 1638
 - ?lamch 1634
 - ieeck 1631
 - ilaenv 1626
 - ilaver 1625
 - iparmq 1629
 - lsame 1632
 - lsamen 1632
 - second/dsecnd 1639
 - xerbla 1640
 - Laplace 2416
 - Laplace problem
 - three-dimensional 2665
 - two-dimensional 2660
 - largest absolute value of element
 - complex Hermitian matrix
 - packed storage 1375
 - complex Hermitian tridiagonal matrix 1376
 - complex symmetric matrix 1378
 - general rectangular matrix 1365, 2005
 - general tridiagonal matrix 1366
 - Hermitian band matrix 1371
 - real symmetric matrix 1378, 2010
 - real symmetric tridiagonal matrix 1376
 - symmetric band matrix 1369
 - symmetric matrix
 - packed storage 1373
 - trapezoidal matrix 1385
 - triangular band matrix 1381
 - triangular matrix
 - packed storage 1383
 - upper Hessenberg matrix 1368, 2008
 - leading dimension 2898
 - leapfrog method 2366
 - LeapfrogStream 2389
 - least squares problems 613
 - length. dimension 2893
 - library version 2783
 - Library Version Obtaining 2783
 - library version string 2785
 - linear combination of vectors 65
 - Linear Congruential Generator 2360
 - linear equations, solving 379, 382, 385, 388, 390,
393, 396, 398, 400, 403, 406, 408, 411,
414, 525, 529, 536, 538, 545, 547, 552,
554, 560, 562, 568, 570, 576, 578, 582,
585, 590, 593, 598, 601, 605, 607, 1347,
1429, 1432, 1663, 1665, 1668, 1670, 1673,
1676, 1679, 1682, 1861, 1864, 1871, 1874,
1877, 1880, 1882, 1889, 1892, 1895, 2115,
2150, 2152
 - tridiagonal symmetric positive-definite matrix
 - LAPACK 576
 - ScaLAPACK 1892
 - band matrix
 - LAPACK 536, 538
 - ScaLAPACK 1871
 - Cholesky-factored matrix
 - LAPACK 393

linear equations, solving (*continued*)
 Cholesky-factored matrix (*continued*)
 ScaLAPACK 1670
 diagonally dominant-like matrix
 banded 1679
 tridiagonal 1676
 general band matrix
 ScaLAPACK 1874
 general matrix
 band storage 382, 1665
 general tridiagonal matrix
 ScaLAPACK 1877
 Hermitian matrix
 error bounds 593, 607
 packed storage 406, 605, 607
 Hermitian positive-definite matrix
 band storage
 LAPACK 568
 ScaLAPACK 1889
 error bounds
 LAPACK 554
 ScaLAPACK 1882
 LAPACK
 linear equations, solving
 multiple right-hand sides
 symmetric positive-definite
 matrix 552
 packed storage 390, 560, 562
 ScaLAPACK 1882
 Hermitian positive-definite tridiagonal linear
 equations 2152
 Hermitian positive-definite tridiagonal matrix 1673
 multiple right-hand sides
 band matrix
 LAPACK 536, 538
 ScaLAPACK 1871
 Hermitian matrix 590, 605
 Hermitian positive-definite matrix
 band storage 568
 square matrix
 LAPACK 525, 529
 ScaLAPACK 1861, 1864
 symmetric matrix 582, 598
 symmetric positive-definite matrix
 band storage 568
 tridiagonal matrix 545, 547
 overestimated or underestimated system 1895

linear equations, solving (*continued*)
 square matrix
 error bounds
 LAPACK 529, 538
 ScaLAPACK 1864
 LAPACK 525, 529
 ScaLAPACK 1861, 1864
 symmetric matrix
 error bounds 585, 601
 packed storage 403, 598, 601
 symmetric positive-definite matrix
 band storage
 LAPACK 568
 ScaLAPACK 1889
 error bounds
 LAPACK 554
 ScaLAPACK 1882
 LAPACK 552, 554
 packed storage 390, 560, 562
 ScaLAPACK 1880, 1882
 symmetric positive-definite tridiagonal linear
 equations 2152
 triangular matrix
 band storage 414, 2115
 packed storage 411
 tridiagonal Hermitian positive-definite matrix
 error bounds 578
 LAPACK 576
 ScaLAPACK 1892
 tridiagonal matrix
 error bounds 547
 LAPACK 385, 396, 545, 547
 LAPACK auxiliary 1432
 ScaLAPACK auxiliary 2150
 tridiagonal symmetric positive-definite matrix
 error bounds 578
 Linear Least Squares (LLS) Problems 947
 Ln 2294
 LoadStreamF 2388
 Log10 2296
 Log1p 2297
 Lognormal 2429
 LQ factorization 617, 644, 649, 1261, 1739, 1741,
 1971
 computing the elements of
 orthogonal matrix Q 644
 real orthogonal matrix Q 1739
 unitary matrix Q 649, 1741
 general rectangular matrix 1261, 1971

lsame 1632, 2794
 lsamen 1632, 2795
 LU factorization 352, 354, 356, 1255, 1268, 1269,
 1271, 1272, 1328, 1332, 1556, 1646, 1648,
 1651, 1660, 1864, 1956, 1960, 1983, 2145,
 2147, 2149
 band matrix
 blocked algorithm 2147
 unblocked algorithm 2145
 diagonally dominant-like tridiagonal matrix 1660
 general band matrix 1255
 general matrix 1271, 1983
 solving linear equations
 general matrix 1268
 square matrix 1864
 tridiagonal matrix 1272, 1332
 triangular band matrix 1956
 tridiagonal band matrix 1960
 tridiagonal matrix 356, 1328, 2149
 with complete pivoting 1269, 1556
 with partial pivoting 1271, 1983

