Stanford Exploration Project, Report 75, November 17, 1997, pages 505–535

Stanford Exploration Project, Report 75, November 17, 1997, pages 505–535

Short Note

An implementation of operators and spaces in C++

Geoff Dunbar[1]

keywords: *arrays, interpolation*

INTRODUCTION

This paper is a companion to the previous paper (Nichols et al., 1992) which gives an overview of the problems that these C++ classes are designed to solve. It is a collection of manual pages describing my C++ work this summer, with a short extra explanation preceeding each page. It is intended as a reference document for someone who wants more details about the code that we have written. It is probably most useful to a reader who is already familiar with C++.

The manual pages are in an order such that every class or function is listed after anything that it depends on. Figure 1 is a schematic of the dependencies between different sections of the code. A line on the plot indicates that the lower class or function depends on the upper. A thick line indicates actual C++ derivation.

CLASSES USED IN DEFINING SPACES

The `space` class contains two different things, the matrix class and the axis descriptions, along with some operations. So, in building up to the `space` class, let us consider the two classes that describe the axes: `Axis` and `Axislist`. `Axis` is basically a collection of the four things that describe an axis: the origin, delta, length, and name. These are all public variables, which can be looked at and changed freely, for anyone who has access to the `Axis` object. In addition, there are a few member functions which provide some useful operations that are often used with `Axis` objects.

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

SYNOPSIS
     #include <Axis.h>
```

[1]**email:** not available

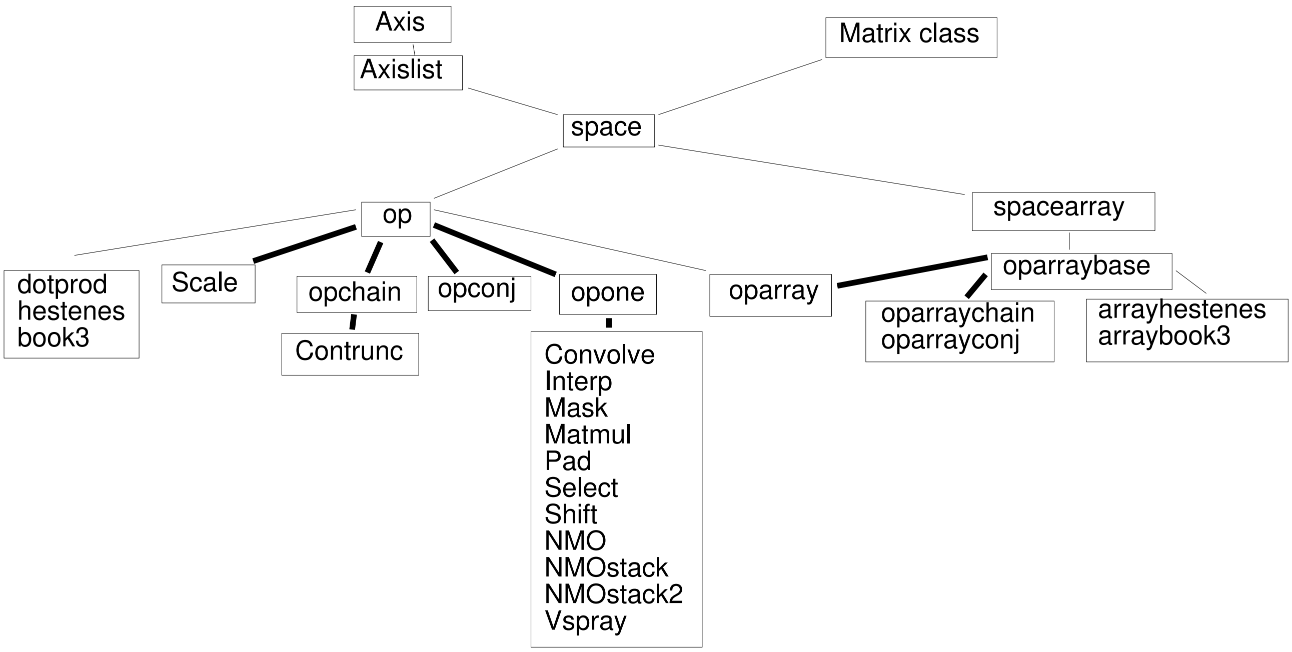

Figure 1: "Schematic of dependencies." dave2-ClassPic [NR]

```
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), and a label for this axis (name). An Axis can be
     constructed from either the above information, or from a SepAxis *.

