*Stanford Exploration Project, Report 70, December 18, 1997, pages 1–58*

# Singular value decomposition on the Connection Machine

*Leonardo Rub*[1]

***keywords:*** *algorithm, computing*

**ABSTRACT**

The singular value decomposition of a matrix provides us with valuable information about the structure of the matrix. The matrices arising in geophysical data analysis are very large and consequently the resulting singular value decompositions are computationally expensive to obtain. The following implementation on a Connection Machine shows that with appropriate computer hardware and programming, matrix size will not be as much of an issue.

## INTRODUCTION

A singular value decomposition (SVD) of a real $m \times n$ matrix $A$ is its factorization into the product of three matrices :

$$A = U\Sigma V^T \tag{1}$$

where $U(m \times m)$ and $V(n \times n)$ are orthogonal matrices and $\Sigma(m \times n)$ is a diagonal matrix, whose diagonal elements are the singular values of $A$.

There are several ways of computing the SVD. With serial processing, variants of the QR algorithm (used to compute the eigenvalues of a matrix) can be used to compute the SVD. The basic approach involves the following (Golub and Van Loan, 1989) :

- form $C = A^T A$
- use the symmetric QR algorithm to compute $V_1^T C V_1 = diag(\sigma_i^2)$
- apply QR factorization with column pivoting to $AV_1$ obtaining $U^T(AV_1)\Pi = R$

[1]**email:** not available

With some manipulations, it can be shown that $R$ is diagonal, so we get : $\Sigma = R$ and $V = V_1 \Pi$, completing the SVD computation.

The drawback with this approach is that the formation of $A^T A$ can lead to a loss of information. A preferable method for computing the SVD is described in Golub and Kahan (1965). The symmetric QR algorithm is implicitly applied to $A^T A$ (in other words, the product $A^T A$ is never calculated) in order to find $U$ and $V$ simultaneously.

The approaches described above are very effective on serial processors, but they are not readily amenable to parallel implementations. With parallel processing, the preferred methods are variations of the Jacobi algorithm used to compute the eigenvalues of a symmetric matrix. The classical Jacobi algorithm (Jacobi, 1846) involves pre-multiplication and post-multiplication of $A$ by rotations of the form (Jacobi rotations) :

$$J(p,q,\theta) = \begin{pmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{pmatrix}$$

$c$ and $s$ are the cosine and sine of an angle $\theta$ determined in accordance with the following objective :

$$D = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}^T \begin{pmatrix} a_{pp} & a_{pq} \\ a_{qp} & a_{qq} \end{pmatrix} \begin{pmatrix} c & s \\ -s & c \end{pmatrix}$$

where $D$ is a diagonal matrix and $a_{pp}, a_{pq}, a_{qp}, a_{qq}$ are elements of $A$. With the resulting $\theta$, $B = J^T A J$ will agree with $A$ except in rows and columns $p$ and $q$, and is closer to diagonal form than $A$. Continuing this pre-multiplication and post-multiplication by matrices $J(p,q,\theta)$, using different $p$ and $q$, we can obtain an orthogonal diagonalization of $A$. This algorithm can be modified to obtain the SVD of an arbitrary $m \times n$ matrix $A$. A parallel implementation involves the storage of two columns of the matrix per processor. The determination of $\frac{n}{2}$ matrices $J(p,q,\theta)$ and post-multiplication can be done in parallel. The problem with this approach is that the transformation matrices $J$ have to move between processors in order to perform the premultiplication that affects the rows.

A parallel implementation using mesh connected processors is described in Brent, et al. (1985). In order to compute the SVD for an $m \times n$ matrix, $O(n^2)$ processors are needed. This requirement seriously limits the size of the matrices that can be decomposed on the Connection Machine (CM).

The method that is most suitable for implementation on the CM is described in Brent and Luk (1985). It requires $O(n)$ processors and time $O(mnS)$ where

$S \approx O(\log n)$ (typically $S \leq 10$). With this method there is no successive pre-multiplication of the matrix $A$ so the transformation matrices don't have to move between processors. This approach to the SVD computation is an implementation of Hestenes' method (Hestenes, 1958).