M

machine parameters
 LAPACK 1634
 ScaLAPACK 2158
 matrix arguments 2893, 2895, 2898
 column-major ordering 2893, 2898
 example 2898
 leading dimension 2898
 number of columns 2898
 number of rows 2898
 transposition parameter 2898
 matrix block
 QR factorization
 with pivoting 1398
 matrix equation
 $AX = B$ 187, 349, 379, 1642, 1663
 matrix one-dimensional substructures 2893
 matrix-matrix operation
 product
 general matrix 156, 160
 rank-2k update
 Hermitian matrix 170
 symmetric matrix 180
 rank-n update
 Hermitian matrix 167

matrix-matrix operation (*continued*)
 rank-n update (*continued*)
 symmetric matrix 177
 scalar-matrix-matrix product
 Hermitian matrix 164
 symmetric matrix 173
 matrix-matrix operation:scalar-matrix-matrix product
 triangular matrix 184
 matrix-vector operation
 product
 Hermitian matrix 104, 107, 114
 real symmetric matrix 125, 132
 triangular matrix 139, 146, 150
 rank-1 update
 Hermitian matrix 110, 117
 real symmetric matrix 127, 134
 rank-2 update
 Hermitian matrix 112, 119
 symmetric matrix 129, 136
 matrix-vector operation:product
 Hermitian matrix
 band storage 104
 packed storage 114
 real symmetric matrix
 packed storage 125
 symmetric matrix
 band storage 121
 triangular matrix
 band storage 139
 packed storage 146
 matrix-vector operation:rank-1 update
 Hermitian matrix
 packed storage 117
 real symmetric matrix
 packed storage 127
 matrix-vector operation:rank-2 update
 Hermitian matrix
 packed storage 119
 symmetric matrix
 packed storage 129
 mkl_?csrgermv 219
 mkl_cspblas_dbsergm 249
 mkl_cspblas_dbssymv 256
 mkl_cspblas_dbsrtrsv 263
 mkl_cspblas_dcoogermv 251
 mkl_cspblas_dcoosymv 258
 mkl_cspblas_dcootrsv 266
 mkl_cspblas_dcsrgermv 247
 mkl_cspblas_dcsrsymv 254

mkl_cspblas_dcsrtrsv 261
 mkl_dbsergemv 221
 mkl_dbsermm 295
 mkl_dbsermv 271
 mkl_dbsersm 315
 mkl_dbsersv 283
 mkl_dbsersymv 230
 mkl_dbsertrsv 239
 mkl_dcoogemv 224
 mkl_dcoomm 302
 mkl_dcoomv 277
 mkl_dcoosm 312
 mkl_dcoosv 289
 mkl_dcoosymv 232
 mkl_dcootrsv 242
 mkl_dcscmm 299
 mkl_dcscmv 274
 mkl_dcscsm 309
 mkl_dcscsv 286
 mkl_dcsrmm 292
 mkl_dcsrmv 268
 mkl_dcsrsm 305
 mkl_dcsrsv 280
 mkl_dcsrsymv 228
 mkl_dcsrtrsv 237
 mkl_ddiagemv 226
 mkl_ddiamm 329
 mkl_ddiamv 318
 mkl_ddiasm 335
 mkl_ddiasv 324
 mkl_ddiasymv 235
 mkl_ddiatrsv 244
 mkl_domain_get_max_threads 2791
 mkl_domain_set_num_threads 2788
 mkl_dskymm 332
 mkl_dskymv 321
 mkl_dskysm 337
 mkl_dskysv 326
 MKL_free 2802
 MKL_FreeBuffers 2798
 mkl_get_dynamic 2792
 mkl_get_max_threads 2790
 MKL_malloc 2801
 MKL_MemStat 2800
 mkl_set_dynamic 2789
 mkl_set_num_threads 2788
 MKLGetVersion 2783
 MKLGetVersionStirng 2785
 Modf 2338

MPI 1641
 Mul 2268
 MulByConj 2269
 Multiplicative Congruential Generator 2360

N

naming conventions 57, 59, 191, 205, 614, 1642,
 2256
 BLAS 59
 LAPACK 614, 1642
 Sparse BLAS Level 1 191
 Sparse BLAS Level 2 205
 Sparse BLAS Level 3 205
 VML 2256
 NearbyInt 2335
 negative eigenvalues 2004
 NegBinomial 2459
 NewStream 2372
 NewStreamEx 2373
 NewTaskX1D 2486
 nonlinear least squares problem 3195
 Nonsymmetric Eigenproblems 1071

O

off-diagonal elements
 initialization 2062
 LAPACK 1528
 ScaLAPACK 2062
 one-dimensional FFTs
 storage effects 2561, 2563
 optimization solvers basics 3195
 orthogonal matrix 699, 731, 829, 889, 1579, 1580,
 1582, 1584, 1585, 1826, 1840, 2085, 2088,
 2091, 2094, 2097
 from LQ factorization
 LAPACK 1582
 ScaLAPACK 2091
 from QL factorization
 LAPACK 1579, 1585
 ScaLAPACK 2085, 2097
 from QR factorization
 LAPACK 1580
 ScaLAPACK 2088
 from RQ factorization
 LAPACK 1584

orthogonal matrix (*continued*)
from RQ factorization (*continued*)
ScaLAPACK 2094

P

p?dbsv 1874
p?dbtrf 1651
p?dbtrs 1679
p?dbtrsv 1956
p?dtsv 1877
p?dttrf 1660
p?dttrs 1676
p?dttrsv 1960
p?gbsv 1871
p?gbtrf 1648
p?gbtrs 1665
p?gebd2 1964
p?gebrd 1840
p?gecon 1685
p?geequ 1714
p?gehd2 1968
p?gehrd 1826
p?gelq2 1971
p?gelqf 1736
p?gels 1895
p?geql2 1974
p?geqlf 1750
p?geqpf 1722
p?geqr2 1977
p?geqrf 1719
p?gerfs 1695
p?gerq2 1980
p?gerqf 1764
p?gesv 1861
p?gesvd 1920
p?gesvx 1864
p?getf2 1983
p?getrf 1646
p?getri 1707
p?getrs 1663
p?ggqrf 1789
p?ggrqf 1795
p?heevx 1911
p?hegst 1858
p?hegvx 1934
p?hetrd 1809
p?labad 2156
p?labrd 1985
p?lachkieee 2157
p?lacon 1990
p?laconsb 1992
p?lACP2 1993
p?lACP3 1995
p?lAcPy 1997
p?laevswp 1999
p?lahqr 1837
p?lahrd 2001
p?laiect 2004
p?lamch 2158
p?lange 2005
p?lanhs 2008
p?lantr 2012
p?lapiv 2015
p?laqge 2019
p?laqsy 2021
p?lared1d 2024
p?lared2d 2025
p?larf 2027
p?larfb 2031
p?larfc 2035
p?larfg 2038
p?larft 2040
p?larz 2043
p?larzb 2047
p?larzt 2056
p?lascl 2060
p?laset 2062
p?lasmsub 2064
p?lasnbt 2160
p?lassq 2065
p?laswp 2068
p?latra 2070
p?latrd 2071
p?latrz 2078
p?lauu2 2081
p?lauum 2083
p?lawil 2084
p?max1 1952
p?org2l/p?ung2l 2085
p?org2r/p?ung2r 2088
p?orgl2/p?ungl2 2091
p?orglq 1739
p?orgql 1753
p?orgqr 1725
p?org2r/p?ungr2 2094
p?orgrq 1767