PUBLIC VARIABLES
     float origin
     float delta
     int length
     char * name

PUBLIC OPERATIONS
     Axis(float origin,float delta,int length,char * name)
          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()
          Returns a pointer to a new SepAxis object, which matches the Axis.

     Axis()
     Axis(const Axis &)
     Axis& operator=(const Axis &)
```

```
    ~Axis()

    int operator==(const Axis &)
         Checks if two Axis have the same values for all their members.

    int subaxis(Axis & a)
         Checks if the Axis is a sub-axis of a. For Axis x to be a sub-axis  of
         Axis y, x must: (1) have the same label as y  (2) have every sample of
         x have the same position as some sample of y.

    float max()
         Returns origin + delta * length.

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

    float value(int)
         Returns the float position of the sample numbered by the in parameter.

SEE ALSO
    Axislist, <type>space, SepAxis
```

The `Axislist` class is also a collection of things: the array of `Axis` objects, and the number of dimensions. Again these are public, and can be freely viewed or changed by anyone who can access the `Axislist` object. Also, a few useful functions are provided.

```
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 dimen-
    sions 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)
     Returns a pointer to a new SepInfo matching the  Axislist,  given  the
     element size.

Axislist(const Axis & a)
     Construct an Axislist of dimension 1, with the a as the first Axis  in
     list.

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

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

Axislist transpose()
     Returns 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=1,int k=2,int l=3)
     Returns   an   Axislist   that   is   transposed   according  to   the   int

     parameters.  list[0] get the Axis number i, list[1] gets Axis[j], etc.
     The dimension is then set according to the greatest non-empty Axis.

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

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

int subspace(Axislist & a)
     Checks if this is a subspace of a. It is if  the  two  have  the  same
     dimension,  and if all the Axis in the list are sub-axes of the match-
     ing Axis in a.list.

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

The space class also contains a matrix class. This is (for now) M++ (Dya, 1991).

This brings us to the first major class: the `space` class. The word space is preceeded by <type> in the man pages; in practice an actual type is in that place. `floatspace` and `complexspace` are the two types currently supported.

The `space` class contains two things, the `Axislist` to describe the axes that the data lies on, and the matrix class containing that data. These are both private members, only accessable through the member functions.

There are two reasons for this,

1. A user of the class should only want to do things to a space that can be accomplished with the member functions.
2. The `Axislist` and matrix must match, and this could not be guaranteed if the variables could be accessed freely. (Not that it is guaranteed by them being private members, but it is a safer interface)

For the special case of operators (the class op), we can get around this restriction, see the `op` section for details.

The complete functionality of the `space` class is described in the man page below. I will discuss a few key points. The first is that the operations for a space, with a few exceptions, do not allow changing of the dimensions or axes of a space. Viewing the information about the axes, with the `getaxislist`() function, or changing particular values in the `space`, by assigning into the `space`, is legal, but the overall "shape" of the space cannot be changed. Having said this, there are a few ways to get a new `space`, with a different "shape" than the original, although the original `space` is not changed. These operations, which change the "shape", include the `extract`, and transpose (`trans`) operation.

All the arithmetic and assignment operators are supported for spaces, such as multiplication, addition, etc. These can be by a value of the type of the space, or by another space. An operation with another space, such as multiplying by another space, requires that the axis information of the two spaces is identical. This operation is then done element by element.

```
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  description  in  an Axislist. The <type>space can be constructed from
     either a SepData *, or from a <type>Array and an Axislist. It can be  mani-
     pulated  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 calculations 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()

Gets a copy of the Axislist of the <type>space. This is just a copy, and changing it will not change the internal axislist.

<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 *)
SepPlane * sepplane()

Same as above, except for SepPlanes instead of SepDatas.

<type>space(<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(<type>Array &,Axislist &)

Same thing, but with an Axislist instead of a SepInfo.

<type>space(Axislist &)

Constructs a <type>space as defined by Axislist, initialized to 0.

<type>space empty()

Returns a <type>space the of the same size, but equal to 0.

~<type>space()

friend ostream & operator<<(ostream &,<type>space &)

To print a <type>space. Actually just prints the matrix, not the axis information.

float norm()
float norm2()

norm return the square root of the sum of the elements squared. norm2

```
     returns norm squared.

