Appendix: Man Pages

Man page: Axis

```
AXIS(3LO)            Linop Reference Manual             AXIS(3LO)

NAME
     Axis - class to describe an axis of a <type>space.

SYNOPSIS
     #include <Axis.h>

DESCRIPTION
     The class Axis contains  a  description  of  an  axis  in  a
     <type>space. The description includes the initial coordinate
     of the data along the axis (origin), the number  of  samples
     along  this  axis  (length),  the  increment between samples
     (delta), a label for this axis (name), and flags to indicate
     wildcarding  for  origin, delta and length (wildo, wildd and
     wildl). An Axis can be constructed  from  either  the  above
     information, or from a SepAxis *.

PUBLIC VARIABLES
     float origin
     float delta
     int length
     char * name
     int wildo
     int wildd
     int wildl

PUBLIC OPERATIONS
     Axis(float origin,float delta,int length,const char *name,
          int wildo=,int wildd=,int wildl=)
          Construct an  Axis  given  a  value  for  each  of  its
          members. The label (name) is copied.

     Axis(SepAxis *)
          Construct an Axis which matches the SepAxis.

     SepAxis * sepaxis() const
          Return a pointer to a new SepAxis object, which matches
          the Axis.

     Axis()
     Axis(const Axis &)
     Axis& operator=(const Axis &)
     ~Axis()

     int operator==(const Axis &) const
          Check if two Axis have the same values  for  all  their
          members that are not wildcarded.

     friend ostream& operator<<(ostream&, const Axis&)
          Print the Axis.

     Axis wild() const
```

Return an Axis with name, origin, delta and length

equal to zero. Serves as a zero Axis.

int iswild() const
    Check if the Axis is a wild axis. That is, name, origin, delta and length are zero.

int subaxis(Axis & a) const
    Check if the Axis is a sub-axis of a. For Axis x to be a sub-axis of Axis y, x must: () have the same label as y (2) have every sample of x have the same position as some sample of y.

float max() const
    Return origin + delta * length.

int index(float) const
    Get the integer index of the closest sample to the float parameter.

float value(int) const
    Return the float position of the sample numbered by the int parameter.

SEE ALSO
    Axislist, <type>space, SepAxis

Man page: Axislist

```
AXISLIST(3LO)        Linop Reference Manual        AXISLIST(3LO)

NAME
     Axislist - class to  contain  the  axis  information  for  a
     <type>space.

SYNOPSIS
     #include <Axislist.h>

DESCRIPTION
     An Axislist contains the axis information for a <type>space,
     including the number of dimensions, and 4 Axis objects. This
     sets the maximum on dimensions at four, and also means  that
     any  axes  up  to  four will exist, even if the dimension is
     lower than 4. An Axislist can be created by initializing  it
     with  the  number  of  dimensions,  and  then assigning Axis
     objects into it, or from a SepInfo *.

PUBLIC VARIABLES
     int ndim
          The number of dimensions.

     Axis list[4]
          The four Axis.

PUBLIC OPERATIONS
     Axislist(int dim)
          Create an Axislist of dimension  dim,  with  all  empty
          Axis.

     Axislist(SepInfo *)
          Construct an Axislist matching the SepInfo.

     SepInfo * sepinfo(int esize)
          Return  a  pointer  to  a  new  SepInfo  matching   the
          Axislist, given the element size.

     Axislist(const Axis & a)
          Construct an Axislist of dimension , with the a as the
          first Axis in list.

     Axislist()
     Axislist(const Axislist &)

     Axislist(const Axislist& old,int wildo,int wildd,int wildl)
          Construct an Axislist the same as old,  with  wildcards
          of every Axis set to the specified values.

     void setorigin(float o,float o,float o2,float o3)
     void setorigin(float o,float o,float o2)
     void setorigin(float o,float o)
     void setdelta(float d,float d,float d2,float d3)
     void setdelta(float d,float d,float d2)
     void setdelta(float d,float d)
```

AXISLIST(3LO)   Linop Reference Manual   AXISLIST(3LO)

```
     void setlength(int l,int l,int l2,int l3)
     void setlength(int l,int l,int l2)
     void setlength(int l,int l)
          Set the origin, delta, or  length  for  each  Axis  for
          which  there  is  a  value  specified  (first  argument
          corresponds to list[], second to list[], etc.).

     Axislist & operator=(const Axislist &)
     ~Axislist()

     friend ostream& operator<<(ostream&,const Axislist&)
          Print the Axislist.
```

Axis & operator[](int) const
Get an Axis by directly indexing the Axislist, rather than accessing member list.

Axis & operator[](const char *) const
Get an Axis from the Axislist by specifying its name.

Axislist transpose() const
Return an Axislist that is the transpose of this one. The first two Axis are switched, which can add a dimension or subtract one, if one of the switching Axis objects are empty.

Axislist transpose(int i,int j=,int k=2,int l=3) const
Return an Axislist that is transposed according to the int parameters. list[] get the Axis number i, list[] gets Axis[j], etc. The dimension is then set according to the greatest non-empty Axis.

void cleanup()
Set the dimension according to the greatest non-empty Axis.

int operator==(const Axislist &) const
Check if two Axislist objects match- if they have the same dimension and all their Axis objects match.

int iswild() const
Check if all the Axis in the list are wild.

Axislist slice(int dim, int num) const
Return an Axislist that describes slice number num along the dim dimension.

int subspace(const Axislist & a) const
Check if this is a subspace of a. To be a subspace the two must have the same dimension, and all the Axis in the list are sub-axes of the matching Axis in a.list.

SEE ALSO
Axis, <type>space, SepInfo

Man page: converter

```
ByteConverter(UI)    Cube Reference Manual    ByteConverter(UI)

NAME
     ByteConverter - base class for doing byte convertion

SYNOPSIS
     #include <Cube/converter.h>

DESCRIPTION
     A ByteConverter transforms an array of float into  an  array
     of  unsigned  char  ready  for  a  raster representation.  A
     ByteConverter is an  element  of  a  SepData  object.   This
     object  stores  internally variables to determined the plot-
     ting parameters such as the percentage  of  clip,  the  tpow
     value, the gpow half way percentile phalf.

PUBLIC OPERATIONS
     ByteConverter(int n,float o,float d,int n2)
          Construct a byte converter object with  the  definition
          of  the  time  axis and the number of element along the
          second axis.

     unsigned char * Byte(int zoPix,int ndPix,int pcPix,
                          int ncPix,float *d)
     unsigned char * Byte(float *d)
          Do the convertion of the input data float into unsigned
          char.  The  pixels  value corresponding to the non data
          value, to the zero data value,  to  the  positive  clip
          value, to the negative clip value may be arguments.

     void SetTpow(float)
     float GetTpow()
          Set or get the tpow value.  The clip, gpow  depends  on
          the  current  tpow  value and are recomputed if tpow is
          changed.

     void SetClip(float)
     float GetClip()
          Set or Get the absolute  clip  value.  A  default  clip
          value is computed from the pclip.

     void SetPclip(float)
     float GetPclip()
          Set or Get the  percentage  of  clipping.  The  default
          value of the clip is 99%.

     void SetGpow(float)
     float GetGpow()
          Set or Get the absolute gpow value.  A  gpow  value  is
          computed from the gpow half way percentile (phalf).

     void SetPhalf(float)
     float GetPhalf()
          Set or Get the gpow half way  percentile.  The  initial
```



```
          value of the phalf is 85%.

PROTECTED OPERATIONS
     void FindExtrem(float *d,int n)
          Find minimum and maximum of the input data.

     void FindMean(float *d,int n)
          Find the absolute mean value.

     void FindClip(float *d,int n)
          Find the absolute clip value from the pclip value  from
          the input data.  The input data must have been multiply
```

by the tpow table and symetrized around the mean.

void FindGpow(float *d,int n)
Find the gpow value from the parameter phalf, from the clip value clip with the input data.

void FindTpow()
Set the tpow value to .

void BuildGpowTable(int *gpowTable,int ngpow,int zp, int pcp,int ncp)
Build a gpow table of dimension ngpow with the input pixels value: zp zero pixel value, pcp positif clip pixel value, ncp negative clip pixel value.

void BuildTpowTable(float *tpowTable,int n,float o,float d)
Build a tpow table of dimension n with the data origin o and the data increment of d. The tpow value has been set by using the function SetTpow.

void SubSample(int n,int n2,int n3,float *d,float *sd, int nn,int nn2,int nn3,int j, int j2,int j3,int f,int f2,int f3)
Subsample the data d beginning at j,j2,j3 every j,j2,j3 samples and output nn,nn2,nn3 samples.

void TpowSubSample(float *ind,float *out,int n,int n2, int j,int j2,float *tpow)
Compute an subsample array out from the input array ind multiplying elements by a tpow table.

float Cent(float pcent,float *data,int n)
Compute the centile of the input data.

PROTECTED VARIABLES

o,d,n,n2
Description of the data

in tzoPix, ndPix, pCPix, nCPix
anchor pixel values

float * tpowTable, *gpowTable
Intern convertion table.

float dmin,dmax,dmean:
data maxima. data mean.

float clip, pclip:
clip value. clip percentile.

float gpow, float phalf, float half:
half way value for gpow. signed power. gpow half way percentile.

float tpow:
coordinate scale factor.

SepMask flags:
flags recording user definitions and computation stage.

SEE ALSO

Seplib, intro

Man page: datastore

```
DataStor(UI)        Cube Reference Manual        DataStor(UI)

NAME
     DataStore - base class  for  a  common  interface  for  disk
     object or memory objects

SYNOPSIS
     #include <Cube/datastore.h>

DESCRIPTION
     A DataStore provide a common protocol to get  and  put  data
     either from or to memory or disk.

PUBLIC OPERATIONS
     DataStore()
          Construct an uninitialized dataStore.

     DataStore(unsigned long len)
          Construct a DataStore with a certain length.

     virtual ~DataStore()
          Delete the data if a MemDataStore, Close and remove the
          file if a DiskDataStore.

     unsigned long Length()
          Return the length of the dataStore.

     virtual void SetLength( unsigned long len )
          Set the length. This will open  the  file  or  allocate
          memory  depending  on  the type of data store. If some-
          thing already exists it will be copied in the beginning
          of the new storage.

     virtual void* Get( unsigned long offs, unsigned long num )
          Get a copy of the data from offset offs, of length num.

     virtual void Put(void* dat,unsigned long offs,
                     unsigned long num)
          Put the data at the offset offs on  a  length  of  num.
          MemDataStore does a copy into the internal buffer.

     virtual void ReadFromFile(FILE* f,unsigned long offs,
                              unsigned long num)
     virtual void WriteToFile( FILE* f, unsigned long offs,
                              unsigned long num )
          Read or Write data from the given file descriptor . The
          data are read and put into the storage. Or the data are
          retrieved from storage and written.

     virtual void CopyData( DataStore * from, unsigned long fromoffs,
     unsigned long tooffs, unsigned long num)
          Copy a portion of the data from another dataStore.

     virtual void Zero()

DataStor(UI)        Cube Reference Manual        DataStor(UI)

          Initialize all the data to zero.

     virtual void* Data()
          Get the data. DiskDataStore returns nil.

     virtual void SetData(void*)
          Set the Data. DiskDataStore copies this  data  to  disk
          and then deletes the data.

DERIVED CLASS
     MemDataStore()
     MemDataStore( unsigned long len )
```

MemDataStore(DataStore*)
Construct a Memory data store. If the length is given the memory is allocated. If a DataStore is given the data are copied from the input data store.