p?orm2l/p?unm2l 2097
 p?orm2r/p?unm2r 2101
 p?ormbr 1845
 p?ormhr 1830
 p?orml2/p?unml2 2106
 p?ormlq 1743
 p?ormql 1757
 p?ormqr 1729
 p?ormr2/p?unmr2 2110
 p?ormrq 1772
 p?ormrz 1782
 p?ormtr 1805
 p?pbsv 1889
 p?pbtrf 1655
 p?pbtrs 1670
 p?pbtrsv 2115
 p?pocon 1688
 p?poequ 1716
 p?porfs 1699
 p?posv 1880
 p?posvx 1882
 p?potf2 2123
 p?potrf 1653
 p?potri 1710
 p?potrs 1668
 p?ptsv 1892
 p?pttrf 1658
 p?pttrs 1673
 p?pttrsv 2119
 p?rscl 2125
 p?stebz 1817
 p?stein 1821
 p?sum1 1955
 p?syev 1900
 p?syevx 1903
 p?sygs2/p?hegs2 2127
 p?sygst 1856
 p?sygvx 1925
 p?sytd2/p?hetd2 2130
 p?sytrd 1801
 p?trcon 1691
 p?trrfs 1703
 p?trti2 2134
 p?trtri 1712
 p?trtrs 1682
 p?tzrzf 1779
 p?unglq 1741
 p?ungql 1755
 p?ungqr 1727

p?ungrq 1769
 p?unmbr 1850
 p?unmhr 1834
 p?unmlq 1747
 p?unmql 1761
 p?unmqr 1733
 p?unmrq 1775
 p?unmrz 1785
 p?unmtr 1813
 Pack 2339
 Packed formats 2554
 packed storage scheme 2895
 parallel direct solver (Pardiso) 2163
 parameter partitioning, for interval systems 2616
 parameters
 for a Givens rotation 77
 modified Givens transformation 81
 pardiso function 2164
 PARDISO* solver 2163
 Partial Differential Equations support 2633, 2659,
 2660, 2661, 2662, 2663, 2664, 2665
 Helmholtz problem on a sphere 2661
 Poisson problem on a sphere 2662
 three-dimensional Helmholtz problem 2663
 three-dimensional Laplace problem 2665
 three-dimensional Poisson problem 2664
 two-dimensional Helmholtz problem 2659
 two-dimensional Laplace problem 2660
 two-dimensional Poisson problem 2660
 PDE support 2633
 PDE Support Code Examples 3041
 pdlaictb 2004
 pdlaictl 2004
 permutation matrix 2863
 pivoting matrix rows or columns 2015
 PL Interface 2657
 platforms supported 54
 points rotation
 in the modified plane 78
 in the plane 75
 Poisson 2455
 Poisson Library 2657, 2658, 2666, 2670, 2673, 2679,
 2685, 2686, 2689, 2691, 2694, 3060
 routines
 ?_commit_Helmholtz_2D 2673
 ?_commit_Helmholtz_3D 2673
 ?_commit_sph_np 2689
 ?_commit_sph_p 2689
 ?_Helmholtz_2D 2679

Poisson Library (*continued*)
 routines (*continued*)
 ?_Helmholtz_3D 2679
 ?_init_Helmholtz_2D 2670
 ?_init_Helmholtz_3D 2670
 ?_init_sph_np 2686
 ?_init_sph_p 2686
 ?_sph_np 2691
 ?_sph_p 2691
 code examples 3060
 free_Helmholtz_2D 2685
 free_Helmholtz_3D 2685
 free_sph_np 2694
 free_sph_p 2694
 structure 2658
 Poisson problem
 on a sphere 2662
 three-dimensional 2664
 two-dimensional 2660
 PoissonV 2457
 Pow 2285
 Pow2o3 2282
 Pow3o2 2283
 Powx 2287
 preconditioners based on incomplete LU factorization
 2238, 2242, 2247, 3091
 code examples 3091
 dcsrilu0 2242
 dcsrilut 2247
 Preconditioners Interface Description 2241
 preconditioning, of an interval system 2631
 process grid 1641
 product
 matrix-vector
 general matrix 91, 95
 Hermitian matrix 104, 107, 114
 real symmetric matrix 125, 132
 triangular matrix 139, 146, 150
 scalar-matrix
 general matrix 156, 160
 Hermitian matrix 164
 scalar-matrix-matrix
 general matrix 156, 160
 Hermitian matrix 164
 symmetric matrix 173
 triangular matrix 184
 vector-scalar 83

product:matrix-vector
 general matrix
 band storage 91
 Hermitian matrix
 band storage 104
 packed storage 114
 real symmetric matrix
 packed storage 125
 symmetric matrix
 band storage 121
 triangular matrix
 band storage 139
 packed storage 146
 pseudorandom numbers 2357
 psiaiect 2004
 pxxerbla 2161, 2793

Q

QL factorization
 computing the elements of
 complex matrix Q 660
 orthogonal matrix Q 1753
 real matrix Q 657
 unitary matrix Q 1755
 general rectangular matrix
 LAPACK 1263
 ScaLAPACK 1974
 multiplying general matrix by
 orthogonal matrix Q 1757
 unitary matrix Q 1761
 QR factorization 617, 622, 625, 629, 635, 1264, 1266,
 1398, 1400, 1722, 1725, 1727, 1977, 1980
 computing the elements of
 orthogonal matrix Q 629, 1725
 unitary matrix Q 635, 1727
 general rectangular matrix
 LAPACK 1264, 1266
 ScaLAPACK 1977, 1980
 with pivoting
 ScaLAPACK 1722
 quasi-random numbers 2357
 quasi-triangular matrix
 LAPACK 829, 889
 ScaLAPACK 1826
 quasi-triangular system of equations 1429