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

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

<type>space trans(int i,int j=1,int k=2,int l=3)
     Transposed by giving axis 0 the axis number i, axis 1 the axis  number
     j, etc.

<type> & operator()(int i,int j=0,int k=0,int
     Gets an element out of the matrix. This element can be changed.

<type>space extract(Axislist & a)
     Gets a <type>space which is the sub-space 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)
     Extracts a sub-space, with one less dimension than the original,  that
     is slice number num along dimension dim of the <type>space.

void insert(<type>space & s)
     Inserts s into the <type>space if s is a sub-space.

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

friend <type>space operator*(<type>,<type>space &)
friend <type>space operator*(<type>space &,<type>)

friend <type>space operator*(<type>space &,<type>space &)
friend <type>space operator/(<type>,<type>space &)
friend <type>space operator/(<type>space &,<type>)
friend <type>space operator/(<type>space &,<type>space &)
friend <type>space operator+(<type>,<type>space &)
friend <type>space operator+(<type>space &,<type>)
friend <type>space operator+(<type>space &,<type>space &)
friend <type>space operator-(<type>,<type>space &)
friend <type>space operator-(<type>space &,<type>)
friend <type>space operator-(<type>space &,<type>space &)
<type>space operator-()
<type>space operator+()
     All the arithmetic operators that are given two  <type>spaces  require
     that  the  <type>spaces  are conformable- that is, their axislists are
     equal.

<type>space & operator=(<type>space &)
<type>space & operator=(<type>)
<type>space & operator+=(<type>space &)
```

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

PRIVATE VARIABLES
     <type>Array array
     Axislist axislist

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

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

The way to do more complex manipulations of spaces is through operators (class `op`). Following is the base class, from which all ops are derived. The op has two operations: the `Forward` and `Conjugate` functions, both of which take a space and return a space. These also exist for spacearrays, but these are just an extension of the space versions. More on spacearrays later.

CLASSES USED IN DEFINING OPERATORS

The power of `ops` in dealing with spaces comes through their special access to the internal members of space, the matrix and `Axislist`. `op` is defined as a friend to the class `space`, so it can use the internal variables of `space`. Any class derived from `op` can use the functions `getdata` and `getinfo` to view or change these variables. A special bonus that a class derived from `op` gets is that it can use the matrix functions directly. The matrix class has many useful functions that make the manipulation of the matrices easier, and should make writing a powerful `op` fairly painless.

Something to notice with `ops` is that, although they are objects, they cannot be copied and assigned like most objects. This is because, in most cases, operators are accessed as `ops`, but are in fact some class derived from `op`. Thus, when `ops` are passed around, or assigned, it must be done by reference, or with pointers. Also, an `op` cannot be a return value, even by reference, because the `op` will be erased when it goes out of scope. This may seem complicated, so here is a complete list of things you can do with an `op`:

1. Construct it
2. Apply it to a space

3. Pass it to a function by reference or as a pointer.

NAME

<type>op - The virtual base class for linear operators.

SYNOPSIS

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

DESCRIPTION

The class <type>op is a virtual base class defining the operations that must be defined for a linear operator. These include a constructor, a destructor (which does not always need to be written), a Forward function, and a Conjugate 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. Lastly, 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.

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(<type>space &) = 0
virtual <type>space Conjugate(<type>space &) = 0
```

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

```
<type>spacearray Forward(<type>spacearray &)
<type>spacearray Conjugate(<type>spacearray &)
```

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

PROTECTED OPERATIONS

```
int rows(<type>spacearray &)
int cols(<type>spacearray &)
```

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(<type>space &)
```

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(<type>space &)
```

getinfo allows access to the Axislist in a <type>space, which can be changed with this function.