DiskDataStore()
DiskDataStore(unsigned long len)

DiskDataStore(DataStore*)
Construct a Disk data store. If a length is given the file is open and seek to this position to really allocate the disk space. If a DataStore is given the data are copied from the input data store.

SEE ALSO
InterViews, intro

Man page: op

NAME

<type>op - A virtual base class for linear operators operating from <type>space to <type>space.

SYNOPSIS

```
#include <<type>op.h>
```

DESCRIPTION

The class <type>op is a virtual base class derived from <type>operator defining the operations that must be defined for a linear operators operating from <type>space to <type>space. These include a constructor, a destructor (which does not always need to be written), a Forward function, and an Adjoint function. The last two take a <type>space and return a <type>space. In addition, the <type>op base class will allow a derived operator to cycle over the <type>spaces of a <type>spacearray automatically.

Something to notice is there is no copy constructor for a <type>op. This is because all classes derived from <type>op can be referred to as a <type>op. The result of this is that <type>ops cannot be copied or assigned, or passed by functions. A function can be passed a pointer or reference to a <type>op, but not the <type>op itself. In addition, a <type>op cannot be a return value, nor can a reference to a <type>op be a return value.

PUBLIC OPERATIONS

```
virtual <type>space Forward(const <type>space &) const =
virtual <type>space Adjoint(const <type>space &) const =
```

These must be written for an operator derived from <type>op.

```
<type>spacearray Forward(const <type>spacearray &) const
<type>spacearray Adjoint(const <type>spacearray &) const
```

These cause an operator to automatically cycle over the <type>spaces in a <type>spacearray, applying itself to each one.

SEE ALSO

<type>operator, <type>space, <type>spacearray, <type>opone

Man page: opadjoint

```
OPADJOINT(3LO)       Linop Reference Manual       OPADJOINT(3LO)

NAME
     <type>opAdjoint  -  Class  to  take   the   adjoint   of   a
     <type>operator.

SYNOPSIS
     #include <<type>opadjoint.h>

DESCRIPTION
     <type>opAdjoint is a class to  automatically  handle  taking
     the  adjoint  of a <type>operator. When a <type>opAdjoint is
     made from a <type>operator, it  calls  the  Adjoint  of  the
     <type>operator  when  Forward  is called, and the Forward of
     the <type>operator  when  Adjoint  is  called.  Notice  that
     <type>opAdjoint    contains    only   a   pointer   to   the
     <type>operator, and has no copy  or  assignment  operations.
     This means that a <type>opAdjoint must be declared, not used
     as a temporary. Also, the  <type>operator  that  is  in  the
     <type>opAdjoint  must be maintained (not changed or deleted)
     for the <type>opAdjoint to work.

PUBLIC OPERATIONS
     <type>opAdjoint(<type>operator &)
     <type>opAdjoint(<type>operator *)
          Create a <type>opAdjoint from the operator.

     friend <type>opAdjoint operator!(<type>operator &)
          Return a <type>opAdjoint created from the operator.

     <type>opAdjoint operator=(const <type>operator &)

     ~<type>opAdjoint()

     <type>spacearray Forward(const <type>spacearray &) const
     <type>spacearray Adjoint(const <type>spacearray &) const

PRIVATE VARIABLE
     <type>operator * op

SEE ALSO
     <type>spacearray, <type>operator
```

Man page: oparray

```
OPARRAY(3LO)         Linop Reference Manual         OPARRAY(3LO)

NAME
     <type>opArray - Class that is an array of operators.

SYNOPSIS
     #include <<type>oparray.h>

DESCRIPTION
     Class <type>opArray is an array of operators, derived  from
     <type>operator,  which can be applied to a <type>spacearray.
     Both a <type>opArray and a <type>spacearray have two  dimen-
     sions, and applying a <type>opArray to a <type>spacearray is
     handled like matrix multiplication. That is, the rows of the
     <type>opArray   are   multiplied   by  the  columns  of  the
     <type>spacearray, and this is summed along.

     The  <type>opArray  does   not   contain   copies   of   the
     <type>operators,  but  rather  pointers  to  them. Thus, the
     <type>operators  cannot  be  changed  or  deleted,  or   the
     <type>opArray will behave strangely.

PUBLIC OPERATIONS
     <type>opArray(int i,int j=)
          Construct a <type>opArray, with the number of rows  and
          columns specified by i and j.

     <type>opArray(const <type>opArray &)
          Create a new <type>opArray with copied pointers to  the
          <type>opArray.

     <type>opArray & operator=(const <type>opArray &)
          Set the pointers to <type>operators equal to  those  of
          the <type>opArray.

     ~<type>opArray()

     const <type>operator & operator()(int i,int j=)
          Return a reference to the  <type>operator  at  position
          (i,j) in the <type>opArray.  This <type>operator cannot
          be changed, but can be applied to a spacearray.

     void set(int i, int j, <type>operator &)
     void set(int i, int j, <type>operator *)
          Set the position (i,j) to the reference or pointer to a
          <type>operator.     The     method     for     creating   a
          <type>opArray is define its size, and then  assign  the
          <type>operators into it.  A pointer to a <type>operator
          with allocated space should be used when the array will
          not be applied within the same scope as it is created.

     void setrow(int row, <type>operator &)
     void setrow(int row, <type>operator *)
          Set the entire row to the reference  or  pointer  to  a

OPARRAY(3LO)         Linop Reference Manual         OPARRAY(3LO)

          <type>operator.

     void setcol(int col, <type>operator &)
     void setcol(int col, <type>operator *)
          Set the entire column to the reference or pointer to  a
          <type>operator.

     <type>spacearray Forward(const <type>spacearray &) const
     <type>spacearray Adjoint(const <type>spacearray &) const
          These are the functions for applying a <type>opArray to
          a  <type>spacearray.  In  the  case  of Forward, if the
          <type>opArray has the same number  of  columns  as  the
```

<type>spacearray has rows, then the rows of the <type>opArray are multiplied by the columns of the <type>spacearray, and summed. This is just like matrix multiplication. Adjoint does the same, except with the transpose of the <type>opArray.

PROTECTED VARIABLES

```
int rows
int cols
<type>operator ** ops
```

PROTECTED OPERATION

```
void unref()
```

SEE ALSO

<type>space, <type>spacearray, <type>operator

Man page: opchain

```
OPCHAIN(3LO)         Linop Reference Manual         OPCHAIN(3LO)

NAME
     <type>opChain - class to store products of operators.

SYNOPSIS
     #include <<type>opchain.h>

DESCRIPTION
     Class <type>opChain is  class  derived  from  <type>operator
     which   allows   for   automatic  handling  of  products  of
     <type>operators. Two <type>operators can be joined  to  pro-
     duce  a  <type>opChain  which,  when the Forward function is
     called,   applies   the   second   <type>operator   to   the
     <type>space,  followed  by  the  first  <type>operator.  The
     Adjoint function calls the Adjoint of  the  <type>operators,
     in  reverse  order. One thing that is important to notice is
     that the <type>opChain contains pointers to <type>operators,
     not    the    <type>operators    themselves.    Thus,    the
     <type>operators must be maintained (not changed or deleted),
     or the <type>opChain will not work.

     Derived classes may construct chains longer  than  length  2
     (without  defining them as chains of chains).  num specifies
     the number of operators.  Operators are applied  in  reverse
     order (last first, second to last second, and so on).

PUBLIC OPERATIONS
     <type>opChain(<type>operator &,<type>operator &)
     <type>opChain(<type>operator *,<type>operator *)
          Create   an   <type>opChain   containing   the    two
          <type>operators.

     friend <type>opChain operator*(<type>operator&,<type>operator&)
          Return  a  <type>opChain  built from the two operators.
          This  is  an   alternative   method   of   creating   a
          <type>opChain.

     <type>spacearray Forward(const <type>spacearray & x) const
     <type>spacearray Adjoint(const <type>spacearray & x) const

     <type>opChain(const <type>opChain &)
     <type>opChain & operator=(const <type>opChain &)
     ~<type>opChain()

PROTECTED VARIABLES
     <type>operator **ops
     int num

SEE ALSO
     <type>spacearray, <type>operator
```

Man page: opdiagarray

```
OPDIAGARRAY(3LO)     Linop Reference Manual      OPDIAGARRAY(3LO)

NAME
     <type>opDiagarray - Class that is a diagonal array of opera-
     tors.

SYNOPSIS
     #include <<type>opdiagarray.h>

DESCRIPTION
     Class  <type>opDiagarray  is  derived  from   <type>opArray.
     Given a single <type>operator and an int in construction, it
     creates a n*n <type>opArray with that <type>operator as  the
     diagonal  and  <type>opEmpty elsewhere.  Values of the array
     may be changed using the <type>opArray set operation.   Note
     that this is not the optimized version of diagonal arrays of
     operators (see <type>opDiagonal).

PUBLIC OPERATIONS
     <type>opDiagarray(int n, <type>operator &)
     <type>opDiagarray(int n, <type>operator *)

SEE ALSO
     <type>opArray,       <type>operator,        <type>opEmpty,
     <type>opDiagonal
```