R

- random number generators 2357
- random stream 2368
- Random Streams 2368
- rank-1 update
 - conjugated, general matrix 100
 - general matrix 98
 - Hermitian matrix
 - packed storage 117
 - real symmetric matrix
 - packed storage 127
 - unconjugated, general matrix 102
- rank-2 update
 - Hermitian matrix
 - packed storage 119
 - symmetric matrix
 - packed storage 129
- rank-2k update
 - Hermitian matrix 170
 - symmetric matrix 180
- rank-n update
 - Hermitian matrix 167
 - symmetric matrix 177
- Rayleigh 2426
- RCI CG Interface 2203
- RCI CG sparse solver routines
 - dcg 2221, 2227
 - dcg_check 2219
 - dcg_get 2223
 - dcg_init 2218
 - dcgmrhs_check 2225
 - dcgmrhs_get 2229
 - dcgmrhs_init 2224
- RCI FGMRES Interface 2209
- RCI FGMRES sparse solver routines
 - dfgmres_check 2231
 - dfgmres_get 2235
 - dfgmres_init 2230
- RCI GFMRES sparse solver routines
 - dfgres 2233
- RCI ISS 2198
- RCI ISS interface 2198
- RCI ISS sparse solver routines
 - implementation details 2237
- real matrix
 - QR factorization
 - with pivoting 1400
 - real symmetric matrix
 - 1-norm value 1378
 - Frobenius norm 1378
 - infinity- norm 1378
 - largest absolute value of element 1378
 - real symmetric tridiagonal matrix
 - 1-norm value 1376
 - Frobenius norm 1376
 - infinity- norm 1376
 - largest absolute value of element 1376
 - reducing generalized eigenvalue problems
 - LAPACK 812
 - ScaLAPACK 1856
 - reduction to upper Hessenberg form
 - general matrix 1968
 - general square matrix 1259
 - refining solutions of linear equations
 - band matrix 454
 - general matrix 451, 1695
 - Hermitian matrix
 - packed storage 481
 - Hermitian positive-definite matrix
 - band storage 466
 - packed storage 463
 - symmetric matrix
 - packed storage 478
 - symmetric positive-definite matrix
 - band storage 466
 - packed storage 463
 - symmetric/Hermitian positive-definite distributed matrix 1699
 - tridiagonal matrix 457
 - RegisterBrng 2464
 - registering a basic generator 2461
 - reordering of matrices 2864
 - Reverse Communication Interface 2198
 - Rex-Rohn test 2626, 2627
 - Rint 2336
 - Ris-Beeck spectral criterion 2626
 - rotation
 - of points in the modified plane 78
 - of points in the plane 75
 - of sparse vectors 201
 - parameters for a Givens rotation 77
 - parameters of modified Givens transformation 81
 - Round 2334
 - routine group 54
 - routine name conventions
 - BLAS 59

routine name conventions (*continued*)

Sparse BLAS Level 1 191
 Sparse BLAS Level 2 205
 Sparse BLAS Level 3 205
 RQ factorization
 computing the elements of
 complex matrix Q 673
 orthogonal matrix Q 1767
 real matrix Q 671
 unitary matrix Q 1769

Rump criterion 2627

S

SaveStreamF 2386

ScaLAPACK 1641

ScaLAPACK routines

 1D array redistribution 2024, 2025

 auxiliary routines

 ?combamax1 1954

 ?dbtf2 2145

 ?dbtrf 2147

 ?dttrf 2149

 ?dttrsv 2150

 ?lamsh 2136

 ?laref 2137

 ?lasorte 2140

 ?lasrt2 2141

 ?pttrsv 2152

 ?stein2 2142

 ?steqr2 2154

 p?dbtrsv 1956

 p?dttrsv 1960

 p?gebd2 1964

 p?gehd2 1968

 p?gelq2 1971

 p?geql2 1974

 p?geqr2 1977

 p?gerq2 1980

 p?getf2 1983

 p?labrd 1985

 p?lacgv 1951

 p?lacon 1990

 p?laconsb 1992

 p?lcp2 1993

 p?lcp3 1995

 p?lacpy 1997

 p?laevswp 1999

ScaLAPACK routines (*continued*)

 auxiliary routines (*continued*)

 p?lahrd 2001

 p?laiect 2004

 p?lange 2005

 p?lanhs 2008

 p?lansy, p?lanhe 2010

 p?lantr 2012

 p?lapiv 2015

 p?laqge 2019

 p?laqsy 2021

 p?lared1d 2024

 p?lared2d 2025

 p?larf 2027

 p?larfb 2031

 p?larfc 2035

 p?larfg 2038

 p?larft 2040

 p?larz 2043

 p?larzb 2047

 p?larzc 2052

 p?larzt 2056

 p?lascl 2060

 p?laset 2062

 p?lasmsub 2064

 p?lassq 2065

 p?laswp 2068

 p?latra 2070

 p?latrd 2071

 p?latrs 2075

 p?latrz 2078

 p?lauu2 2081

 p?lauum 2083

 p?lawil 2084

 p?max1 1952

 p?org2l/p?ung2l 2085

 p?org2r/p?ung2r 2088

 p?orgl2/p?ungl2 2091

 p?org2r/p?ungr2 2094

 p?orm2l/p?unm2l 2097

 p?orm2r/p?unm2r 2101

 p?orml2/p?unml2 2106

 p?ormr2/p?unmr2 2110

 p?pbtrsv 2115

 p?potf2 2123

 p?pttrsv 2119

 p?rscl 2125

 p?sum1 1955

 p?sygs2/p?hegs2 2127

ScaLAPACK routines (*continued*)

auxiliary routines (*continued*)

p?sytd2/p?hetd2 2130

p?trti2 2134

pdlaiectb 2004

pdlaiectl 2004

pslaiect 2004

block reflector

triangular factor 2040, 2056

Cholesky factorization 1658

complex matrix

complex elementary reflector 2052

complex vector

1-norm using true absolute value 1955

complex vector conjugation 1951

condition number estimation

p?gecon 1685

p?pocon 1688

p?trcon 1691

driver routines

p?dbsv 1874

p?dtsv 1877

p?gbsv 1871

p?gels 1895

p?gesv 1861

p?gesvd 1920

p?gesvx 1864

p?heevx 1911

p?hegvx 1934

p?pbsv 1889

p?posv 1880

p?posvx 1882

p?ptsv 1892

p?syev 1900

p?syevx 1903

p?sygvx 1925

error estimation

p?trrf 1703

error handling

pxerbla 2161, 2793

general matrix

block reflector 2047

elementary reflector 2043

LU factorization 1983

reduction to upper Hessenberg form 1968

general rectangular matrix

elementary reflector 2027

LQ factorization 1971

QL factorization 1974

ScaLAPACK routines (*continued*)

general rectangular matrix (*continued*)