## HESTENES' METHOD

To compute the SVD of $A$, an orthogonal matrix $V$ is generated such that the transformed matrix $AV = W$ has orthogonal columns. Scaling each column of $W$ so that its 2-norm is unity we get :

$$W = \tilde{U}\tilde{\Sigma}$$

where $\tilde{U}$ is an $m \times n$ matrix and $\tilde{\Sigma}$ is an $n \times n$ nonnegative diagonal matrix. The corresponding SVD of $A$ is given by :

$$A = \tilde{U}\tilde{\Sigma}V^T \tag{2}$$

Comparing equation (2) with equation (1), we notice the different sizes of $U$ and $\tilde{U}$ as well as $\Sigma$ and $\tilde{\Sigma}$. However, since the null columns of $U$ and $\tilde{U}$ are associated with zero diagonal elements of $\Sigma$ and $\tilde{\Sigma}$, respectively, the non-null columns of $U$ and $\tilde{U}$ form orthogonal bases for $range(A)$ and there is no essential difference between equations (1) and (2). Successive Jacobi rotations are applied to $A$ in order to obtain $W$, giving us the following iteration :

$$A_{k+1} = A_k J_k$$

where $J_k = J(p, q, \theta)$ and $\theta$ is determined so that the columns of $p$ and $q$ of $A_{k+1}$ are orthogonal. Denoting column $p$ of $A_k$ as $a_p^{(k)}$, the requirements on $\theta$ can be stated as:

$$\begin{pmatrix} a_p^{(k+1)} & a_q^{(k+1)} \end{pmatrix} = \begin{pmatrix} a_p^{(k)} & a_q^{(k)} \end{pmatrix} \begin{pmatrix} c & s \\ -s & c \end{pmatrix}$$

where $c$ and $s$ are the cosine and sine of $\theta$, such that $a_p^{(k+1)T} a_q^{(k+1)} = 0$, leading to the following determination of $c$ and $s$ :

$$a_p^{(k+1)} = c a_p^{(k)} - s a_q^{(k)}$$

$$a_q^{(k+1)} = s a_p^{(k)} + c a_q^{(k)}$$

$$a_p^{(k+1)T} a_q^{(k+1)} = cs a_p^{(k)T} a_p^{(k)} + c^2 a_p^{(k)T} a_q^{(k)} - s^2 a_q^{(k)T} a_p^{(k)} - cs a_q^{(k)T} a_q^{(k)} = 0$$

Denoting $\alpha = a_p^{(k)T} a_p^{(k)}, \beta = a_q^{(k)T} a_q^{(k)}, \gamma = a_p^{(k)T} a_q^{(k)} = a_q^{(k)T} a_p^{(k)}$ we get :

$$cs\alpha + (c^2 - s^2)\gamma - cs\beta = 0$$

Dividing by $cs$ :

$$\alpha + \left(\frac{c}{s} - \frac{s}{c}\right)\gamma - \beta = 0$$

Denoting $t = \frac{s}{c}$ and $\xi = \frac{\beta - \alpha}{2\gamma}$ we get :

$$t^2 + 2\xi t - 1 = 0$$

giving us $t = \frac{sign(\xi)}{|\xi| + \sqrt{1+\xi^2}}, c = \frac{1}{\sqrt{1+t^2}}$ and $s = tc$.

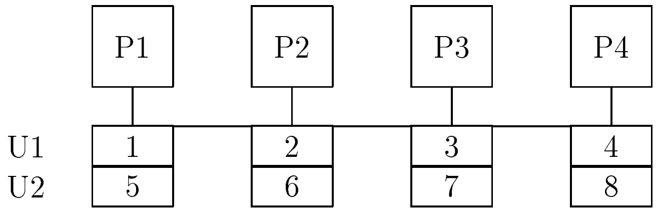
For every iteration we choose different column pairs $(p, q)$ and obtain a matrix $J_k$ that will orthogonalize those columns. As we are computing $W$, we can also compute $V$ beginning with the $n \times n$ identity matrix and performing the following with each iteration :

$$V_{k+1} = V_k J_k$$

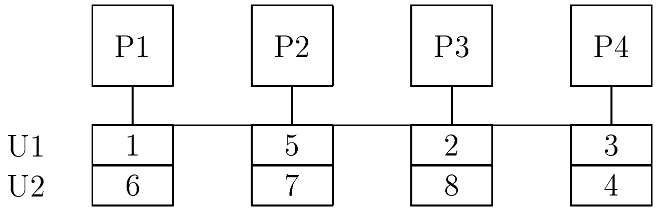
There are several ways of choosing the pairs of columns $(p,q)$ that are orthogonalized at every step. The approach used here is one that is suitable for parallel implementation on the CM.