Man page: opdiagonal

```
OPDIAGONAL(3LO)      Linop Reference Manual      OPDIAGONAL(3LO)

NAME
     <type>opDiagonal - Optimized diagonal array of operators.

SYNOPSIS
     #include <<type>oparray.h>

DESCRIPTION
     Class <type>opDiagonal acts  like  a  diagonal  fopArray  of
     operators.  It  is derived from <type>operator, NOT fopArray
     because it only stores and applies the operators  along  the
     diagonal.   The  functions  supplied  result  in  the  same
     behavior as those for fopArrays, except that  indexing  only
     requires  one  int  value  instead of two (since the row and
     column number  are  always  equal  for  diagonal  elements).
     Non-diagonal  positions  can neither be accessed nor set and
     are treated as "zero" operators.

     The  <type>opDiagonal  does  not  contain  copies   of   the
     <type>operators,  but  rather  pointers  to  them. Thus, the
     <type>operators  cannot  be  changed  or  deleted,  or   the
     <type>opArray will behave strangely.

PUBLIC OPERATIONS
     <type>opDiagonal(int i)
          Construct a <type>opDiagonal, with the number  of  rows
          and columns specified by i.

     <type>opDiagonal(int i, foperator &)
     <type>opDiagonal(int i, foperator *)
          Construct from the row/column number and an operator to
          insert at every diagonal position.

     <type>opDiagonal(const <type>opDiagonal &)
          Create a new <type>opDiagonal with copied  pointers  to
          the <type>opDiagonal.

     <type>opDiagonal & operator=(const <type>opDiagonal &)
          Set the pointers to <type>operators equal to  those  of
          the <type>opDiagonal.

     ~<type>opDiagonal()

     const <type>operator & operator()(int i)
          Return a reference to the  <type>operator  at  position
          (i,i)  in  the  array.   This  <type>operator cannot be
          changed, but can be applied to a spacearray.

     void set(int i, <type>operator &)
     void set(int i, <type>operator *)
          Set the position (i,i) to the reference or pointer to a
          <type>operator.  A  pointer  to  a  <type>operator with
          allocated space should be used when the array will  not

OPDIAGONAL(3LO)      Linop Reference Manual      OPDIAGONAL(3LO)

          be applied within the same scope as it is created.

     <type>spacearray Forward(const <type>spacearray &) const
     <type>spacearray Adjoint(const <type>spacearray &) const
          These are the functions for applying a <type>opDiagonal
          to  a  <type>spacearray. In the case of Forward, if the
          <type>opDiagonal has the same number of columns as  the
          <type>spacearray  has  rows,  the  function  behaves as
          though the rows of the <type>opDiagonal are  multiplied
          by  the  columns  of  the <type>spacearray, and summed.
          This is just like matrix multiplication.  Adjoint  does
```

```
          the same, except with the transpose of the
          <type>opDiagonal.

PRIVATE VARIABLE
     int size
          <type>operator ** ops

PRIVATE OPERATION
     void unref()

SEE ALSO
     <type>opArray, <type>space, <type>spacearray, <type>operator
```

Man page: opdiagscale

```
NAME
     <type>opDiagScale - Operator to  scale  each  element  of  a
     <type>space by a constant.

SYNOPSIS
     #include <<type>opdiagscale.h>

DESCRIPTION
     <type>opDiagscale is a <type>op  which  multiplies  a  whole
     <type>space  by  a  <type>space  of constants.  The constant
     <type>space is either supplied in the constructor, or if two
     <type>spaces  are  supplied, the input <type>space is multi-
     plied by sqrt(sp/sp2)

PUBLIC OPERATIONS
     <type>opDiagscale(const <type>space&)
          Create a  <type>opDiagscale  from  the  <type>space  by
          which to multiply.

     <type>opDiagscale(const <type>space& sp,const <type>space& sp2)
          Create  a <type>opDiagscale from two <type>spaces.  The
          constant <type>space to multiply by is sqrt(sp/sp2)

     <type>space Forward(const <type>space &) const
     <type>space Adjoint(const <type>space &) const

PRIVATE OPERATIONS
     void apply(<type>Array &,<type>Array &,int,
                const Axislist &,const Axislist &) const

PRIVATE VARIABLE
     <type>space scale

SEE ALSO
     <type>space, <type>op
```

Man page: opdotprod

NAME

<type>opdotprod - Function that performs the dot product test on a operator.

SYNOPSIS

```
#include <<type>opdotprod.h>
```

DESCRIPTION

```
<type> <type>opdotprod(const Axislist & in,const Axislist
                       const <type>operator &,long seed=)
```

This function tests a <type>operator to make sure that its Forward and Adjoint functions really are adjoint to each other. The function takes in, the input Axislist of the <type>operator, and out, the output Axislist, and the <type>operator. It returns the fractional difference in the two results given by the test. This should be just round-off error in the <type>operator, near machine precision. If it is fairly large, the operator is probably coded wrong.

<type>opdotprod uses drand48 to generate the random numbers for the test. If a seed is passed to the function, it is used to seed the random number generator using srand48; otherwise, the generator is unseeded.

SEE ALSO

<type>space, <type>operator

Man page: opempty

NAME
<type>opEmpty - Operator to return an empty (wild) <type>space.

SYNOPSIS
```
#include <<type>opempty.h>
```

DESCRIPTION
<type>opEmpty is a <type>op which creates a wild Axis and returns a <type>space created from it. The Adjoint function behaves the same as Forward. In addition, subtraction, and multiplication (but not division) with other <type>spaces, these <type>spaces are treated as "zero."

PUBLIC OPERATIONS
```
<type>opEmpty()

<type>space Forward(const <type>space &) const
<type>space Adjoint(const <type>space &) const
```

SEE ALSO
<type>space, <type>op, <type>opDiagonal

Man page: operator

OPERATOR(3LO) Linop Reference Manual OPERATOR(3LO)

NAME

<type>operator - A virtual base class for all linear operators operating from <type>spacearray to <type>spacearray.

SYNOPSIS

#include <<type>operator.h>

DESCRIPTION

The class <type>operator is a virtual base class defining the operations that must be defined for all linear operators operating from <type>spacearray to <type>spacearray. These include a constructor, a destructor (which does not always need to be written), a Forward function, and a Adjoint function operating from <type>spacearray to <type>spacearray. All derived classes must define Forward and Adjoint. Any derived class is provided with functions to get the <type>Array or Axislist from a <type>space, and the rows and cols from a <type>spacearray.

The <type>operator class also handles reference counting. When an object of a subclass is constructed, its reference count is automatically initialized. When a <type>operator is to be deleted, it should have only reference to it. In the current implementation, no warning is given if this is not the case. <type>operators such as <type>oparray, <type>opchain and <type>optrans, which contain multiple references to the same <type>operator, internally handle reference counting. When pointers are explicitly assigned to already existing <type>operators by the user, ref() must be called to increment the count, and deref() should be called in the place of delete.

Something to notice is there is no copy constructor for a <type>operator. This is because all classes derived from <type>operator can be referred to as a <type>operator. The result of this is that <type>operators cannot be copied or assigned, or passed by functions. A function can be passed a pointer or reference to a <type>operator, but not the <type>operator itself. In addition, a <type>operator cannot be a return value, nor can a reference to a <type>operator be a return value.

PUBLIC OPERATIONS

<type>operator()
    Set references to one for a new <type>operator.

void ref()
    Increment references.

void deref()
    Decrement references and deletes <type>operator if no references remain.

OPERATOR(3LO) Linop Reference Manual OPERATOR(3LO)

virtual ˜<type>operator()
    Decrements references.

virtual <type>spacearray Forward(const <type>spacearray &) const =
virtual <type>spacearray Adjoint(const <type>spacearray &) const =
    These must be written for an operator derived from <type>operator.

PROTECTED OPERATIONS

int rows(const <type>spacearray &) const

```
int cols(const <type>spacearray &) const
     A class derived from <type>op gets  special  access  to
     the  private  members of <type>spacearray through these
     functions- it can get (but not change)  the  number  of
     rows and columns.

<type>Array & getdata(const <type>space &) const
     A class derived from <type>op also gets special  access
     to  the  private members of <type>space. getdata allows
     access to the <type>Array in a <type>space,  which  can
     be changed with this function.

Axislist & getinfo(const <type>space &) const
     Allow access to the Axislist in  a  <type>space,  which
     can be changed with this function.

Note that getdata and getinfo return  references.   Attempts
to   use   the  return  values  outside  the  scope  of  the
<type>space will result in loss of  data  and  unpredictable
behavior.
```

PRIVATE VARIABLES

```
int references
```

SEE ALSO

<type>op, <type>spacearray, <type>opChain, <type>opTrans, <type>opArray, <type>opPatch

Man page: opinter

```
NAME
     <type>opInterp - Operator to interpolate data.

SYNOPSIS
     #include <<type>opinter.h>

DESCRIPTION
     <type>opInterp takes a <type>space, and changes  the  length
     of  one  of the axes. In doing this, it maps one sample into
     each of  the  points  in  the  output  space,  interpolating
     linearly  to  determine  which  point  of the input space is
     used. To do this the opposite way (mapping each point of the
     input  space to the output space), use the Adjoint function.
     To construct a <type>opInterp, you simply specify the  input
     Axis, and the output Axis.

PUBLIC OPERATIONS
     <type>opInterp(const Axis& in,const Axis& out)
          Construct a <type>Interp given the in and out Axis.

     <type>space Forward(const <type>space &) const
     <type>space Adjoint(const <type>space &) const

PRIVATE OPERATION
     void apply(<type>Array &,<type>Array &,int,
                 const Axislist &,const Axislist &) const

SEE ALSO
     <type>space, <type>op, <type>opone
```

Man page: opmask

```
OPMASK(3LO)          Linop Reference Manual          OPMASK(3LO)

NAME
     <type>opMask - Operator to mask a <type>space along an axis.

SYNOPSIS
     #include <<type>opmask.h>

DESCRIPTION
     <type>opMask is a <type>opone  which  masks  a  <type>space,
     meaning  it  sets some of the values equal to zero. The axis
     to mask along, and the positions along this axis to  set  to
     zero, are specified in the constructor. The Adjoint function
     does the same thing as the Forward function.

PUBLIC OPERATIONS
     <type>opMask(const Axis &,const int *)
          Construct a <type>opMask given the Axis to mask  along,
          and an int array which has zeros at the positions which
          are to be set to zero. The int array is copied into the
          operator.

     <type>opMask(const Axis &,const <type>space &)
          Construct a <type>opMask given the Axis to mask  along,
          and a <type>space used to mask.

     ~<type>opMask()

     <type>space Forward(const <type>space &) const
     <type>space Adjoint(const <type>space &) const

PRIVATE VARIABLE
     int * mask

PRIVATE OPERATION
     void apply(<type>Array &,<type>Array &,int,
                 const Axislist &,const Axislist &) const

SEE ALSO
     <type>space, <type>opone
```

Man page: opmatmul

```
OPMATMUL(3LO)        Linop Reference Manual        OPMATMUL(3LO)

NAME
     <type>opMatmul - <type>opwhich performs a matrix multiply.

SYNOPSIS
     #include <<type>opmatmul.h>

DESCRIPTION
     This class is an operator, derived from  <type>opone,  which
     multiplies  a  <type>space by a <type>space, or <type>Array,
     in the manner of matrix multiply.  When  you  construct  the
     <type>Matmul,  you specify the left side matrix, and the two
     axes, one to contract and the other to expand. The  Axis  to
     contract  along  matches  the  columns of the left side, the
     Axis to expand, the rows. When the <type>Matmul  is  applied
     to  a  <type>space,  through the Forward function, it multi-
     plies the <type>space by the left-side, along the  contract-
     ing Axis.

PUBLIC OPERATIONS
     <type>opMatmul(const <type>space &)
          Construct a <type>Matmul with the  <type>space  as  the
          matrix to multiply by.

     <type>opMatmul(const Axis & in,const Axis & out,
               const <type>Array &)
          Construct a <type>Matmul by specifying the Axis to con-
          tract (in), expand (out), and the <type>Array to multi-
          ply by.

     <type>space Forward(const <type>space &) const
     <type>space Adjoint(const <type>space &) const

PRIVATE VARIABLE
     <type>Array matrix

PRIVATE OPERATION
     void apply(<type>Array &,<type>Array &,int,
            const Axislist &,const Axislist &) const

SEE ALSO
     <type>space, <type>op, <type>opone
```