SEE ALSO

```
     <type>space, <type>spacearray, <type>opone,  <type>opchain,  <type>optrans,

     <type>oparray
```

`Scale` is an operator derived directly from`op`; it multiplies a space by a constant.

```
NAME
     <type>Scale - Operator to scale a <type>space.

SYNOPSIS
     #include <<type>opscale.h>

DESCRIPTION
     <type>Scale is a <type>op which multiplies a whole <type>space by a <type>.
     The Conjugate function behaves the same as Forward.

PUBLIC OPERATIONS
     <type>Scale(<type> value)
          Create a <type>Scale which multiples by value.

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

PRIVATE VARIABLE
     <type> factor

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

Although a simple operator such as `Scale` is easy to write as a direct derivative of `op`, most operators are derived from `opone`. Class `opone` handles some of the work that most operators share, making it better to derive an operator from.`opone` automatically arranges the input space in the proper order for the central routine, setting up the output space, and cycling over any extra axes. Thus, deriving an operator from `opone` saves a lot of work; all that needs to be defined in the derived class are the constructor, and the central routine (and maybe a destructor).

```
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 destructor (if one is needed), and the dodata 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(<type>space &)
<type>space Conjugate(<type>space &)
These functions call the functions which rearrange the input space, create the output space, cycle over extraneous axes, and finally call the dodata routine. Thus, in writing an operator derived from <type>opone, everything is handled except for the dodata routine itself.

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

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 the are not in inaxis will be cycled over.

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 Conjugate function is called.

PROTECTED OPERATIONS

virtual void forwarddata(<type>Array & in,<type>Array &
The function that Forward calls to process the input <type>Array. It

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

virtual Axislist forwardinfo(Axislist &)
This function is called by Forward, given the input <type>spaces Axislist, and returns the Axislist of the output <type>space.

virtual void conjdata(<type>Array &,<type>Array &)
virtual Axislist conjinfo(Axislist &)
Same as above, except for Conjugate.

virtual void dodata(<type>Array & a,<type>Array &

The dodata function is meant to be overwritten by a derived class. When called by Forward, conj=0, a is the input <type>Array, and b is the output. When called by Conjugate, conj=1, a is the output, and b is the input.

PRIVATE OPERATIONS

<type>space forwardorder(<type>space &)
void forwardhelp(int,int,int,Axislist &,<type>Array &,<type>Array &)

type>space conjorder(<type>space &)
void conjhelp(int,int,int,Axislist &,<type>Array &,<type>Array &)

DERIVED CLASSES

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

1. Write a constructor the 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 dodata routine to do the work of the operator. For conj=0, the Forward case, dodata will take a <type>Array matching inaxis, and return a <type>Array matching outaxis. For conj=1, Conjugate, a <type>Array matching inaxis must be created from a <type>Array matching outaxis.

SEE ALSO

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

The next few man pages are operators derived from `opone`. First, a matrix multiply operator, `Matmul`, which multiplies by a space.

NAME

<type>Matmul - <type>opwhich performs a matrix multiply.

SYNOPSIS

#include <<type>opmul.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 multiplies the <type>space by the left-side, along the contracting Axis.

PUBLIC OPERATIONS

<type>Matmul(<type>space &)

```
          Constructs a <type>Matmul with the <type>space as the matrix to multi-
          ply by.

     <type>Matmul(Axis & in,Axis & out,<type>Array &)
          Constructs a <type>Matmul by specifying the  Axis  to  contract  (in),
          expand (out), and the <type>Array to multiply by.

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

PRIVATE VARIABLE
     <type>Array matrix

PRIVATE OPERATION
     void dodata(<type>Array &,<type>Array &,int)

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

This operator, Pad, pads or truncates one of the axes in a space.

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

SYNOPSIS
     #include <<type>oppad.h>

DESCRIPTION
     <type>Pad 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>Pad, the number of samples  to  pad  or
     truncate  at  the front and back are specified. The Conjugate function does
     the opposite- that is, pads where Forward truncates,  and  truncates  where
     Forward pads.

PUBLIC OPERATIONS
     <type>Pad(Axis &,int front,int back)
          Construct a <type>Pad 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>space Forward(<type>space &)
     <type>space Conjugate(<type>space &)

PRIVATE VARIABLES
     int front
     int back

PRIVATE OPERATION
     void dodata(<type>Array &,<type>Array &,int)
```

SEE ALSO

<type>space, <type>opone

Shift moves the data in a space along one of its axes.