## PARALLEL IMPLEMENTATION ON THE CM

In the discussion to follow we assume that $A$ has an even number of columns. If the number of columns is odd, we pad $A$ with a column of zeroes. We use $\frac{n}{2} + 1$ processors to store an $m \times n$ matrix $A_k$, where each processor stores two columns of $A_k$. Following the explanation given in Golub and Van Loan (1989), for purposes of illustration we assume that $A$ has 8 columns. Then we can store $A$ in the following way :




where $U1, U2$ are 2-dimensional arrays, $P_n$ is processor $n$ and the numbers indicate columns of $A_k$. Then we can orthogonalize the four sets of two columns in parallel, using the method described in the previous section. After the columns are orthogonalized, we keep column 1 fixed and rotate the ring defined by (2 3 4 8 7 6 5), obtaining the following configuration :




Illustrating the original layout of A as

| | | | | |
|---|---|---|---|---|
| U1 | 1 | 2 | 3 | 4 |
| U2 | 5 | 6 | 7 | 8 |
| buffer | X | X | X | X |

the rotation is achieved in six steps on the CM :

1 buffer = cshift(U2,dim=2,shift=-1)

| | | | | |
|---|---|---|---|---|
| U1 | 1 | 2 | 3 | 4 |
| U2 | 5 | 6 | 7 | 8 |
| buffer | 8 | 5 | 6 | 7 |

2 buffer(:,1:4:3) = U1(:,1:4:3)

| | | | | |
|---|---|---|---|---|
| U1 | 1 | 2 | 3 | 4 |
| U2 | 5 | 6 | 7 | 8 |
| buffer | 1 | 5 | 6 | 4 |

3 U1 = cshift(U1,dim=2,shift=-1)

4 U2 = cshift(U2,dim=2,shift=1)

| | | | | |
|---|---|---|---|---|
| U1 | 4 | 1 | 2 | 3 |
| U2 | 6 | 7 | 8 | 5 |
| buffer | 1 | 5 | 6 | 4 |

5 U1(:,1:2) = buffer(:,1:2)

| | | | | |
|---|---|---|---|---|
| U1 | 1 | 5 | 2 | 3 |
| U2 | 6 | 7 | 8 | 5 |
| buffer | 1 | 5 | 6 | 4 |

6 U2(:,4) = buffer(:,4)

| | | | | |
|---|---|---|---|---|
| U1 | 1 | 2 | 3 | 4 |
| U2 | 5 | 6 | 7 | 8 |
| buffer | 1 | 5 | 6 | 4 |

| | | | | |
|---|---|---|---|---|
| U1 | 1 | 5 | 2 | 3 |
| U2 | 6 | 7 | 8 | 4 |
| buffer | 1 | 5 | 6 | 4 |

The rotation can be achieved in less than six steps. We chose this approach because the only data movement between processors is done efficiently with cshift instructions.

The resulting column pairs can then be orthogonalized in parallel. After seven rotations we have processed all the column pairs in $A$. In general, for an $n$ column matrix, it takes $n-1$ steps to process all the column pairs. Each set of $n-1$ steps is referred as a sweep. Throughout the sweep we check the values of all $\lambda = \frac{|\gamma|}{\sqrt{\alpha\beta}}$,with $\alpha, \beta$ and $\gamma$ having been defined in the previous section. If $\lambda < \epsilon$ for a sufficiently small $\epsilon > 0$ holds for each set of $\alpha, \beta, \gamma$ in a sweep, then we can say that we have orthogonalized $A$ and obtained $W$. We can also obtain $V$ if we begin with an $n \times n$ identity matrix and at every step of the algorithm we process each $V_k$ in a manner similar to each $A_k$.

In Brent and Luk (1985), it is stated that although the authors know of no example for which the algorithm fails to give convergence, convergence is not guaranteed. In order to enforce convergence, a threshold approach (Wilkinson, 1965) can be chosen. A threshold value can be associated with each sweep. Any pair of columns in that sweep, with $\frac{|\gamma|}{\sqrt{\alpha\beta}}$ below the threshold, is not processed.