Man page: opmerge

```
OPMERGE(3LO)         Linop Reference Manual          OPMERGE(3LO)

NAME
     <type>opMerge - Operator to merge  vector  <type>spacearrays
     into <type>spaces.

SYNOPSIS
     #include <<type>opmerge.h>

DESCRIPTION
     <type>opMerge is derived  from  <type>operator.  tt  Forward
     merges  a  vector <type>spacearray into a <type>space (a x
     <type>spacearray). This new <type>space has one more  dimen-
     sion  than  the elements of the vector <type>spacearray. The
     length of the Axis to this extra dimension must be equal  to
     the  length  of  the  Axis given to the constructor. Adjoint
     slices a <type>space at every sample along the given Axis to
     form  a  vector  <type>spacearray  of  spaces with one fewer
     dimension.

PUBLIC OPERATIONS
     <type>opmerge(const Axis &)
          Constructor to define the number of rows that the  vec-
          tor <type>spacearray must have.

     <type>spacearray Forward(const <type>spacearray &) const
     <type>spacearray Adjoint(const <type>spacearray &) const

PROTECTED VARIABLES
     Axis extra

SEE ALSO
     <type>spacearray, <type>space, <type>operator, Axis
```

Man page: opone

```
OPONE(3LO)          Linop Reference Manual          OPONE(3LO)

NAME
     <type>opone - base class for most linear operators.

SYNOPSIS
     #include <<type>opone.h>

DESCRIPTION
     The <type>opone class is a class derived from  <type>op,  to
     be  used as a base class that makes writing linear operators
     easier. It handles creating the output space, arranging  the
     input space (through transposes) so that the axes are in the
     right order for calculating, and cycling over any extraneous
     axes.  Thus,  to  write  an  operator  which is derived from
     <type>opone, you need only write the constructors, the  des-
     tructor  (if  one  is needed), and the apply function, to do
     the calculation.

PUBLIC OPERATIONS
     <type>opone()

     <type>opone(const Axislist &,const Axislist &)
          Create  an  <type>opone  given  an  input  and   output
          Axislist for the operator.

     <type>space Forward(const <type>space &) const
     <type>space Adjoint(const <type>space &) const
          These functions call the functions which rearrange  the
          input  space,  create  the  output  space,  cycle  over
          extraneous axes, and finally call  the  apply  routine.
          Thus,  in writing an operator derived from <type>opone,
          everything is handled  except  for  the  apply  routine
          itself.

     <type>spacearray Forward(const <type>spacearray &) const
     <type>spacearray Adjoint(const <type>spacearray &) const
          These are simply  passed  through  to  <type>op,  which
          calls  Forward  or  Adjoint for each <type>space in the
          <type>spacearray.

     virtual Axislist forwardinfo(const Axislist &) const
     virtual Axislist adjointinfo(const Axislist &) const
          These functions  are  called  by  Forward  or  Adjoint,
          respectively, given the input <type>space Axislist, and
          return the Axislist of the output <type>space.

PROTECTED VARIABLES
     Axislist inaxis
          An Axislist which describes the  input  axes  that  are
          required  of  a <type>space for Forward to be called on
          it. Any axes that the <type>space has that are  not  in
          inaxis will be cycled over.

OPONE(3LO)          Linop Reference Manual          OPONE(3LO)

     Axislist outaxis
          An Axislist  describing  the  axes  that  Forward  will
          create.  Also,  any  axes  that are cycled over will be
          stuck on the back of the output Axislist.

     The two previous definitions can be reversed  for  when  the
     Adjoint function is called.

PROTECTED OPERATIONS
     virtual void forwarddata(<type>Array& in,<type>Array& out,
                              const Axislist & outlist,
                              const Axislist & inlist) const
          Called by Forward to process the input <type>Array.  It
```

can be overwritten by an operator derived from <type>opone, but if not, it calls the apply function.

```
virtual void forwardtransform(Axislist & out,
                              const Axislist & in) const
```

Copy the incoming Axislist into the outgoing one or calls forwardwildtrans if wildcards are present.

```
virtual forwardwildtrans(Axislist & out,
                         const Axislist & in) const
```

Called by forwardtransform to create the outgoing Axislist when wildcards are present. It MUST be defined by any derived class that uses wildcards. When the function exits, all no wildcarded values may remain.

```
virtual void adjointdata(<type>Array in&,<type>Array& out,
                         const Axislist & out
                         const Axislist & in) const
virtual void adjointtransform(Axislist & out,
                              const Axislist & in) const
virtual void adjointwildtrans(Axislist & out,
                              const Axislist & in) const
```

Same as above, except for Adjoint.

```
virtual void apply(<type>Array& a,<type>Array& b,
                   int adj, Axislist & alist,
                   const Axislist& blist) const
```

The apply function is meant to be overwritten by a derived class. When called by Forward, adj=, a is the input <type>Array, and b is the output. When called by Adjoint, adj=, a is the output, and b is the input.

PRIVATE OPERATIONS

```
<type>space forwardorder(const <type>space &) const
void forwardhelp(int,int,int,const Axislist &,<type>Array &,
                 <type>Array &,const Axislist &) const
type>space adjointorder(const <type>space &) const
void adjointhelp(int,int,int,const Axislist &,<type>Array &,
```



```
                 <type>Array &,const Axislist &) const

int wildcard(const Axislist &) const
```

DERIVED CLASSES

To write an operator derived from <type>opone, you must:

. Write a constructor that defines inaxis, outaxis, and initializes any internal variables.

2. If the constructor used new or malloc on any variable, write a destructor to delete or free these variables.

3. Write the apply routine to do the work of the operator. For adj=, the Forward case, apply will take a <type>Array matching inaxis, and return a <type>Array matching outaxis. For adj=, the Adjoint case, a <type>Array matching inaxis must be created from a <type>Array matching outaxis.

4. Write forwardwildtrans and adjointwildtrans if wildcards will be used.

SEE ALSO

<type>space, <type>op, <type>Array, Axislist

Man page: oponepatch

```
OPONEPATCH(3LO)      Linop Reference Manual       OPONEPATCH(3LO)

NAME
     <type>opOnepatch - Operator to make a 2-D patch from  a  2-D
     <type>space

SYNOPSIS
     #include <type>oponepatch.h

DESCRIPTION
     The <type>opOnePatch  class  is  derived  from  <type>opone.
     Forward  takes an input <type>space and specifications for a
     patch and returns a smaller <type>space of that  description
     if  it  is a subspace.  Adjoint returns a<type>space of full
     size containing the patch and zeros elsewhere.

     <type>opOnepatch3 is the equivalent 3-D operator.

PUBLIC OPERATIONS
     <type>opOnepatch(const Axislist& spacelist,float origini,
                      int leni,float originj,int lenj)
          Construct from axislist of the  full  space  and  patch
          origin and length along each axis.

     <type>opOnepatch(float origini,int leni,float originj,
                      int lenj,const Axislist & patchlist)
          Construct from axislist of the  patch  and  full  space
          origin and length along first axis and second axis

     <type>opOnepatch(const Axislist & spacelist,float i,
                      float j, float i, float j)
          Construct from patch corner coordinates and full  space
          axislist

     <type>opOnepatch(const Axislist & spacelist,
                      const Axislist & patchlist)
          Construct from full space and patch axislists.

     <type>space Forward(const <type>space> &) const

     <type>space Adjoint(const <type>space> &) const
          Construct from full space and patch axislists

PRIVATE OPERATIONS
     void apply(<type>Array &,<type>Array &,int,const Axislist&,
                const Axislist &) const

SEE ALSO
     <type>opone,         <type>opPatch,         <type>opUnpatch,
     <type>opOnepatch3
```

Man page: oponepatch3

```
OPONEPATCH3(3LO)     Linop Reference Manual      OPONEPATCH3(3LO)

NAME
     <type>opOnepatch3 - Operator to make a 3-D patch from a  3-D
     <type>space

SYNOPSIS
     #include <type>oponeptch3.h

DESCRIPTION
     The <type>opOnePatch3 class  is  derived  from  <type>opone.
     Forward  takes an input <type>space and specifications for a
     patch and returns a smaller <type>space of that  description
     if  it is a subspace.  Adjoint returns a <type>space of full
     size containing the patch and zeros elsewhere.

     <type>opOnepatch is the equivalent 2-D operator.

PUBLIC OPERATIONS
     <type>opOnepatch3(const Axislist& spacelist,float origini,
                      int leni,float originj,int lenj,
                      float origink,int lenk)
          Construct from axislist of the  full  space  and  patch
          origin and length along each axis.

     <type>opOnepatch3(float origini,int leni,float originj,
                      int lenj,float origink,int lenk,
                      const Axislist& patchlist)
          Construct from axislist of the  patch  and  full  space
          origin and length along each axis.

     <type>opOnepatch3(const Axislist & spacelist,float i,
                      float j,float k,float i,float j,float k)
          Construct  from patch corner coordinates and full space
          axislist.

     <type>opOnepatch3(const Axislist & spacelist,
                      const Axislist & patchlist)
          Construct from full space and patch axislists

     <type>space Forward(const <type>space> &) const

     <type>space Adjoint(const <type>space> &) const

PRIVATE OPERATIONS
     void apply(<type>Array &,<type>Array &,int,const Axislist&,
                const Axislist &) const

SEE ALSO
     <type>opone,       <type>opPatch3,       <type>opUnpatch3,
     <type>opOnepatch
```

Man page: oppad

```
OPPAD(3LO)           Linop Reference Manual           OPPAD(3LO)

NAME
     <type>opPad - Operator to pad or truncate a <type>space.

SYNOPSIS
     #include <<type>oppad.h>

DESCRIPTION
     <type>opPad is a <type>opone to pad or  truncate  along  one
     axis  of a <type>space. Padding involves adding zeros on one
     end, truncating  shortens  the  data.  In  constructing  the
     <type>opPad, the number of samples to pad or truncate at the
     front and back are specified. The Adjoint function does  the
     opposite,  that  is, pads where Forward truncates, and trun-
     cates where Forward pads.

PUBLIC OPERATIONS
     <type>opPad(const Axis &,int front,int back)
          Construct a <type>opPad which  operates  on  the  Axis,
          padding  front  number  of zeros to the front, and back
          zeros to the back. If front or back  are  negative,  it
          truncates instead of padding (also by that number).

     <type>opPad(const char *name,int front,int back)
          Construct from the name of the Axis to operate  on  and
          the number to pad or truncate in the front and back.

     <type>space Forward(const <type>space &) const
     <type>space Adjoint(const <type>space &) const

PRIVATE VARIABLES
     int front
     int back

PRIVATE OPERATIONS
     void forwardwildtrans(Axislist&,const Axislist&) const
     void adjointwildtrans(Axislist&,const Axislist&) const

     void apply(<type>Array &,<type>Array &,int,
                const Axislist &,const Axislist &) const

SEE ALSO
     <type>space, <type>opone
```