NAME

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

SYNOPSIS

#include <<type>opshift.h>

DESCRIPTION

<type>Shift 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>Shift(Axis &,int shift)

Create a <type>Shift 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.

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

PRIVATE OPERATION

void dodata(<type>Array &,<type>Array &,int)

SEE ALSO

<type>space,

Convolve performs a 1 dimensional convolution on one of the axes of a space.

NAME

<type>Convolve - Operator to do 1-D convolution

SYNOPSIS

#include <<type>opconv.h>

DESCRIPTION

<type>Convolve is a <type>opone which convolves by a filter along one dimension. Constructing the operator involves specifying the Axis to convolve along, and the filter to convolve with. The output, along the Axis of convolution, will be longer, by the length of the filter - 1, than the input.

PUBLIC OPERATIONS

<type>Convolve(<type>Array &,Axis &)

Create a <type>Convolve giving a <type>Array filter and an Axis to

```
          convolve. The filter is copied into the <type>Convolve.

     <type>Convolve(<type>space &,Axis &)
          Same, except using a <type>space as the filter.

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

PRIVATE VARIABLE
     <type>Array filter

PRIVATE OPERATION
     void dodata(<type>Array &,<type>Array &,int)

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

`Interp` does a linear interpolation along one of the axes of a space. This changes the length of the data along this axis.

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

SYNOPSIS
     #include <<type>opinter.h>

DESCRIPTION
     <type>Interp 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 out-
     put 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  Conjugate  function.  To  construct  a
     <type>Interp, you simply specify the input Axis, and the output Axis.

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

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

PRIVATE OPERATION
     void dodata(<type>Array &,<type>Array &,int)

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

`Mask` is an operator which sets some of the values in a space, along one of the axes, equal to zero.

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

SYNOPSIS
     #include <<type>opmask.h>

DESCRIPTION
     <type>Mask 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  Con-
     jugate function does the same thing as the Forward function.

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

     ~<type>Mask()

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

PRIVATE VARIABLE
     int * mask

PRIVATE OPERATION
     void dodata(<type>Array &,<type>Array &,int)

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

`Select` creates a sub-space which is formed by taking some of the values from a space along one of the axes.

```
NAME
     <type>Select - Operator to select values from a <type>space.

SYNOPSIS
     #include <<type>opselect.h>

DESCRIPTION
     <type>Select  is  a  <type>opone  which  operates  along  one  axis  of   a
     <type>space.  It  selects some of the values along this axis to go into the
     output <type>space, leaving others out. In  creating  a  <type>Select,  the
     Axis  to  operate on is specified, as well as which positions to select the
     values from. The Conjugate function reverses this,  putting  zeros  in  the
     positions which are not selected by the Forward operator.

PUBLIC OPERATIONS
```

<type>Select(Axis &,int *)
Create a <type>Select, specifying the Axis to operate on and an int array indicating which positions are selected. The operator selects the positions where the int array is not equal to zero.

~<type>Select()

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

PRIVATE VARIABLE
int * mask

PRIVATE OPERATION
void dodata(<type>Array &,<type>Array &,int)

SEE ALSO
<type>space,

NMO performs normal moveout on a space.

NAME
NMO - Operator to perform normal moveout on a floatspace.

SYNOPSIS
#include <NMO.h>

DESCRIPTION
NMO is a floatopone which performs normal moveout on a floatspace. Constructing the NMO requires specification of the time axis, offset axis, and a slow parameter or array (slowness is 1/velocity).

PUBLIC OPERATIONS
NMO(Axis & time,Axis & offset,float slow)
NMO(Axis & time,Axis & offset,float slow[])
Create an /fBNMO/fP with slow as an array- must be the same length as the time axis. The slow array is copied into the operator.

~NMO()

floatspace Forward(floatspace &)
floatspace Conjugate(floatspace &)

PRIVATE VARIABLE
float * slow

PRIVATE OPERATION
void dodata(floatArray &,floatArray &,int)

SEE ALSO
<type>space, <type>opone, NMOstack, NMOstack2

`NMOstack` performs normal moveout and stack on a space.