QR factorization 1977

reduction to bidiagonal form 1985

reduction to real bidiagonal form 1964

row interchanges 2068

RQ factorization 1980

generalized eigenvalue problems

p?hegst 1858

p?sygst 1856

Householder matrix

elementary reflector 2038

LQ factorization

p?gelq2 1971

p?gelqf 1736

p?orglq 1739

p?ormlq 1743

p?unglq 1741

p?unmlq 1747

LU factorization

p?dbtrsv 1956

p?dttrf 1660

p?dttrsv 1960

p?getf2 1983

matrix equilibration

p?geequ 1714

p?poequ 1716

matrix inversion

p?getri 1707

p?potri 1710

p?trtri 1712

nonsymmetric eigenvalue problems

p?gehrd 1826

p?lahqr 1837

p?ormhr 1830

p?unmhr 1834

QL factorization

?geqlf 1750

?ungql 1755

p?geql2 1974

p?orgql 1753

p?ormql 1757

p?unmql 1761

QR factorization

p?geqpf 1722

p?geqr2 1977

p?ggqrf 1789

p?orgqr 1725

p?ormqr 1729

ScaLAPACK routines (*continued*)QR factorization (*continued*)

p?ungqr 1727

p?unmqr 1733

RQ factorization

p?gerq2 1980

p?gerqf 1764

p?ggrqf 1795

p?orgrq 1767

p?ormrq 1772

p?ungrq 1769

p?unmrq 1775

RZ factorization

p?ormrz 1782

p?tzrzf 1779

p?unmrz 1785

singular value decomposition

p?gebrd 1840

p?ormbr 1845

p?unmbr 1850

solution refinement and error estimation

p?gerfs 1695

p?porfs 1699

solving linear equations

?dttrsv 2150

?pttrsv 2152

p?dbtrs 1679

p?dttrs 1676

p?gbtrs 1665

p?getrs 1663

p?potrs 1668

p?pttrs 1673

p?trtrs 1682

symmetric eigenproblems

p?hetrd 1809

p?ormtr 1805

p?stebz 1817

p?stein 1821

p?sytrd 1801

p?unmtr 1813

symmetric eigenvalue problems

?stein2 2142

?steqr2 2154

trapezoidal matrix 2078

triangular factorization

?dbtrf 2147

?dttrf 2149

p?dbtrsv 1956

p?dttrsv 1960

ScaLAPACK routines (*continued*)triangular factorization (*continued*)

p?gbtrf 1648

p?getrf 1646

p?pbtrf 1655

p?potrf 1653

p?pttrf 1658

triangular system of equations 2075

updating sum of squares 2065

utility functions and routines

p?labad 2156

p?lachkieee 2157

p?lamch 2158

p?lasnbt 2160

pxerbla 2161, 2793

scalar-matrix product 156, 160, 164, 173

scalar-matrix-matrix product 156, 160, 164, 173, 184

general matrix 156, 160

symmetric matrix 173

triangular matrix 184

scaling

general rectangular matrix 2019

symmetric/Hermitian matrix 2021

scaling factors

general rectangular distributed matrix 1714

Hermitian positive definite distributed matrix 1716

symmetric positive definite distributed matrix 1716

scattering compressed sparse vector's elements into

full storage form 203

Schulz interval procedure 2624

Schulz iterative method 2624

Schur decomposition 912, 916

Schur factorization 1334, 1336, 1387

second/dsecnd 2796

Service Functions 2258

Service Routines 2370

setcpufrequency 2797

SetErrorCallBack 2351

SetErrorStatus 2349

SetInternalDecimation 2495

SetMode 2345

SetValue 2542

SetValueDM 2594

simple driver 1642

Sin 2301

SinCos 2303

single node matrix 2136

singular value decomposition

LAPACK 699, 1096

singular value decomposition (*continued*)

- LAPACK routines, singular value decomposition 1840
- ScaLAPACK 1840, 1920
- See also LAPACK routines, singular value decomposition 699
- Singular Value Decomposition 1096
- Sinh 2316
- SkipAheadStream 2393
- slag2d 1622
- small subdiagonal element 2064
- smallest absolute value of a vector element 87
- sNewAbstractStream 2380
- solution partitioning 2619
- solver
 - direct 2861
 - iterative 2861
- Solver
 - Sparse 2163
- solving linear equations 382
- solving linear equations. linear equations 1665
- solving linear equations. See linear equations 1347
- sorting
 - eigenpairs 2140
 - numbers in increasing/decreasing order
 - LAPACK 1541
 - ScaLAPACK 2141
- Sparse BLAS Level 1 190, 191
 - data types 191
 - naming conventions 191
- Sparse BLAS Level 1 routines and functions 191, 192, 194, 195, 197, 198, 200, 201, 203
 - ?axpyi 192
 - ?dotci 195
 - ?doti 194
 - ?dotui 197
 - ?gthr 198
 - ?gthrz 200
 - ?roti 201
 - ?sctr 203
- Sparse BLAS Level 2 204, 205
 - naming conventions 205
- sparse BLAS Level 2 routines
 - mkl_?csrgemv 219
 - mkl_cspblas_dbsrgemv 249
 - mkl_cspblas_dbsrsymv 256
 - mkl_cspblas_dbsrtrsv 263
 - mkl_cspblas_dcoogemv 251
 - mkl_cspblas_dcoosymv 258

sparse BLAS Level 2 routines (*continued*)

- mkl_cspblas_dcootrsv 266
- mkl_cspblas_dcsrgemv 247
- mkl_cspblas_dcsrsymv 254
- mkl_cspblas_dcsrtrsv 261
- mkl_dbsrgemv 221
- mkl_dbsrmv 271
- mkl_dbsrsv 283
- mkl_dbsrsymv 230
- mkl_dbsrtrsv 239
- mkl_dcoogemv 224
- mkl_dcoomv 277
- mkl_dcoosv 289
- mkl_dcoosymv 232
- mkl_dcootrsv 242
- mkl_dcscmv 274
- mkl_dcscsv 286
- mkl_dcsrmv 268
- mkl_dcsrsv 280
- mkl_dcsrsymv 228
- mkl_dcsrtrsv 237
- mkl_ddiagemv 226
- mkl_ddiamv 318
- mkl_ddiasv 324
- mkl_ddiasymv 235
- mkl_ddiatsrv 244
- mkl_dskymv 321
- mkl_dskysv 326
- Sparse BLAS Level 3 204, 205
 - naming conventions 205
- sparse BLAS Level 3 routines
 - mkl_dbsrmm 295
 - mkl_dbsrsm 315
 - mkl_dcoomm 302
 - mkl_dcoosm 312
 - mkl_dcscmm 299
 - mkl_dcscsm 309
 - mkl_dcsrmm 292
 - mkl_dcsrsm 305
 - mkl_ddiamm 329
 - mkl_ddiasm 335
 - mkl_dskymm 332
 - mkl_dskysm 337
- sparse matrices 204
- sparse matrix 204
- Sparse Matrix Storage Formats 206
- Sparse Solver
 - direct sparse solver interface
 - dss_create 2186