Man page: oppatch

```
OPPATCH(3LO)         Linop Reference Manual         OPPATCH(3LO)

NAME
     <type>opPatch - Operator to do 2-D patching

SYNOPSIS
     #include <<type>oppatch.h>

DESCRIPTION
     The <type>opPatch  class  is  derived  from  <type>operator.
     Forward  takes a single space, wrapped in a <type>spacearray
     and uses <type>opOnepatch to convert it to a one-dimensional
     <type>spacearray  of  smaller  spaces, whose number and size
     are specified in the constructor.  The spacing of patches is
     such that the patches always end evenly at edges of the ori-
     ginal <type>space and overlap is approximately even. If  the
     input   <type>spacearray   contains   more   than  a  single
     <type>space, only the (,) <type>space is patched  and  the
     others   are   ignored.   Adjoint  takes  a  one-dimensional
     <type>spacearray  of  patched  <type>spaces  and  returns  a
     <type>spacearray containing a single <type>space.

     It is important to note that the Adjoint  case  is  not  the
     inverse of the Forward case because of possible overlapping.
     Weighted patching in the <type>opUnpatch  operator  must  be
     used   as   the   inverse.   The   inverse  of  Forward  of
     <type>opPatch is  Adjoint  of  <type>opUnpatch.  Adjoint  of
     <type>opPatch is the inverse of Forward of <type>opUnpatch.

     Equivalent operators exist for  three-dimensional  patching:
     <type>opPatch3 and <type>opUnpatch3.

PUBLIC OPERATIONS
     <type>opPatch()
     <type>opPatch(const <type>opPatch &)

     <type>opPatch(int numi,int numj,int sizei,
                   int sizej,const Axislist &)
          Construct from the number and  size  of  patches  along
          each Axis and the Axislist of the single <type>space.

     <type>spacearray Forward(const <type>spacearray &) const
     <type>spacearray Adjoint(const <type>spacearray &) const

PROTECTED VARIABLES
     int numi
     int numj
     int sizei
     int sizej
     Axislist wallaxes

SEE ALSO
     <type>operator,      <type>opOnepatch,      <type>opUnpatch,
     <type>opPatch3, <type>opUnpatch3, <type>opOnepatch3
```

Man page: oppatch3

```
OPPATCH3(3LO)       Linop Reference Manual       OPPATCH3(3LO)

NAME
     <type>opPatch3 - Operator to do 3-D patching

SYNOPSIS
     #include <<type>opptch3.h>

DESCRIPTION
     The <type>opPatch3 class  is  derived  from  <type>operator.
     Forward  takes a single space, wrapped in a <type>spacearray
     and   uses   <type>opOnepatch3   to   convert   it   to    a
     <type>spacearray  of  smaller  spaces, whose number and size
     are specified in the constructor.  The spacing of patches is
     such that the patches always end evenly at edges of the ori-
     ginal <type>space and overlap is approximately even. If  the
     input   <type>spacearray   contains   more   than  a  single
     <type>space, only the (,) <type>space is patched  and  the
     others  are  ignored.   Adjoint  takes a <type>spacearray of
     patched <type>spaces and returns a <type>spacearray contain-
     ing a single <type>space.

     It is important to note that the Adjoint  case  is  not  the
     inverse of the Forward case because of possible overlapping.
     Weighted patching in the <type>opUnpatch3 operator  must  be
     used   as   the   inverse.   The   inverse  of  Forward  of
     <type>opPatch3 is Adjoint of  <type>opUnpatch3.  Adjoint  of
     <type>opPatch3    is    the    inverse    of    Forward   of
     <type>opUnpatch3.

     Equivalent operators  exist  for  two-dimensional  patching:
     <type>opPatch and <type>opUnpatch.

PUBLIC OPERATIONS
     <type>opPatch3()
     <type>opPatch(const <type>opPatch &)

     <type>opPatch3(int numi,int numj,int numk,int sizei,
                    int sizej,int sizek,const Axislist &)
          Construct from the number and  size  of  patches  along
          each Axis and the Axislist of the single <type>space.

     <type>spacearray Forward(const <type>spacearray &) const
     <type>spacearray Adjoint(const <type>spacearray &) const

PROTECTED VARIABLES
     int numi
     int numj
     int numk
     int sizei
     int sizej
     int sizek
     Axislist wallaxes

OPPATCH3(3LO)       Linop Reference Manual       OPPATCH3(3LO)

SEE ALSO
     <type>operator,    <type>opOnepatch3,    <type>opUnpatch3,
     <type>opPatch, <type>opUnpatch, <type>opOnepatch
```

Man page: opscale

```
OPSCALE(3LO)         Linop Reference Manual          OPSCALE(3LO)

NAME
     <type>opScale - Operator to scale a <type>space.

SYNOPSIS
     #include <<type>opscale.h>

DESCRIPTION
     <type>opScale  is  a  <type>op  which  multiplies  a   whole
     <type>space  by  a <type>. The Adjoint function divides by a
     <type>.

PUBLIC OPERATIONS
     <type>opScale(<type> value)
          Create a <type>opScale which multiples or divides by  a
          value.

     <type>space Forward(const <type>space &) const
     <type>space Adjoint(const <type>space &) const

PRIVATE VARIABLE
     <type> factor

SEE ALSO
     <type>space, <type>op
```

Man page: opshift

NAME

<type>opShift - Operator to shift a <type>space along an axis.

SYNOPSIS

```
#include <<type>opshift.h>
```

DESCRIPTION

<type>opShift is a <type>opone which shifts one of the axes of a <type>space, changing the origin. It does nothing to the data.

PUBLIC OPERATIONS

```
<type>opShift(const Axis& inaxis,int shift)
```

Create a <type>opShift which moves the origin of the Axis by shift positions. If shift is positive, the axis is advanced (the origin increases), otherwise the axis regresses.

```
<tpye>opShift(const char *axname,int shift)
```

Same as above, except the Axis is specified by name only.

```
<type>space Forward(const <type>space&) const
<type>space Adjoint(const <type>space&) const
```

PRIVATE OPERATIONS

```
void apply(<type>Array &,<type>Array &,int,
            const Axislist &,const Axislist &) const

void forwardwildtrans(Axislist&,const Axislist&) const
void adjointwildtrans(Axislist&,const Axislist&) const
```

PRIVATE VARIABLE

```
int shift
```

SEE ALSO

<type>space, <type>opone

Man page: opstack

```
OPSTACK(3LO)         Linop Reference Manual         OPSTACK(3LO)

NAME
     <type>opStack - Applies a reduction or expansion  along  one
     axis

SYNOPSIS
     #include <<type>opstack.h>

DESCRIPTION
     When applied forwards it adds  an  extra  dimension  to  the
     space,  the  data  is replicated across the extra dimension.
     When applied adjoint it  removes  that  dimension  from  the
     space  by  summing over the extra dimension. The constructor
     is given the axis to be stacked or spread over  and  option-
     ally the position at which it is inserted. If no position is
     specified the forward operator inserts the extra axis as the
     first axis of the space.

PUBLIC OPERATORS
     <type>opStack(const Axis & inax,int pos=-)
          Supply axis to be stacked over and optionally the posi-
          tion to insert it

     <type>spacearray Forward(const <type>spacearray& x) const
     <type>spacearray Adjoint(const <type>spacearray& x) const

PROTECTED OPERATIONS
     <type>space Forward(const <type>space &) const
          Replicate over axis.

     <type>space Adjoint(const <type>space &) const
          Stack over axis

PROTECTED VARIABLES
     Axis stackax;
     int pos;

SEE ALSO
     <type>op, <type>spacearray, <type>space, Axis
```

Man page: opunpatch

```
OPUNPATCH(3LO)       Linop Reference Manual       OPUNPATCH(3LO)

NAME
     <type>opUnpatch - Operator to do weighted 2-D patching

SYNOPSIS
     #include <<type>opunpatch.h>

DESCRIPTION
     The <type>opUnpatch class  is  derived  from  <type>opPatch.
     Forward  takes a single space, wrapped in a <type>spacearray
     and   uses   <type>opOnepatch   to   convert   it   to   a
     <type>spacearray  of  smaller  spaces, whose number and size
     are specified in the constructor.  The spaces  are  weighted
     according  to their overlap.  The spacing of patches is such
     that the patches always end evenly at edges of the  original
     <type>space  and overlap is approximately even. If the input
     <type>spacearray contains more than  a  single  <type>space,
     only  the  (,)  <type>space  is patched and the others are
     ignored.   Adjoint  takes  a  <type>spacearray  of   patched
     <type>spaces,  weights  them  and returns a <type>spacearray
     containing a single <type>space.

     Forward uses the formula
          Patches=WindWeight*(<type>opPatch.Forward(WallWeight*space))
     Adjoint uses the formula
          space=WallWeight*(<type>opPatch.Adjoint(WindowWeight*Patches))

     WindowWeight is a weighting function that may be overwritten
     by derived classes and the WallWeight is calculated automat-
     ically.

     <type>opUnpatch is  intended  as  an  inverse  operator  for
     <type>opPatch.   Adjoint of the former is inverse to Forward
     of the latter and vice versa.

     3-D unpatching is carried out by opUnpatch3.

PUBLIC OPERATORS
     <type>opUnpatch(const <type>opPatch &)
          Construct   from   same   specifications    as    given
          <type>opPatch

     <type>opUnpatch(int numi,int numj,int sizei,
                     int sizej,const Axislist &)
          Construct from patch number and size specifications and
          Axislist of the single space.

     <type>spacearray Adjoint(const <type>spacearray &) const
     <type>spacearray Forward(const <type>spacearray &) const

PROTECTED OPERATORS
     virtual <type>spacearray MakeWindWt
                                (const <type>spacearray &) const

OPUNPATCH(3LO)       Linop Reference Manual       OPUNPATCH(3LO)

          Given a <type>spacearray with <type>spaces of the  same
          sizes  and  order  as  the  <type>spacearray of patches
          created by <type>opPatch, return a <type>spacearray  of
          <type>spaces  of  weights  to  match  each  patch.   By
          default, all weights are one.  This can be  overwritten
          by derived classes to change the weighting function.

PRIVATE OPERATORS
     <type>space MakeWallWt(const <type>spacearray &) const
     <type>spacearray makeFWWindWt() const

SEE ALSO
```

`<type>operator`, `<type>opPatch`, `<type>opOnepatch`, `<type>opUnpatch3`, `<type>opPatch3`,