```
NAME
     NMOstack - Operator to perform normal moveout and stack on a floatspace.

SYNOPSIS
     #include <NMOstack.h>

DESCRIPTION
     NMOstack is a floatopone to perform normal moveout and stack  the  results.
     In  constructing  an NMOstack, the time Axis, offset Axis, and a slow value
     or array must be specified. The offset axis is  contracted  along,  so  the
     output space has one less dimension than the input space.

PUBLIC OPERATIONS
     NMOstack(Axis & time,Axis & offset,float slow)
     NMOstack(Axis & time,Axis & offset,float slow[])
          This constructor takes the slow parameter as an array, which  must  be
          the same length as the time Axis. This array is copied into the opera-
          tor.

     ~NMOstack()

     floatspace Forward(floatspace &)
     floatspace Conjugate(floatspace &)

PRIVATE VARIABLE
     float * slow

PRIVATE OPERATION
     void dodata(floatArray &,floatArray &,int)

SEE ALSO
     <type>space, <type>opone, NMO, NMOstack2
```

`NMOstack2` does the same as NMOstack, but outputting a 2 dimensional space, rather than eliminating one of the axes.

```
NAME
     NMOstack2 - Operator to perform normal moveout and stack on a floatspace.

SYNOPSIS
     #include <NMOstack2.h>

DESCRIPTION
     NMOstack2 is a floatopone to perform normal moveout and stack the  results.
     In  constructing an NMOstack2, the time Axis, offset Axis, and a slow value
     or array must be specified. NMOstack2 reduces the length of the offset axis
     to 1 in the output space, so the output space has the same number of dimen-
     sions as the input space.
```

```
PUBLIC OPERATIONS
     NMOstack2(Axis & time,Axis & offset,float slow)
     NMOstack2(Axis & time,Axis & offset,float slow[])
          This constructor takes the slow parameter as an array, which  must  be
          the same length as the time Axis. This array is copied into the opera-
          tor.

     ~NMOstack2()

     floatspace Forward(floatspace &)
     floatspace Conjugate(floatspace &)

PRIVATE VARIABLE
     float * slow

PRIVATE OPERATION
     void dodata(floatArray &,floatArray &,int)

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

`Vspray` does calculates the velocity spectra of a space.

```
NAME
     Vspray - Operator to perform velocity spectra.

SYNOPSIS
     #include <Vspray.h>

DESCRIPTION
     Vspray is a floatopone which performs velocity  spectra.  Constructing  the
     Vspray  requires specification of the data Axislist- with a time and offset
     axis, and the velocity space Axislist- with a tau and  velocity  axis.  The
     time and tau axes must be the same.

PUBLIC OPERATIONS
     Vspray(Axislist & data,Axislist & vspace)
     floatspace Forward(floatspace &)
     floatspace Conjugate(floatspace &)

PRIVATE OPERATION
     void dodata(floatArray &,floatArray &,int)

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

Operators can be applied to spaces singly, as one would do with the above operators. In addition, classes exist to enable creating the conjugate operator, and to create operators which are the product of operators. The class `opconj` handles the conjugate of an operator.

```
NAME
     <type>opconj - Class to take the conjugate of a <type>op.

SYNOPSIS
     #include <<type>opt>

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

FRIEND OPERATION
     friend <type>opconj conj(<type>op &)

PUBLIC OPERATIONS
     <type>space Forward(<type>space &)
     <type>space Conjugate(<type>space &)

     <type>opconj(<type>op &)

PRIVATE VARIABLE
     <type>op * op

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

Class `opchain` stores the product of operators. This product can be done by the user, or set up by a class derived from opchain.