## CODING, TESTING AND PERFORMANCE EVALUATION

All the code needed for the SVD computation was written as a CM FORTRAN subroutine. The interface to a calling routine and the code listing are shown in the appendix. Even though the goal was to handle a general $m \times n$ matrix, all performance evaluations leading to this report were made on $n \times n$ matrices where $n$ is an integral power of 2. This choice of matrices has facilitated fine tuning on the CM and only minor modifications will be needed to handle general matrices once performance has been optimized with the restricted set.

All the testing was done on a CM with 128 physical processors (amounting to $4K$ virtual processors). Single precision was used throughout in order to be able to process as large matrices as possible. Even so, the largest matrices that can be processed are $1K \times 1K$ matrices. With the current data layout, performance improves with increasing matrix size. The best performance achieved so far has been 14 Mflops with $1K \times 1K$ matrices. The main reason accounting for this less than optimal performance is the time needed to shift the columns of the matrix after every orthogonalization step. However, the code optimization phase is not complete, and the following remains to be done :

- Modify the data layout so more than one processor is used to access a column pair. The immediate impact is an increase in the flop rate because more processors are used in the computations. In making this modification, special attention has to be given to column shifting with an eye for optimizing that function.
- Alter the column shifting procedure described in the previous section in order to minimize processor idle time. As mentioned before, this modification may have to be coordinated with the changes in data layout.

The flop rate should improve by a factor of three to five once the changes noted above are made.

## CONCLUSIONS

When implementing an apparently parallel algorithm, there are certain issues not emphasized in theoretical discussions which may play an unexpectedly important role in actual performance. In this implementation two such issues come to mind. The first one is the effect column shifting has on performance, even though it appears to be a fairly trivial task. The second one is the drastic effect data layout can have on performance on a CM.

In general, programming on the CM has required a struggle to get acquainted with the CM FORTRAN compiler's many idiosyncrasies. A great deal of effort was made to achieve the current less than optimal performance. However, once the optimization phase is complete, SEP will have an efficient program to obtain the SVD of medium to large matrices. At that point, running the program on a larger CM will allow processing of most matrices of interest in geophysical data analysis.

## ACKNOWLEDGMENTS

Comments from Professor Gene Golub (Stanford), Professor Frank Luk (Cornell), and specially Hongyuan Zha (Stanford) helped me identify the literature relevant to this undertaking. The students at SEP helped me get acquainted with SEP's computing environment. Last, but not least, Biondo Biondi (Thinking Machines) has provided guidance to achieve current performance levels, and suggestions to obtain substantially improved performance. Thank you all for your comments, explanations, and suggestions.

## REFERENCES

Brent, R.P., and Luk, F.T., 1985, The solution of singular value and symmetric eigenvalue problems on multiprocessor arrays: SIAM J. Sci. and Stat. Comp. **6**, 69–84.

Brent, R.P., Luk, F.T., and Van Loan, C.F., 1985, Computation of the singular value decomposition using mesh connected processors: J. VLSI Computer Systems **1**, 242–270.

Golub, G.H., and Kahan, W., 1965, Calculating the singular value and pseudo-inverse of a matrix: SIAM J. Num. Anal. Ser. **B 2**, 205–224.

Golub, G.H., and Van Loan, C.F., 1989, Matrix Computations: The John Hopkins University Press, Baltimore and London.

Hestenes, M.R., 1958, Inversion of matrices by biorthogonalization and related results: J. Soc. Indust. Appl. Math. **6**, 51–90.

Jacobi, C.G.J., 1846, Uber ein Leichtes Verfahren Die in der Theorie der Sacularstorungen Vorkommendern Gleichungen Numerisch Aufzulosen: Crelle's **30**, 51–94.

Wilkinson, J.H., 1965, The Algebraic Eigenvalue Problem: Clarendon Press, Oxford.

## APPENDIX A

### Interface to SVD computation subroutine

subroutine svdcomp(A,S,V,svord,m,n,isw,error)

A : Input matrix stored as linear array in row major order. On output contains matrix U in row major order