-
- Sparse Solver (*continued*)
 - direct sparse solver interface (*continued*)
 - dss_define_structure
 - dss_define_structure 2187
 - dss_delete 2192
 - dss_factor_real, dss_factor_complex 2190
 - dss_reorder 2189
 - dss_solve_real, dss_solve_complex 2191
 - dss_statistics 2193
 - mkl_cvt_to_null_terminated_str 2196
 - iterative sparse solver interface
 - dcg 2221
 - dcg_check 2219
 - dcg_get 2223
 - dcg_init 2218
 - dcgmrhs 2227
 - dcgmrhs_check 2225
 - dcgmrhs_get 2229
 - dcgmrhs_init 2224
 - dfgmres 2233
 - dfgmres_check 2231
 - dfgmres_get 2235
 - dfgmres_init 2230
 - preconditioners based on incomplete LU factorization
 - dcsrilu0 2242
 - dcsrilut 2247
 - Sparse Solvers 2163
 - sparse vectors 190, 191, 192, 194, 195, 197, 198, 200, 201, 203
 - adding and scaling 192
 - complex dot product, conjugated 195
 - complex dot product, unconjugated 197
 - compressed form 190
 - converting to compressed form 198, 200
 - converting to full-storage form 203
 - full-storage form 190
 - Givens rotation 201
 - norm 191
 - passed to BLAS level 1 routines 191
 - real dot product 194
 - scaling 191
 - split Cholesky factorization (band matrices) 827
 - Sqr 2266
 - Sqrt 2277
 - square matrix
 - 1-norm estimation
 - LAPACK 1278, 1280
 - ScaLAPACK 1990
 - status checking
 - FFT 2524
 - stream 2368
 - stream descriptor 2359
 - stride, increment 2893
 - Sub 2265
 - sum
 - of magnitudes of the vector elements 64
 - of sparse vector and full-storage vector 192
 - of vectors 65
 - sum of squares
 - updating
 - LAPACK 1542
 - ScaLAPACK 2065
 - support functions
 - MKL_free 2802
 - MKL_malloc 2801
 - MKL_MemStat 2800
 - support routines
 - MKL_FreeBuffers 2798
 - SVD (singular value decomposition)
 - LAPACK 699
 - ScaLAPACK 1840
 - swapping adjacent diagonal blocks 1322, 1612
 - swapping vectors 85
 - Sylvester's equation 886
 - symmetric band matrix
 - 1-norm value 1369
 - Frobenius norm 1369
 - infinity- norm 1369
 - largest absolute value of element 1369
 - Symmetric Eigenproblems 972
 - symmetric indefinite matrix
 - factorization with diagonal pivoting method 1608
 - symmetric matrix 121, 125, 127, 129, 132, 134, 136, 173, 177, 180, 367, 374, 398, 403, 434, 438, 500, 504, 731, 809, 811, 1246, 1248, 1249, 1251, 1561, 1604, 1606, 1900, 1903, 2021, 2071, 2127, 2130
 - Bunch-Kaufman factorization
 - packed storage 374
 - eigenvalues and eigenvectors 1900, 1903
 - estimating the condition number
 - packed storage 438
 - generalized eigenvalue problems 811
 - inverting the matrix
 - packed storage 504
 - matrix-vector product
 - band storage 121

- symmetric matrix (*continued*)
 - matrix-vector product (*continued*)
 - packed storage 125, 1246
 - rank-1 update
 - packed storage 127, 1248
 - rank-2 update
 - packed storage 129
 - rank-2k update 180
 - rank-n update 177
 - reducing to standard form
 - LAPACK 1604
 - ScaLAPACK 2127
 - reducing to tridiagonal form
 - LAPACK 1561
 - ScaLAPACK 2071
 - scalar-matrix-matrix product 173
 - scaling 2021
 - solving systems of linear equations
 - packed storage 403
- symmetric matrix in packed form
 - 1-norm value 1373
 - Frobenius norm 1373
 - infinity- norm 1373
 - largest absolute value of element 1373
- symmetric positive definite distributed matrix
 - computing scaling factors 1716
 - equilibration 1716
- symmetric positive-definite band matrix
 - Cholesky factorization 1598
- symmetric positive-definite distributed matrix
 - inverting the matrix 1710
- symmetric positive-definite matrix
 - Cholesky factorization
 - band storage 363, 1655
 - LAPACK 358, 1600
 - packed storage 360
 - ScaLAPACK 1653, 2123
 - estimating the condition number
 - band storage 429
 - packed storage 427
 - tridiagonal matrix 432
 - inverting the matrix
 - packed storage 498
 - solving systems of linear equations
 - band storage 393, 1670
 - LAPACK 388
 - packed storage 390
 - ScaLAPACK 1668

- symmetric positive-definite tridiagonal matrix
 - solving systems of linear equations 1673
- system of linear equations
 - with a triangular matrix
 - band storage 142
 - packed storage 148
- systems of linear equations 379, 385, 396, 2150
 - linear equations 2150
- systems of linear equationslinear equations 1663

T

- Tan 2305
- Tanh 2318
- threading control
 - mkl_domain_get_max_threads 2791
 - mkl_domain_set_num_threads 2788
 - mkl_get_dynamic 2792
 - mkl_get_max_threads 2790
 - mkl_set_dynamic 2789
 - mkl_set_num_threads 2788
- Threading Control 2787
- timing functions
 - getcpuclocks 2796
 - getcpufrequency 2797
 - second/dsecnd 2796
 - setcpufrequency 2797
- TR routines
 - dtrnls_delete 2729
 - dtrnls_get 2727
 - dtrnls_init 2724
 - dtrnls_solve 2725
 - dtrnlsdbc_delete 2749
 - dtrnlsdbc_get 2747
 - dtrnlsdbc_init 2744
 - dtrnlsdbc_solve 2746
 - nonlinear least squares problem
 - with linear bound constraints 2743
 - without constraints 2723
 - organization and implementation 2721
- transposition parameter 2898
- trapezoidal matrix
 - 1-norm value 1385
 - Frobenius norm 1385
 - infinity- norm 1385
 - largest absolute value of element 1385
 - reduction to triangular form 2078