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

SYNOPSIS
     #include <<type>opc.h>

DESCRIPTION
     Class <type>opchain  is  class  derived  from  <type>op  which  allows  for
     automatic  handling  of  products  of  <type>ops. Multiplying two <type>ops
     yields a <type>opchain which, when the Forward function is called,  applies
     the second <type>op to the <type>space, followed by the first <type>op. The
     Conjugate function calls the Conjugate of the <type>ops, in reverse  order.
     One  thing  that  is important to notice is that the <type>opchain contains
```

pointer to <type>ops, not the <type>ops themselves. Thus, the <type>ops must be maintained (not changed or deleted), or the <type>opchain will not work.

PUBLIC OPERATIONS

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

This handles products of <type>ops. If the arguments are <type>ops, rather than <type>opchains, they will be converted to <type>opchains by the constructor below. Then, a <type>opchain containing the product of the <type>ops is returned.

```
<type>space Forward(<type>space &);
<type>space Conjugate(<type>space &);

<type>spacearray Forward(<type>spacearray & x)
<type>spacearray Conjugate(<type>spacearray & x)

<type>opchain();
<type>opchain(<type>op &);
<type>opchain(const <type>opchain &);
<type>opchain & operator=(const <type>opchain &);
~<type>opchain();
```

PRIVATE VARIABLES

```
<type>op **ops
int num
```

PRIVATE OPERATION

```
void append(<type>op &)
```

DERIVED CLASSES

It is possible to derive classes from <type>opchain, which represent specific products of <type>ops. Classes of this form will contain the operators of the chain, as private members. Then, the only code that needs to be written is a constructor, which first initializes the operators in the chain (by calling their constructors), and then calling the copy constructor on the product of these operators. This class will then act as the product of the operators within it.

SEE ALSO

<type>space, <type>spacearray, <type>op, <type>oparray,

`Contrunc` is a class derived from opchain which convolves along 1 dimension, and then truncates the data to a certain length. `Contrunc` contains a Convolve operator, a `Shift` operator, and a `Pad` operator, which it applies in order.

NAME

<type>Contrunc - Operator to convolve in 1-D and truncate.

SYNOPSIS

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

```
DESCRIPTION
     <type>Contrunc convolves the traces of a <type>space by a filter, truncates
     the  result,  and shifts the data so that it starts in the same position as
     the input. Constructing a  <type>Contrunc  requires  specification  of  the
     filter,  the Axis to apply it to, and the number of samples to truncate off
     the  front  and  back  of  the  result.  <type>Contrunc  is  derived   from
     <type>opchain, as it is a composite operator.

PUBLIC OPERATIONS
     <type>Contrunc(<type>Array &,Axis &,int front,int back)
          Construct the <type>Contrunc given a <type>Array to use as the filter,
          an Axis to apply it too, and the number of samples to truncate off the
          front and back. The output data will be shifted to have the same  ori-
          gin as the input data.

     <type>Contrunc(<type>space &,Axis &,int front,int back)
          Same thing, except use the <type>space as the filter.

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

PRIVATE VARIABLES
     <type>Convolve conv
     <type>Pad trunc
     <type>Shift shift

SEE ALSO
     <type>space,   <type>op,   <type>opchain,   <type>Convolve,   <type>Pad,
     <type>Shift
```

The man pages up to this point have described the framework for single operators and spaces; that is, without arrays of operators or spaces. So, we can examine some of the functions which use this framework. First, given an operator, there exists a test, the dot product test, which verifies that the `Forward` and `Conjugate` functions are really conjugate to each other.

```
NAME
     <type>dotprod - Function that performs the dot product test on a operator.