S : Output array of singular values

V : Output matrix V as linear array in row major order

svord : Indices pointing to elements of S in decreasing order, and to corresponding columns of A and V

isw : Number of sweeps needed to achieve convergence

error : 0 if convergence is achieved, 1 otherwise

## SVD computation subroutine

```
      subroutine svdcomp(A,S,V,svord,m,n,isw,error)
      implicit none
c
c     mxweep : maximum number of sweeps allowed
c     mxnupd : maximum number of implicit norm updates
c     epsilon : machine precision
c     tol : tolerance used for termination of algorithm
c
      integer mxweep,mxnupd
      real epsilon,tol
      parameter(mxweep=30)
      parameter(mxnupd=10)
      parameter(epsilon=0.5e-6,tol=5.0e-2)
c
c     Error code returned
c
      integer error
c
c     Dimensions of A
c
      integer m,n
c
c     SVD arrays
c     as input A contains the original matrix
c     as output A contains matrix U of SVD
c
      real A(*),S(*),V(*)
cmf$  layout A(:serial),S(:serial),V(:serial)
c
c     Ordering of singular values
c
      integer svord(*)
cmf$  layout svord(:serial)
c
c     CM arrays
c
c     U1,U2 : each one contains half the columns of A and an extra column
c             as a buffer for shifting
c     V1,V2 : each one contains half the columns of V and an extra column
c             as a buffer for shifting
c     alpha,beta : square of the norms of the columns of U1,U2
c                  will also hold sine theta, cosine theta for Jacobi
c                  transformations
c                  at the end of the run they will hold the singular values
c     gamma : inner product of columns in the same processor
c     inprod : normalized inner products used to determine if transformations
c            are applied based on thresholds. Also used to determine
c            termination of algorithm
c     buffer : holds temporary results
c     U1norm : squares of column 2-norms of U1
c     U2norm : squares of column 2-norms of U2
c     sing : stores singular values for transmission to front end and
```

```
c            for sorting in descending order
c     order : ordering of singular values
c
      real U1(m,(n+1)/2)
cmf$  layout U1(:serial,:news)
      real U2(m,(n+1)/2)
cmf$  layout U2(:serial,:news)
      real V1(m,(n+1)/2)
cmf$  layout V1(:serial,:news)
      real V2(m,(n+1)/2)
cmf$  layout V2(:serial,:news)
      real alpha((n+1)/2),beta((n+1)/2)
cmf$  layout alpha(:news),beta(:news)
      real gamma((n+1)/2),inprod((n+1)/2)
cmf$  layout gamma(:news),inprod(:news)
      real buffer(m,(n+1)/2)
cmf$  layout buffer(:serial,:news)
      real U1norm((n+1)/2),U2norm((n+1)/2)
cmf$  layout U1norm(:news),U2norm(:news)
      real sing(n)
cmf$  layout sing(:news)
      integer order(n)
cmf$  layout order(:news)
c
c     lc : last column in U1
c
      integer lc
c
c     sflag : true if stopping criterion has been met
c
      logical sflag
c
c     Temporary variables used in conjunction with the computation
c     of smax and smin
c
      real max1,max2,min1,min2
c
c     Loop counters
c     isw : sweep counter
c     irot : rotation counter
c     nupd : number of times implicit norm update has been used
c
      integer i,j,k,isw,irot,nupd
c
c     Timing variables
c
      integer itimer1,itimer2
      real flops
      real nsum,nsub,nmul,ndiv,nsqrt,nabs,nsign
      real wsum,wsub,wmul,wdiv,wsqrt,wabs,wsign
      data wsum/1.0/
      data wsub/1.0/
      data wmul/1.0/
      data wdiv/1.0/
```

```
      data wsqrt/1.0/
      data wabs/1.0/
      data wsign/1.0/
c
      data itimer1/1/
      data itimer2/2/
c
c     Initialize U1,U2 <- A
c
      lc = (n+1)/2
      do i=1,m-1,2
         call cmf_to_cm(U1(i,:),A(n*(i-1)+1),lc)