-
- trapezoidal matrix (*continued*)
 - RZ factorization
 - LAPACK 682
 - ScaLAPACK 1779
 - triangular band matrix
 - 1-norm value 1381
 - Frobenius norm 1381
 - infinity- norm 1381
 - largest absolute value of element 1381
 - triangular banded equations
 - LAPACK 1554
 - ScaLAPACK 2115
 - triangular distributed matrix
 - inverting the matrix 1712
 - triangular factorization
 - band matrix 354, 1648, 1651, 1956, 2147
 - general matrix 352, 1646
 - Hermitian matrix
 - packed storage 376
 - Hermitian positive-definite matrix
 - band storage 363, 1655
 - packed storage 360
 - tridiagonal matrix 365, 1658
 - symmetric matrix
 - packed storage 374
 - symmetric positive-definite matrix
 - band storage 363, 1655
 - packed storage 360
 - tridiagonal matrix 365, 1658
 - tridiagonal matrix
 - LAPACK 356
 - ScaLAPACK 2149
 - triangular matrix 139, 142, 146, 148, 150, 153, 184, 408, 411, 414, 443, 445, 448, 508, 510, 829, 889, 1385, 1571, 1573, 1612, 1619, 1682, 1826, 2012, 2081, 2083, 2134
 - 1-norm value
 - LAPACK 1385
 - ScaLAPACK 2012
 - estimating the condition number
 - band storage 448
 - packed storage 445
 - Frobenius norm
 - LAPACK 1385
 - ScaLAPACK 2012
 - infinity- norm
 - LAPACK 1385
 - ScaLAPACK 2012
 - triangular matrix (*continued*)
 - inverting the matrix
 - LAPACK 1619
 - packed storage 510
 - ScaLAPACK 2134
 - largest absolute value of element
 - LAPACK 1385
 - ScaLAPACK 2012
 - matrix-vector product
 - band storage 139
 - packed storage 146
 - product
 - blocked algorithm 1573, 2083
 - LAPACK 1571, 1573
 - ScaLAPACK 2081, 2083
 - unblocked algorithm 1571
 - ScaLAPACK 1826
 - scalar-matrix-matrix product 184
 - solving systems of linear equations
 - band storage 142, 414
 - packed storage 148, 411
 - ScaLAPACK 1682
 - swapping adjacent diagonal blocks 1612
 - triangular matrix in packed form
 - 1-norm value 1383
 - Frobenius norm 1383
 - infinity- norm 1383
 - largest absolute value of element 1383
 - triangular system of equations
 - solving with scale factor
 - LAPACK 1565
 - ScaLAPACK 2075
 - tridiagonal system of equations 1601
 - tridiagonal matrix 385, 396, 422, 731, 2150
 - estimating the condition number 422
 - solving systems of linear equations
 - ScaLAPACK 2150
 - tridiagonal triangular factorization
 - band matrix 1960
 - tridiagonal triangular system of equations 2119
 - trigonometric transform
 - backward cosine 2635
 - backward sine 2634
 - backward staggered cosine 2636
 - backward staggered sine 2634
 - backward twice staggered cosine 2636
 - backward twice staggered sine 2635
 - forward cosine 2635
 - forward sine 2634

trigonometric transform (*continued*)
 forward staggered cosine 2635
 forward staggered sine 2634
 forward twice staggered cosine 2636
 forward twice staggered sine 2635
 Trigonometric Transform interface 2633, 2636, 2640,
 2641, 2644, 2646, 2648, 3041
 code examples 3041
 routines
 ?_backward_trig_transform 2646
 ?_commit_trig_transform 2641
 ?_forward_trig_transform 2644
 ?_init_trig_transform 2640
 free_trig_transform 2648
 Trigonometric Transforms interface 2640
 Trunc 2333
 trust-region algorithm 3196
 TT interface 2633
 TT routines 2640
 two matrices
 QR factorization
 LAPACK 691
 ScaLAPACK 1789

U

Uniform (continuous) 2402
 Uniform (discrete) 2441
 UniformBits 2444
 unitary matrix 699, 731, 829, 889, 1579, 1580, 1582,
 1584, 1585, 1826, 1840, 2085, 2088, 2091,
 2094, 2097
 from LQ factorization
 LAPACK 1582
 ScaLAPACK 2091
 from QL factorization
 LAPACK 1579, 1585
 ScaLAPACK 2085, 2097
 from QR factorization
 LAPACK 1580
 ScaLAPACK 2088
 from RQ factorization
 LAPACK 1584
 ScaLAPACK 2094
 ScaLAPACK 1826, 1840
 Unpack 2342
 Unpack Functions 2257

updating
 rank-1
 general matrix 98
 Hermitian matrix 110, 117
 real symmetric matrix 127, 134
 rank-1, conjugated
 general matrix 100
 rank-1, unconjugated
 general matrix 102
 rank-2
 Hermitian matrix 112, 119
 symmetric matrix 129, 136
 rank-2k
 Hermitian matrix 170
 symmetric matrix 180
 rank-n
 Hermitian matrix 167
 symmetric matrix 177
 updating:rank-1
 Hermitian matrix
 packed storage 117
 real symmetric matrix
 packed storage 127
 updating:rank-2
 Hermitian matrix
 packed storage 119
 symmetric matrix
 packed storage 129
 upper Hessenberg matrix 829, 889, 1368, 1826, 2008
 1-norm value
 LAPACK 1368
 ScaLAPACK 2008
 Frobenius norm
 LAPACK 1368
 ScaLAPACK 2008
 infinity- norm
 LAPACK 1368
 ScaLAPACK 2008
 largest absolute value of element
 LAPACK 1368
 ScaLAPACK 2008
 ScaLAPACK 1826
 user time 1639

V

vector arguments 190, 2893, 2894
 array dimension 2893

vector arguments (*continued*)

- default 2894
- examples 2893
- increment 2893
- length 2893
- matrix one-dimensional substructures 2893
- sparse vector 190

vector conjugation 1240, 1951

vector indexing 2259

vector mathematical functions 2260, 2263, 2265, 2266, 2268, 2269, 2271, 2272, 2274, 2275, 2277, 2279, 2280, 2281, 2282, 2283, 2285, 2287, 2289, 2291, 2293, 2294, 2296, 2297, 2299, 2301, 2303, 2304, 2305, 2307, 2309, 2311, 2312, 2314, 2316, 2318, 2319, 2321, 2323, 2324, 2326, 2328, 2330, 2331, 2333, 2334, 2335, 2336, 2338

- absolute value 2272

- addition 2263

- complementary error function value 2326

- complex exponent of real vector elements 2304

- computing a rounded integer value and raising

- inexact result exception 2336

- computing a rounded integer value in current

- rounding mode 2335

- computing a truncated integer value 2338

- conjugation 2271

- cosine 2299

- cube root 2280

- denary logarithm 2296

- division 2275

- error function value 2324

- exponential 2291

- exponential of elements decreased by 1 2293

- four-quadrant arctangent 2312

- hyperbolic cosine 2314

- hyperbolic sine 2316

- hyperbolic tangent 2318

- inverse cosine 2307

- inverse cube root 2281

- inverse error function value 2328

- inverse hyperbolic cosine 2319

- inverse hyperbolic sine 2321

- inverse hyperbolic tangent 2323

- inverse sine 2309

- inverse square root 2279

- inverse tangent 2311

- inversion 2274

- multiplication 2268

vector mathematical functions (*continued*)