Man page: opunpatch3

```
OPUNPATCH3(3LO)      Linop Reference Manual      OPUNPATCH3(3LO)

NAME
     <type>opUnpatch3 - Operator to do weighted 3-D patching

SYNOPSIS
     #include <<type>opunptch3.h>

DESCRIPTION
     The <type>opUnpatch class is  derived  from  <type>opPatch3.
     Forward  takes a single space, wrapped in a <type>spacearray
     and   uses   <type>opOnepatch   to   convert   it   to   a
     <type>spacearray  of  smaller  spaces, whose number and size
     are specified in the constructor.  The spaces  are  weighted
     according  to their overlap.  The spacing of patches is such
     that the patches always end evenly at edges of the  original
     <type>space  and overlap is approximately even. If the input
     <type>spacearray contains more than  a  single  <type>space,
     only  the  (,)  <type>space  is patched and the others are
     ignored.  Adjoint  takes  a  <type>spacearray  of   patched
     <type>spaces,  weights  them  and returns a <type>spacearray
     containing a single <type>space.

     Forward uses the formula
          Patches=WindWeight*(<type>opPatch3.Forward(WallWeight*space))
     Adjoint uses the formula
          space=WallWeight*(<type>opPatch3.Adjoint(WindowWeight*Patches))

     WindowWeight is a weighting function that may be overwritten
     by derived classes and the WallWeight is calculated automat-
     ically.

     <type>opUnpatch is  intended  as  an  inverse  operator  for
     <type>opPatch3.  Adjoint of the former is inverse to Forward
     of the latter and vice versa.

     2-D unpatching is carried out by opUnpatch.

PUBLIC OPERATORS
     <type>opUnpatch3(const <type>opPatch3 &)
          Construct   from   same   specifications    as    given
          <type>opPatch3

     <type>opUnpatch3(int numi,int numj,int numk,int sizei,
                      int sizej,int sizek,const Axislist &)
          Construct from patch number and size specifications and
          Axislist of the single space.

     <type>spacearray Adjoint(const <type>spacearray &) const
     <type>spacearray Forward(const <type>spacearray &) const

PROTECTED OPERATORS
     virtual <type>spacearray MakeWindWt
                              (const <type>spacearray &) const

OPUNPATCH3(3LO)      Linop Reference Manual      OPUNPATCH3(3LO)

          Given a <type>spacearray with <type>spaces of the  same
          sizes  and  order  as  the  <type>spacearray of patches
          created by <type>opPatch3, return a <type>spacearray of
          <type>spaces  of  weights  to  match  each  patch.   By
          default, all weights are one.  This can be  overwritten
          by derived classes to change the weighting function.

PRIVATE OPERATORS
     <type>space MakeWallWt(const <type>spacearray &) const
     <type>spacearray makeFWWindWt() const

SEE ALSO
```

`<type>operator`, `<type>opPatch3`, `<type>opOnepatch3`, `<type>opUnpatch`, `<type>opPatch`, `<type>opOnepatch`

Man page: sepaxis

```
SepAxis(UI)          Cube Reference Manual          SepAxis(UI)

NAME
     SepAxis - Class describing axis of the data set.

SYNOPSIS
     #include <Cube/sepaxis.h>

DESCRIPTION
     The class SepAxis contains the information about the  dimen-
     sion  of the axis (len), the seismic coordinates of the data
     (origin, increment between sample delta), and  the  descrip-
     tion  of the axis (label).  The class SepAxis reads the axis
     definition from the header file.

PUBLIC OPERATIONS
     SepAxis(SepInput* dataset,int axisnum)
          Create a SepAxis number axisnum from the SepInput.

     SepAxis(float o,float d,int n,char *label,int axisnum)
          Create a SepAxis from a complete input description.

     SepAxis * Copy()
          Create a copy of the axis object.

     SepAxis * Pad(int padstart, int padend)
          Create and return a padded copy of itself.  The  origin
          and  the  number  of  element are computed from the two
          given indeces. The title is copied.

     int Len()
     float Origin()
     float Delta()
     float End()
     char * Label()
          Return axis parameters.

PROTECTED VARIABLES
     int len:
          axis number of samples.

     float origin:
          axis world origin.

     float delta:
          axis world sampling interval.

     char* label:
          axis label.

     int axisnum:
          Axis number.

SepAxis(UI)          Cube Reference Manual          SepAxis(UI)

SEE ALSO
     SepInfo, SepInput, intro
```

Man page: sepcube

```
SepCube(UI)          Cube Reference Manual          SepCube(UI)

NAME
     SepCube - Cube data object

SYNOPSIS
     #include <Cube/sepcube.h>

DESCRIPTION
     A SepCube is a SepData object.

PUBLIC OPERATIONS
     SepCube()
          Create an empty SepCube with no data, empty axes.

     SepCube(SepData *)
          Create a sepplane by copying an input SepData.

     SepCube(SepInput *)
          Create a SepCube from an SepInput object. The data  are
          read and put into memory.

     SepCube(SepInfo *,void *data,int esize)
          Create a SepCube from a dimension  description  object,
          the data and the element size. The data are not copied.
          The SepInfo object is copied.

     SepCube(SepCube *)
          Construct a SepCube from a SepCube. Copy the data  from
          the cube and define the axes.

     SepCube * Copy()
          Copy the SepCube (data, axis, ...).

     SepCube * Byte(int zp,int nv,int pp,int np)
          Create a new SepCube by transforming the data  of  this
          into  bytes.  zp is the zero pixel value needed for the
          byte conversion.  nv is the non value pixel.  pp is the
          positive  clip  pixel  value.   np is the negative clip
          pixel value.

     SepCube * Window(int f,int j,int n,int f2,int j2,int n2,
                      int f3,int j3,int n3)
          Create a new SepCube by windowing this.

SEE ALSO
     SepInput, SepData, intro
```

Man page: sepdata

```
SepData(UI)         Cube Reference Manual         SepData(UI)

NAME
     SepData - Base class for the sep data objects

SYNOPSIS
     #include <Cube/sepdata.h>

DESCRIPTION
     SepData is the base class for all SEP data objects. All data
     objects  have a description of their structures using a Sep-
     Info data structure. A SepData object allows operation  such
     as  Window, Merge, Pad, Copy. Each of these operations modi-
     fies the description of the data  structure.   These  opera-
     tions are done in place, the data are overwritten by the new
     ones.

PUBLIC OPERATIONS
     SepData()
     SepData(SepData *)
     SepData(SepInfo *, void *data, int esize)
     SepData(SepInput*)
          Create a SepData object from scratch, from  an  another
          SepData  object,  from  a SepInfo object describing the
          data, or from a SepInput object.

     virtual ~SepData()
          Delete the data, the byte converter,  the  info  struc-
          ture.

     void Pad(int padStart[], int padEnd[])
          Pad this object to a new size. Two arrays corresponding
          of  the  padding  in  each  dimension  must be given in
          input. The data may be padded at the  beginning  or  at
          the end. The data will be padded with zeros. A new data
          array is created with the old one inside.

     void Merge(SepData *)
          Merge a data object with the current object.  The  data
          are  overwritten  in  the  following  way: the data are
          copied from the old ones and the input data are  merged
          into the new array. The dimension description object is
          modified.  If  the  object  was  unitialized,  i.e.  no
          dimension  information  have  been provided, the object
          copies the dimension description, the  data,  from  the
          input  object.   If  the  input object is not contained
          into this and the extension is  possible  the  data  is
          appended  to  the SepData object.  If the input is con-
          tained into this, the data in  the  input  object  have
          priority over this data.
          All these structure dimension are made on  the  coordi-
          nates  of  the input data SepInfo object. To add traces
          to plane the origin of the trace must reflect its posi-
          tion into the plane.

SepData(UI)         Cube Reference Manual         SepData(UI)

     void Window(int first[],int jump[],int num[])
          The object is windowed following the  input  arguments.
          Three  arrays  precising  the  windowing parameters for
          each dimension must be precised.  The data are overrit-
          ten  by the windowed data. The dimension description is
          deleted and  reconstructed  with  the  new  parameters.
          This  function  call  the  protected function doWindow.
          This function is redefined in the subclass  to  propose
          better parameter specification.

     SepData * Copy()
```

Return an exact copy of the current object. Copy call the virtual function doCopy.

SepData * Byte(int zp,int np,int pcp,int ncp)
Return a byte converted object of the current object. The arguments describe the pixel values color associated respectively to the zero data value, the non data value, the positive clip, the negative clip.

ByteConverter * GetByteConverter()
Return the byte converter used to to the byte conversion. The byte converter holds information such as clip values, gpow value, ...

SepInfo * Info()
Return the dimension description of the data.

void * Data()
Return the data associated to this object.

SepAxis * Axis(int n)
Return axis information of the axis number n.

char * Title()
Return the title

int Esize()
Return element size of the data.

PROTECTED OPERATIONS

SepData(DataStore*)
SepData(SepInfo *, DataStore *d, int esz)
Construct a SepData from a DataStore. This constructor is used into the derived class to construct directly either a MemoryDataStore or a DiskDataStore.

virtual SepData * doCreate(SepInfo *,DataStore *data,int esize)
Create a new object from a dimension description object, the array of data, and the element size. This function must be implemented in all the subclasses of the SepData.

virtual SepData * doCopy()
Create a new object from a copy of the other objects. This function must be implemented in all the derived class of the SepData.

virtual SepData * badObj(int stat)
Construct a null SepData from a status code. This virtual function should return an instance of the derived class in classes derived from this one.

virtual void initClass(SepInfo *)
Initialize the class specific private variables. This virtual function when used in conjunction with setPrivate should initialize all private variables.

void setPrivate(SepInfo *,DataStore *d,int es)
Initialize the general private variables. This function when used in conjunction with initClass should initialize all private variables.

boolean extendOK(SepInfo *)
Return true if the extension to the new sepInfo description is OK. This function is called when trying to modify an object of the current class. This function checks the dimensionality regarding its extensibility and its constants axes. Origin and length of the constant axis must be equal. The class constructor should have set the constant axis flags to the appropriate values, e.g. a plane of constant 3-axis value will have

the third constant Axis value set true so that any objects constructed will also be planes with a constant 3-axis value. If the new object is going to have greater dimensionality than this, this object check that the extendDimension flag is true.

void setSubData(SepData *)
Insert the input smaller object into the current object. The input data overwrite the data into the current object.

void doWindow(int first[], int jump[], int length[])
Window the data using the input window description. first gives the first element to be copied. jump is the increment (in samples) between input elements on each dimension. length is the number of element to copy for each dimension. This function overwrites its data and its dimension description by the windowed Data.

void doPad(int padstart[],int padend[])
Pad this object to a new size. The input array precised in samples how many sample must be added to the beginning or to the end of the data set for each dimension.

The data will be padded with zeros. A new data array is created with the old one inside. The data is overwritten by the new padded data. The SepInfo object is deleted and reconstructed with the new dimensions.

void doMerge(SepData *)
Merge a data object with this one to form a new data object. The data in the second object has priority over the data in this one.

SepData * doByte(int zp, int np, int pcp, int ncp)
Perform the byte conversion.

SepData * doExtract(SepInfo*)
Extract a new dataset from the current one from the input dimension description.

SEE ALSO
ByteConverter.