         call cmf_to_cm(U2(i,:),A(n*(i-1)+lc+1),n-lc)
         call cmf_to_cm(U1(i+1,:),A(n*i+1),lc)
         call cmf_to_cm(U2(i+1,:),A(n*i+lc+1),n-lc)
      enddo
      if (2*(m/2) .ne. m) then
         call cmf_to_cm(U1(m,:),A(n*(m-1)+1),lc)
         call cmf_to_cm(U2(m,:),A(n*(m-1)+lc+1),n-lc)
      endif
      if (lc .ne. n-lc) U2(:,lc) = 0.0
c
c     Initialize V1,V2 <- I
c
      V1 = 0.0
      V2 = 0.0
      forall (i=1:lc) V1(i,i) = 1.0
      forall (i=1:n-lc) V2(i+lc,i) = 1.0
c
c     Initialize looping variables
c
      isw = 0
      sflag = .false.
      nupd = mxnupd
c
c     Start timer 1
c
      call CM_timer_start(itimer1)
c
c     Loop over all sweeps
c
      do while (isw .lt. mxweep .and. .not. sflag)
         isw = isw + 1
         sflag = .true.
c
c     Loop over all rotations
c
         do irot=1,2*lc-1
c
c        compute inner products
c
            nupd = nupd + 1
            if (nupd .ge. mxnupd) then
               U1norm = 0.0
```

```
            U2norm = 0.0
            do i=1,m-1,2
               U1norm = U1norm + U1(i,:) * U1(i,:)
               U1norm = U1norm + U1(i+1,:) * U1(i+1,:)
               U2norm = U2norm + U2(i,:) * U2(i,:)
               U2norm = U2norm + U2(i+1,:) * U2(i+1,:)
            enddo
            if (2*(m/2) .ne. m) then
               U1norm = U1norm + U1(m,:) * U1(m,:)
               U2norm = U2norm + U2(m,:) * U2(m,:)
            endif
            nupd = 0
         endif
         gamma = 0.0
         do i=1,m-1,2
            gamma = gamma + U1(i,:) * U2(i,:)
            gamma = gamma + U1(i+1,:) * U2(i+1,:)
         enddo
         if (2*(m/2) .ne. m) gamma = gamma + U1(m,:) * U2(m,:)
c
c       If any of the normalized inner products is greater than
c       the tolerance we must execute the next sweep after this one
c
         where (U1norm .eq. 0.0) gamma = 0.0
         where (U2norm .eq. 0.0) gamma = 0.0
         where (gamma .eq. 0.0)
            inprod = 0.0
         elsewhere
            inprod = abs(gamma / sqrt(U1norm * U2norm))
         endwhere
         if (any(inprod .gt. tol)) sflag = .false.
c
c       In this set of calculations alpha and beta are used for
c       temporary storage and at the end they will have the
c       contents of the Jacobi rotation matrices
c       gamma is also used for temporary storage
c
         where (inprod .gt. epsilon)
c
c       eta = (U2norm - U1norm) / (2.0 * gamma) stored in alpha
c
            alpha = (U2norm - U1norm) / (2.0 * gamma)
c
c       t = sign(eta)/(abs(eta) + sqrt(1.0+eta^2)) stored in beta
c
            beta = sign(1.0,alpha) / (abs(alpha) +
     *                   sqrt(1.0+alpha*alpha))
c
c       Update norms
c
            U1norm = U1norm - beta * gamma
            U2norm = U2norm + beta * gamma
c
c       cos(theta) = 1.0/sqrt(1.0*t^2) stored in alpha
```

```
c
               alpha = 1.0/sqrt(1.0+beta*beta)
c
c       sin(theta) = t * cos(theta) stored in beta
c
               beta = beta * alpha
            elsewhere
               alpha = 1.0
               beta = 0.0
            endwhere
c
c       Apply one sided Jacobi transformations, use gamma variable as buffer
c
            do i=1,m-1,2
               gamma = U1(i,:) * alpha - U2(i,:) * beta
               U2(i,:) = U1(i,:) * beta + U2(i,:) * alpha
               U1(i,:) = gamma
               gamma = U1(i+1,:) * alpha - U2(i+1,:) * beta
               U2(i+1,:) = U1(i+1,:) * beta + U2(i+1,:) * alpha
               U1(i+1,:) = gamma
               gamma = V1(i,:) * alpha - V2(i,:) * beta
               V2(i,:) = V1(i,:) * beta + V2(i,:) * alpha
               V1(i,:) = gamma
               gamma = V1(i+1,:) * alpha - V2(i+1,:) * beta
               V2(i+1,:) = V1(i+1,:) * beta + V2(i+1,:) * alpha