- multiplication of conjugated vector element 2269

- natural logarithm 2294

- natural logarithm of vector elements increased by 1 2297

- power 2285

- power (constant) 2287

- power $2/3$ 2282

- power $3/2$ 2283

- rounding to nearest integer value 2334

- rounding towards minus infinity 2330

- rounding towards plus infinity 2331

- rounding towards zero 2333

- sine 2301

- sine and cosine 2303

- square root 2277

- square root of sum of squares 2289

- squaring 2266

- subtraction 2265

- tangent 2305

Vector Mathematical Functions 2255

vector multiplication

- LAPACK 1603

- ScaLAPACK 2125

vector pack function 2339

vector statistics functions

- Bernoulli 2446

- Beta 2438

- Binomial 2450

- Cauchy 2423

- CopyStream 2384

- CopyStreamState 2385

- DeleteStream 2383

- dNewAbstractStream 2377

- Exponential 2413

- Gamma 2435

- Gaussian 2405

- GaussianMV 2408

- Geometric 2448

- GetBrngProperties 2465

- GetNumRegBrngs 2398

- GetStreamStateBrng 2397

- Gumbel 2432

- Hypergeometric 2452

- iNewAbstractStream 2375

- Laplace 2416

- LeapfrogStream 2389

- LoadStreamF 2388

- Lognormal 2429

vector statistics functions (*continued*)

- NegBinomial 2459
- NewStream 2372
- NewStreamEx 2373
- Poisson 2455
- PoissonV 2457
- Rayleigh 2426
- RegisterBrng 2464
- SaveStreamF 2386
- SkipAheadStream 2393
- sNewAbstractStream 2380
- Uniform (continuous) 2402
- Uniform (discrete) 2441
- UniformBits 2444
- Weibull 2420

vector unpack function 2342

vector-scalar product 83, 192

- sparse vectors 192

vectors

- adding magnitudes of vector elements 64
- copying 67
- dot product
 - complex vectors 73
 - complex vectors, conjugated 71
 - real vectors 68
- element with the largest absolute value 86
- element with the largest absolute value of real part and its index 1954
- element with the smallest absolute value 87
- Euclidean norm 74
- Givens rotation 77
- linear combination of vectors 65
- modified Givens transformation parameters 81
- rotation of points 75
- rotation of points in the modified plane 78
- sparse vectors 191
- sum of vectors 65
- swapping 85
- vector-scalar product 83

vml

- Functions Interface 2257
- Input Parameters 2258
- Output Parameters 2259

VML 2255

VML arithmetic functions 2263

VML exponential and logarithmic functions 2291

VML functions

- mathematical functions
 - Abs 2272

VML functions (*continued*)

mathematical functions (*continued*)

- Acos 2307
- Acosh 2319
- Add 2263
- Asin 2309
- Asinh 2321
- Atan 2311
- Atan2 2312
- Atanh 2323
- Cbrt 2280
- Ceil 2331
- CIS 2304
- Conj 2271
- Cos 2299
- Cosh 2314
- Div 2275
- Erf 2324
- Erfc 2326
- ErfInv 2328
- Exp 2291
- Expm1 2293
- Floor 2330
- Hypot 2289
- Inv 2274
- InvCbrt 2281
- InvSqrt 2279
- Ln 2294
- Log10 2296
- Log1p 2297
- Modf 2338
- Mul 2268
- MulByConj 2269
- NearbyInt 2335
- Pow 2285
- Pow2o3 2282
- Pow3o2 2283
- Powx 2287
- Rint 2336
- Round 2334
- Sin 2301
- SinCos 2303
- Sinh 2316
- Sqr 2266
- Sqrt 2277
- Sub 2265
- Tan 2305
- Tanh 2318
- Trunc 2333

VML functions (*continued*)

pack/unpack functions

Pack 2339

Unpack 2342

service functions

ClearErrorCallBack 2355

ClearErrStatus 2351

GetErrorCallBack 2354

GetErrStatus 2350

GetMode 2347

SetErrorCallBack 2351

SetErrStatus 2349

SetMode 2345

VML hyperbolic functions 2314

VML mathematical functions

arithmetic 2263

exponential and logarithmic 2291

hyperbolic 2314

power and root 2274

rounding 2330

special 2324

trigonometric 2299

VML Mathematical Functions 2257

VML Pack Functions 2257

VML Pack/Unpack Functions 2339

VML power and root functions 2274

VML rounding functions 2330

VML Service Functions 2344

VML special functions 2324

VML trigonometric functions 2299

VSL Fortran header 2357

VSL routines

advanced service subroutines

GetBrngProperties 2465

RegisterBrng 2464

convolution/correlation

CopyTask 2510

DeleteTask 2508

Exec 2498

Exec1D 2500

ExecX 2503

ExecX1D 2506

NewTask 2479

NewTask1D 2481

NewTaskX 2483

NewTaskX1D 2486

SetInternalPrecision 2492

generator subroutines

Bernoulli 2446

VSL routines (*continued*)generator subroutines (*continued*)

Beta 2438

Binomial 2450

Cauchy 2423

Exponential 2413

Gamma 2435

Gaussian 2405

GaussianMV 2408

Geometric 2448

Gumbel 2432

Hypergeometric 2452

Laplace 2416

Lognormal 2429

NegBinomial 2459

Poisson 2455

PoissonV 2457

Rayleigh 2426

Uniform (continuous) 2402

Uniform (discrete) 2441

UniformBits 2444

Weibull 2420

service subroutines

CopyStream 2384

CopyStreamState 2385

DeleteStream 2383

dNewAbstractStream 2377

GetNumRegBrngs 2398

GetStreamStateBrng 2397

iNewAbstractStream 2375

LeapfrogStream 2389

LoadStreamF 2388

NewStream 2372

NewStreamEx 2373

SaveStreamF 2386

SkipAheadStream 2393

sNewAbstractStream 2380

VSL routines:convolution/correlation

SetInternalDecimation 2495

SetMode 2490

SetStart 2494

W

Weibull 2420

Wilkinson transform 2084

X

xerbla 2793
xerbla, error reporting routine 59, 1640, 2260

Z

zlag2c 1623