Man page: sepinfo

```
SepInfo(UI)          Cube Reference Manual          SepInfo(UI)

NAME
     SepInfo - Class describing data components.

SYNOPSIS
     #include <Cube/sepinfo.h>

DESCRIPTION
     The class SepInfo contains information about the  axes,  the
     number  of  dimension of the data set, the total size of the
     data, the element size, the title.

PUBLIC OPERATIONS
     SepInfo()
          Create a three-dimensional information  structure  with
          empty axes, a default element size of 4.

     SepInfo(int ndim)
          Create an information structure of ndim dimensions  all
          the axes being initialized to no elements.

     SepInfo(SepAxis *[], int ndim, int esz, char *title=nil)
          Create an information  structure  of  ndim  dimensions,
          with  the  the  input  axes, and with the input element
          size.

     SepInfo(SepInput*)
          A SepInfo object is created from a SepInput object. The
          SepInfo  object  get  the  element size, the title, the
          axes.  At least three axes are  initialized  eventually
          to empty.

     ~SepInfo()
          Delete the axes.

     SepInfo * Copy()
          Create a copy of the current SepInfo object.  The  axes
          are copied. The title is copied.

     SepInfo * Pad(int padstart[], int padend[])
          Create a new SepInfo objects with new padded axes. Copy
          the element size and the title.

     SepAxis* Axis(int i=,2,3)
          Get the axis information on the axis number i.

     int  Dimension()
          Return the number of dimension of the data.

     int  TotalSize()
          Return the total size of the data set.

     int  Esize()

SepInfo(UI)          Cube Reference Manual          SepInfo(UI)

          Return the element size .

     char* Title()
          Return a copy of the title of the data set.

PROTECTED OPERATIONS
     void setEsize(int)
          Set the element size to the input value. Recompute  the
          total data set size.

     void SetAxis(SepAxis *,int n)
          Set the axis number n to the input axis.

     boolean contains(SepInfo *)
```

Return true if the input infoObject is a subset of this. Check that the new object does not have too many dimensions or if it does check that they are all of length . Check that each axis is within the bounds of the corresponding axis for this object, i.e. calculate starting and ending indices of the input SepInfo and check whether they are inside current indices.

boolean compatible(SepInfo *)

Check whether or not the input info object is compatible with the current object. i.e. if they have compatible esize, compatible sampling interval, compatible starting values on their common axes.

SepInfo * unionInfo(SepInfo *)

Figure out the size of an info object that is a union of this and the new info object. The two info objects must be compatible. Compute starting indices and number of element of the new axes. The new object copy the esize and title of this.

unsigned long offset(int [])
unsigned long offset(float [])

Return the offset in bytes from the start of the object described by this info object, given a list of indices or a list of coordinates.

float * Start()

Build an array of coordinates for the start of the data.

boolean Step(float[], int)

Step these floating point values one sample along the specified axis. If we get to the end of the axis, step the next axis.

PROTECTED VARIABLES

SepAxis** axislist:

Array of SepAxis.

int dimension:

number of dimension of the data set.

int totalsize:

total size of the data set.

int esize:

element size.

char * title:

Title of the data set.

SEE ALSO

SepInput, SepAxis, intro

Man page: sepinput

```
SepInput(UI)        Cube Reference Manual        SepInput(UI)

NAME
     SepInput - Base class for reading input datas

SYNOPSIS
     #include <Cube/sepinput.h>

DESCRIPTION
     SepInput is responsible for reading data  descriptions  from
     the  header  file  specified  in  input.  This class is also
     responsible for getting the data from disk and puting it  on
     a  DataStore object.  A SepInput is described with a SepInfo
     object which contains the description of n  SepAxis  objects
     and  of the element size of the data.  No data are stored in
     memory inside this class.

PUBLIC OPERATIONS
     SepInput( char* name )
          Create a SepInput object from an header  file.  If  the
          name  is  equal to stdin the standard input is taken as
          the header file.  The input data file is opened. If  no
          data file or no header file is found an error status is
          set, and must be tested with the  function  Valid()  or
          Status().

     ~SepInput()
          The destructor closes the input data file.

     boolean Valid()
     int Status()
     char* StatusMessage()
     void ResetStatus()
          Valid() returns whether or not the status is  OK.   The
          function Status() returns the status.  There are 6 pos-
          sible values for the status:  OK,  NOHEADER,  NOSTREAM,
          BADAXIS,  BADINDEX,  OUTOFMEM  .  The  function SepSta-
          tusMessage( int ) prints on the standard error file  an
          error   message  corresponding  to  the  current  error
          status.(this  function   is   defined   in   sepdefs.h)
          ResetStatus() allows to set the status back to OK.

     SepAxis* Axis(int n=,2,3)
          Return the number n SepAxis object.

     SepInfo* Info()
          Return the SepInfo object.

     boolean Hetch(char*, int& )
     boolean Hetch(char*, float& )
     boolean Hetch(char*, char* )
          Get parameters from the  input  header  file.  true  is
          returned  if  the  variable was defined into the header
          file. otherwise false is returned.

SepInput(UI)        Cube Reference Manual        SepInput(UI)

     boolean Getch(char*, int& )
     boolean Getch(char*, float& )
     boolean Getch(char*, char* )
          Get parameters from the command line . true is returned
          if the variable was defined on the command line. other-
          wise false is returned.

     boolean Fetch(char*, int& )
     boolean Fetch(char*, float& )
     boolean Fetch(char*, char* )
          Get parameters from the  command  line  or  the  header
          file. true is returned if the variable was defined oth-
```

erwise false is returned.

void Sample(DataStore *,int i, int i2, int i3)
Get a sample pointed by the three input indices. A read on disk is performed. If the indices were invalid an error status is set.

void Trace(DataStore *,int ax, int i, int ax2, int i2)
Get a trace perpendicular to the plane given by the axis ax and ax2, and at the position i, i2 into this plane.

void Plane(DataStore *,int ax, int i)
Get a plane perpendicular to the axis ax at the position i on this axis.

void Cube(DataStore *)
Get the cube from the disk into the datastore.

PROTECTED OPERATIONS

```
void getSample(DataStore* d, int len, int n, int n2,
               int i, int i2, int i3)

void getTrace(DataStore* d, int len, int n, int n2, int n3,
              int ax, int i, int ax2, int i2)

void getPlane(DataStore* d, int len, int n, int n2, int n3,
              int ax, int i)
```

PROTECTED VARIABLES

char* headername:
The input header is in a file of this name. If the input was on stdin and the input was from a pipe a temporary file is created and the header copied to it.

boolean fromStdin, isaPipe:
Return true if the header file is the standard input and if it is a pipe.

FILE* instream:

File descriptor of the data file.

SepInfo* info:
SepInfo object.

int status:
error status.

SEE ALSO
SepInfo, intro

Man page: sepoutput

```
SepOutput(UI)       Cube Reference Manual       SepOutput(UI)

NAME
     SepOutput - Base class for writing seplib data files.

SYNOPSIS
     #include <Cube/sepoutput.h>

DESCRIPTION
     SepOutput is responsible for writting data  descriptions  to
     the header file specified in the constructor.  This class is
     also responsible for puting the data on disk.  No  data  are
     stored in memory inside this class.

PUBLIC OPERATIONS
     SepOutput( char* name, SepData * )
          Create a SepOutput object from a SepData object and the
          name of the output header file. If the name is equal to
          stdout the standard output is taken as the header file.
          The  output  header file is opened. If the SepData con-
          tains a pointer to a SepInput object  then  the  header
          file  from  the SepInput is copied to the output header
          file. If an error occurs the status variable is set and
          must be tested with the function Status().

     ~SepOutput()
          If the object has not already been closed the  destruc-
          tor  closes  the  header file and then writes a copy of
          the data in SepData object to disk.  The  only  way  in
          which the object may have been already closed is if the
          SepData object was deleted. (  see  SepData::~SepData()
          ).

     boolean Putch(char*, int )
     boolean Putch(char*, float )
     boolean Putch(char*, char* )
          Put parameters into the output  header  file.  true  is
          returned  if  everything  is  Ok.  Otherwise  false  is
          returned.

     void PutLine(char* )
          Write a line of text to the output header,  this  func-
          tion does NOT append a newline to the character string.

     int Status()
          The status of the object, zero = OK. CHECK IT!.

PROTECTED FUNCTIONS
     void allParOut()
          Write all the default  parameters  (n,n2,...)  on  the
          header file.

     void doClose(boolean copy)
          Close up the dataset and write  the  data  to  disk  if

SepOutput(UI)       Cube Reference Manual       SepOutput(UI)

          required.  If the argument is true the data will always
          be copied. If the argument is false  and  the  data  is
          already  on disk it will not be copied. This routine is
          only called with the  argument  false  when  a  SepData
          object  is  deleted,  this  is the only time when it is
          safe to assume a copy is not necessary.  The  SepOutput
          destructor calls doClose(true).

     char * makeName()
          Make up a name for the output data file.

     void blipPipe()
```

Do the standard seplib things if the output is a pipe.

PROTECTED VARIABLES

char* hdrname:
The output header name .

boolean toStdout:
true if the header file is the standard output.

boolean isaPipe:
true if the header file is going down a pipe.

boolean pipeData:
true if the data is going down the pipe.

FILE* headstream:
File descriptor of the header file.

FILE* outstream:
File descriptor of the data file.

SepData* sepData:
SepData object that will be written to output on close.

int status:
error status.

DERIVED CLASSES

NAME

SepPipeOutput - piped output to a command.

SYNOPSIS

```
#include <Cube/seppipeout.h>
```

DESCRIPTION

SepPipeOutput is a SepOutput object that pipes data to another command instead of writing it to a data file. The command is specified in the constructor. A pipe to the command is created with the unix "popen" routine.

PUBLIC OPERATIONS

SepPipeOutput(char* command, SepData *)
Create a SepPipeOutput object from a SepData object and the command to pipe the data to. If the SepData contains a pointer to a SepInput object then the header file from the SepInput is copied to the pipe.

~SepPipeOutput()
If the object has not already been closed the destructor copies the data to a file and closes the pipe. It then returns without waiting for the command on the other end of the pipe to terminate.

SEE ALSO

SepData, intro.