               V1(i+1,:) = gamma
            enddo
            if (2*(m/2) .ne. m) then
               gamma = U1(m,:) * alpha - U2(m,:) * beta
               U2(m,:) = U1(m,:) * beta + U2(m,:) * alpha
               U1(m,:) = gamma
               gamma = V1(m,:) * alpha - V2(m,:) * beta
               V2(m,:) = V1(m,:) * beta + V2(m,:) * alpha
               V1(m,:) = gamma
            endif
c
c       Shifting when lc > 2
c
c           call CM_timer_start(itimer2)
            if (lc .gt. 2) then
c
c       Shift the columns of U1,U2,V1,V2 to obtain next
c       combination of columns in sweep
c
               buffer = cshift(U2,dim=2,shift=-1)
               buffer(:,1) = U1(:,1)
               buffer(:,lc) = U1(:,lc)
               U1 = cshift(U1,dim=2,shift=-1)
               U2 = cshift(U2,dim=2,shift=1)
               U1(:,1:2) = buffer(:,1:2)
               U2(:,lc) = buffer(:,lc)
c
               buffer = cshift(V2,dim=2,shift=-1)
               buffer(:,1) = V1(:,1)
```

```
            buffer(:,lc) = V1(:,lc)
            V1 = cshift(V1,dim=2,shift=-1)
            V2 = cshift(V2,dim=2,shift=1)
            V1(:,1:2) = buffer(:,1:2)
            V2(:,lc) = buffer(:,lc)

c        Shift squares of column norms U1norm, U2norm

            gamma = cshift(U2norm,dim=1,shift=-1)
            gamma(1) = U1norm(1)
            gamma(lc) = U1norm(lc)
            U1norm = cshift(U1norm,dim=1,shift=-1)
            U2norm = cshift(U2norm,dim=1,shift=1)
            U1norm(1:2) = gamma(1:2)
            U2norm(lc) = gamma(lc)
         elseif (lc .eq. 2) then
c
c       Shift when lc = 2
c
            buffer = cshift(U2,dim=2,shift=-1)
            U2(:,2) = U1(:,2)
            U1(:,2) = buffer(:,2)
            U2(:,1) = buffer(:,1)
c
            buffer = cshift(V2,dim=2,shift=-1)
            V2(:,2) = V1(:,2)
            V1(:,2) = buffer(:,2)
            V2(:,1) = buffer(:,1)
c
            gamma = cshift(U2norm,dim=1,shift=-1)
            U2norm(2) = U1norm(2)
            U1norm(2) = gamma(2)
            U2norm(1) = gamma(1)
         endif
c        flops = 0.0
c        Call MY_stop_timer('shift '//char(0),itimer2,0,flops)
      enddo
   enddo
c
c  Timing information
c
   nsum = wsum * float(m*2*lc + (m+3)*lc + 2*(m-1)*lc/mxnupd)
   nsub = wsub * float(m*2*lc + 4*lc)
   nmul = wmul * float(m*lc*(9+2/mxnupd) + 7*lc)
   ndiv = wdiv * float(4*lc)
   nsqrt = wsqrt * float(3*lc)
   nabs = wabs * float(2*lc)
   nsign = wsign * float(lc)
   flops = float(isw * (2*lc-1)) *
  *                   (nsum+nsub+nmul+ndiv+nsqrt+nabs+nsign)
   call MY_stop_timer('main loop '//char(0),itimer1,0,flops)
   if (sflag) then
      error = 0
   else
```

```
            error = 1
         endif
c
c          Compute singular values and U
c

         buffer = U1 * U1
         alpha = sqrt(sum(buffer,dim=1))
         buffer = spread(alpha,dim=1,ncopies=m)
         where (buffer .ne. 0.0) U1 = U1 / buffer
c
         buffer = U2 * U2
         beta = sqrt(sum(buffer,dim=1))
         buffer = spread(beta,dim=1,ncopies=m)
         where (buffer .ne. 0.0) U2 = U2 / buffer
c
c          Set output arrays A,S,V to U,SIGMA,V
c
         do i=1,m
            call cmf_from_cm(A(n*(i-1)+1),U1(i,:),lc)
            call cmf_from_cm(A(n*(i-1)+lc+1),U2(i,:),n-lc)
         enddo
c
         sing(1:lc) = alpha
         sing(lc+1:n) = beta(1:n-lc)
         call cmf_from_cm(S,sing,n)
c
         do i=1,n
            call cmf_from_cm(V(n*(i-1)+1),V1(i,:),lc)
            call cmf_from_cm(V(n*(i-1)+lc+1),V2(i,:),n-lc)
         enddo
c
c        Generate singular value ordering vector svord
c
         call cmf_order(order,sing,1,.true.)
         do i=1,n
            svord(n+1-order(i)) = i
         enddo
         return
         end
```