Man page: sepplane

```
SepPlane(UI)        Cube Reference Manual        SepPlane(UI)

NAME
     SepPlane - Plane data object

SYNOPSIS
     #include <Cube/sepplane.h>

DESCRIPTION
     A SepPlane is two-dimensional SepData object.

PUBLIC OPERATIONS
     SepPlane()
          Create an empty sepPlane with no data, empty axes.

     SepPlane(SepData *)
          Create a seplane by copying an input SepData.

     SepPlane(SepInput *,int ax,int islice)
     SepPlane(SepCube *,int ax,int islice)
          Create a SepPlane from a SepInput object or  a  SepCube
          perpendicular  to  the  axis  ax at the index islice on
          this axis.  The data are read from the  input  file  or
          copy from the cube. The information dimensions are read
          from the input file or deduced from the cube specifica-
          tion.

     SepPlane(SepInfo *,void *data,int esize)
          Create a SepPlane from a dimension description  object,
          the data and the element size. The data are not copied.
          The SepInfo object is copied.

     SepPlane * Copy()
          Copy the SepPlane (data,  axis,  ...).  This  functions
          call the virtual function doCopy.

     SepPlane * Byte(int zp,int nv,int pp,int np)
          Create a new SepPlane by transforming the data of  this
          into  bytes.  zp is the zero pixel value needed for the
          byte conversion.  nv is the non-value pixel.  pp is the
          positive  clip  pixel  value.   np is the negative clip
          pixel value.

     SepPlane * Window(int f,int j,int n,int f2,int j2,int n2)
          Create a new SepPlane by windowing this. This  function
          call the basic function doWindow.

&d2,int &es)
     void* GetData(int &n,int &n2,float &o,float &o2,float &d,float
          Return the data and axes description.

     void UserCoord(int &i3,float &u3)
          Return the index of the plane and its user origin.

SepPlane(UI)        Cube Reference Manual        SepPlane(UI)

SEE ALSO
     SepInput, SepData, SepCube, intro
```

Man page: space

```
SPACE(3LO)          Linop Reference Manual          SPACE(3LO)

NAME
     <type>space - Class for a space with elements of <type>

SYNOPSIS
     #include <<type>space.h>

DESCRIPTION
     A <type>space is a multi-dimensional array of data (of  type
     <type>),  with  a  description, along each dimension, of the
     origin, length, and delta that the data  is  located  along.
     The  data  is stored in a <type>Array, and the axis descrip-
     tion in an Axislist. The <type>space can be constructed from
     either  a  SepData *, or from a <type>Array and an Axislist.
     It can be manipulated with arithmetic operators, either with
     a  matching  <type>space  or  a  <type>,  and  inserted from
     another <type>space, or extracted from a <type>space.

     Linear operators, <type>op, are meant to do most of the cal-
     culations  on  a  <type>space. They are defined as a friend,
     and have direct access to the matrix class, and thus can use
     all the operators for the matrix class to do calculations.

PUBLIC OPERATIONS
     Axislist getaxislist() const
          Get a copy of the Axislist of the <type>space. This  is
          just a copy, and changing it will not change the inter-
          nal axislist.

     const <type>Array & getarray() const
          Return a constant reference to the array of the  space.
          The data cannot be changed, only viewed.  Use getdata()
          for <type>operators to change the data.

     <type>space()
     <type>space(const <type>space &)

     <type>space(SepInput *)
          Construct  <type>space  from  a  SepInput  *.  The
          <type>space and SepInput then share the data memory, so
          the SepInput should not be used or deleted after this.

     <type>space(SepData *)
          Construct <type>space from a SepData *. The <type>space
          and  SepData then share the data memory, so the SepData
          should not be used or deleted after this.

     SepData * sepdata()
          Return a pointer  to  a  new  SepData.  As  above,  the
          <type>space  and  SepData  share  the  data,  so  the
          <type>space should not be used after this.

     <type>space(SepPlane *)

SPACE(3LO)          Linop Reference Manual          SPACE(3LO)

     SepPlane * sepplane()
          Same as above, except for SepPlanes instead  of  SepDa-
          tas.

     <type>space(const <type>Array &,SepInfo *)
          Construct a <type>space from the  <type>Array  and  the
          SepInfo.  This checks to see if the <type>Array matches
          the SepInfo, exiting with an error if it doesn't.

     <type>space(const <type>Array &,const Axislist &)
          Same thing, but with an Axislist instead of a SepInfo.
```

<type>space(const Axislist &)
Construct a <type>space as defined by Axislist, initialized to zero.

<type>space empty() const
Return a <type>space the of the same size, but equal to zero.

~<type>space()

friend ostream & operator<<(ostream&,const <type>space &)
Print a <type>space. Actually just prints the matrix, not the axis information.

float norm() const
float norm2() const
norm returns the square root of the sum of the elements squared. norm2 returns norm squared.

<type> cross(const <type>space & s) const
If s is conformable (matching Axislist), cross multiply the two <type>space objects, element by element, and sums.

<type>space trans() const
Return a <type>space that is the transpose, meaning the first two axes are switched.

<type>space trans(int i,int j=,int k=2,int l=3) const
Transpose by giving axis the axis number i, axis the axis number j, etc.

<type> & operator()(int i,int j=,int k=,int l=) const
Get an element out of the matrix. This element can be changed.

<type>space extract(const Axislist & a) const
Get a <type>space which is the subspace of this one defined by a. This <type>space is a copy of the extracted parts, and altering it will not change the

original <type>space.

<type>space extractslice(int dim,int num) const
Extract a subspace, with one less dimension than the original, that is slice number num along dimension dim of the <type>space.

void insert(const <type>space & s)
Insert s into the <type>space if s is a subspace.

void insertslice(<type>space s,int dim,int num)
Insert s into the <type>space, making it slice num of dimension dim.

friend <type>space operator*(<type>,const <type>space &)
friend <type>space operator*(const <type>space &,<type>)
friend <type>space operator*(const <type>space &,
const <type>space &)
friend <type>space operator/(<type>,const <type>space &)
friend <type>space operator/(const <type>space &,<type>)
friend <type>space operator/(const <type>space &,
const <type>space &)
friend <type>space operator+(<type>,const <type>space &)
friend <type>space operator+(const <type>space &,<type>)
friend <type>space operator+(const <type>space &,
const <type>space &)
friend <type>space operator-(<type>,const <type>space &)
friend <type>space operator-(const <type>space &,<type>)

```
     friend <type>space operator-(const <type>space &,
                                   const <type>space &)
     <type>space operator-()
     <type>space operator+()
          All  the  arithmetic  operators  that  are  given   two
          <type>spaces require that the <type>spaces are conform-
          able; that is, their axislists are  equal  or  that  at
          least  one of the <type>spaces is a zero space (meaning
          its Axislist is made up of all wild Axis objects).

     <type>space & operator=(const <type>space &)
     <type>space & operator=(<type>)
     <type>space & operator+=(const <type>space &)
     <type>space & operator+=(<type>)
     <type>space & operator-=(const <type>space &)
     <type>space & operator-=(<type>)
     <type>space & operator*=(const <type>space &)
     <type>space & operator*=(<type>)
     <type>space & operator/=(const <type>space &)
     <type>space & operator/=(<type>)

PRIVATE VARIABLES
     <type>Array array
     Axislist axislist
```



```
PRIVATE OPERATIONS
     int compatible(const <type>space &) const
     void checkspace() const
     Index getindex(const Axis&, const Axis&) const

SEE ALSO
     Axis, Axislist, <type>spacearray, <type>op, <type>Array
```

Man page: spacearray

```
SPACEARRAY(3LO)      Linop Reference Manual       SPACEARRAY(3LO)

NAME
     <type>spacearray - A class that is an array of <type>spaces.

SYNOPSIS
     #include <<type>spacea.h>

DESCRIPTION
     A   <type>spacearray   is   a   two-dimensional   array   of
     <type>spaces.  A <type>spacearray is constructed by specify-
     ing the size, and then assigning <type>spaces into  it.  The
     arithmetic  operators  apply to <type>spacearrays, both with
     <type>s and  other  <type>spacearrays.  <type>operators  are
     meant   as   the   main   method   for   calculating   using
     <type>spacearrays- it is a friend to the class.

PUBLIC OPERATIONS
     <type>spacearray(int i=,int j=)
          Construct a <type>spacearray with i rows and j columns.

     <type>spacearray(const <type>space &)
          Construct  a  x   <type>spacearray   containing   the
          <type>space.

     <type>spacearray empty() const
          Return a <type>spacearray with the same number of  rows
          and  columns,  the  same  axislists,  but  with all the
          spaces equal to zero.

     <type>spacearray(const <type>spacearray &)
     ~<type>spacearray()

     float norm() const
     float norm2() const
          norm returns the square root of the sum of the  squares
          of   the  elements  in  all  the  <type>spaces  in  the
          <type>spacearray. norm2 returns the norm squared.

     <type> cross(const <type>spacearray &) const
          If the two  <type>spacearrays  are  conformable,  cross
          multiply the two, element by element, and sums. For two
          <type>spacearrays to be conformable, they must have the
          same number of rows and columns, and each corresponding
          pair of <type>spaces must be conformable.

     <type>spacearray trans() const
          Returns a  new  <type>spacearray  that  is  created  by
          switching the rows and columns of the <type>spacearray.
          All of the <type>spaces are copied.

     friend <type>spacearray operator+
                         (<type>,const <type>spacearray &)
     friend <type>spacearray operator+

SPACEARRAY(3LO)      Linop Reference Manual       SPACEARRAY(3LO)

                         (const <type>spacearray &,<type>)
     friend <type>spacearray operator+
      (const <type>spacearray &,const <type>spacearray &)

     friend <type>spacearray operator-
                         (<type>,const <type>spacearray &)
     friend <type>spacearray operator-
                         (const <type>spacearray &,<type>)
     friend <type>spacearray operator-
      (const <type>spacearray &,const <type>spacearray &)

     friend <type>spacearray operator*
```

```
                    (<type>,const <type>spacearray &)
     friend <type>spacearray operator*
                    (const <type>spacearray &,<type>)
     friend <type>spacearray operator*
      (const <type>spacearray &,const <type>spacearray &)

     friend <type>spacearray operator/
                    (<type>,const <type>spacearray &)
     friend <type>spacearray operator/
                    (const <type>spacearray &,<type>)
     friend <type>spacearray operator/
      (const <type>spacearray &,const <type>spacearray &)

     <type>spacearray operator+()
     <type>spacearray operator-()
          The   arithmetic   operators,   applied   to   two
          <type>spacearrays, check to see if the two are conform-
          able, and if they are, performs the  operation  element
          by element.

     <type>spacearray & operator=(const <type>spacearray &)
     <type>spacearray & operator=(<type>)
     <type>spacearray & operator+=(const <type>spacearray &)
     <type>spacearray & operator+=(<type>)
     <type>spacearray & operator-=(const <type>spacearray &)
     <type>spacearray & operator-=(<type>)
     <type>spacearray & operator*=(const <type>spacearray &)
     <type>spacearray & operator*=(<type>)
     <type>spacearray & operator/=(const <type>spacearray &)
     <type>spacearray & operator/=(<type>)

     <type>space & operator()(int i=,int j=)
          Return a reference to the  <type>space  at  coordinates
          (i,j). This <type>space can be assigned to.

     friend ostream& operator<<(ostream&,const <type>spacearray &)

     int numrows()
     int numcols()
          Eeturn  the  number  of   row   or   columns   in   the
```



```
          <type>spacearray.

PRIVATE VARIABLES
     int rows
     int cols
     <type>space * spaces

PRIVATE OPERATIONS
     void compatible(const <type>spacearray &) const
     <type>space & getspace(int i,int j) const

SEE ALSO
     <type>space, <type>op, <type>opArray
```