SYNOPSIS
     #include <<type>dotprod.h>

DESCRIPTION
     <type> <type>dotprod(Axislist & in,Axislist & out,<type>op &,long seed=0)

     This function tests a <type>op to make sure that its Forward and  Conjugate
     functions  really  are  conjugate to each other. The function takes in, the
     input Axislist of the <type>op, and  out,  the  output  Axislist,  and  the
     <type>op.  It returns the fractional difference in the two results given by
```

the test. This should be just round-off error in the <type>op, near machine precision. If it is fairly large, the operator is probably coded wrong.

<type>dotprod 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>op

Another thing we can do with this framework is write general conjugate gradient solvers. One of these is a Hestenes solver.

NAME
<type>hestenes - Function to solve conjugate gradient problem.

SYNOPSIS
#include <<type>hest.h>

DESCRIPTION
<type>space <type>hestenes(<type>space x0,<type>space & y,<type>op & A,int niter)

<type>hestenes runs the Hestenes conjugate gradient solver on the problem

min || A x - y ||^2

given x0 as an initial guess for x. The solver currently ignores the initial guess. The solver runs for niter iterations, then returns its solution.

SEE ALSO
<type>space, <type>op, <type>book3, <type>arrayhestenes

Here is another solver.

NAME
<type>book3 - Function to solve conjugate gradient problem.

SYNOPSIS
#include <<type>book3.h>

DESCRIPTION
<type>space <type>book3(<type>space x0,<type>space & y,<type>op & A,int niter)

<type>book3 runs a conjugate gradient solver on the problem

```
     min || A x - y ||^2

     given x0 as an initial guess for x. The solver runs for  niter  iterations,
     then returns its solution.

SEE ALSO
     <type>space, <type>op, <type>arraybook3, <type>hestenes
```

In addition to the basic framework of spaces and operators, arrays of the two are also supported by another set of classes. The class `spacearray` has very similar functionality to `space`, except now the class contains a 2 dimensional array of spaces.

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

SYNOPSIS
     #include <<type>spacea.h>

DESCRIPTION
     A  <type>spacearray  is  a  2  dimensional   array   of   <type>spaces.   A
     <type>spacearray  is constructed by specifying 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>ops and <type>oparrays
     are meant as the main method for calculating using <type>spacearrays-  both
     are friends to the class.

PUBLIC OPERATIONS
     <type>spacearray(int i=1,int j=1)
          Constructs a <type>spacearray with i rows and j columns.

     <type>spacearray(<type>space &)
          Constructs a 1x1 <type>spacearray containing the <type>space.

     <type>spacearray empty()
          Returns a <type>spacearray with the same number of rows  and  columns,
          the same axislists, but with all the spaces = 0.

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

     float norm()
     float norm2()
          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(<type>spacearray &)
          If the two <type>spacearrays are  conformable,  cross  multiplies  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()
          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>,<type>spacearray &)
     friend <type>spacearray operator+(<type>spacearray &,<type>)
     friend <type>spacearray operator+(<type>spacearray &,<type>spacearray &)

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

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

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

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

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

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

     <type>space & operator()(int i=0,int j=0)
          Returns a reference to the  <type>space  at  coordinates  (i,j).  This
          <type>space can be assigned to.

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

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

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

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

The class `oparraybase` provides a base class for operator arrays. This class is not meant for use, but is necessary so that the various types of oparrays can be treated as one type. The same rules that apply to operators apply to oparraybases and their derived classes.

```
NAME
     <type>oparraybase - Virtual base class for arrays of <type>ops.

SYNOPSIS
     #include <<type>opabase.h>
          It is doubtful you would ever include this file, as it is included  in
          <<type>opa.h>, which is the useful class.

DESCRIPTION
     This  class  provides  a  virtual  base  class   that   <type>oparray   and
     <type>oparrayconj  are  derived from. This is necessary for operator chain-
     ing, but the class is not useful for anything.

PUBLIC OPERATIONS
     virtual <type>spacearray Forward(<type>spacearray &) = 0
     virtual <type>spacearray Conjugate(<type>spacearray &) = 0

SEE ALSO
     <type>spacearray, <type>oparray, <type>oparrayconj
```

Class `oparray` is the actual class the contains an array of operators. When an `oparray` is applied to a `spacearray`, the product is handled like a matrix multipliy. That is, the rows of the `oparray` are applied to the columns of the `spacearray`, and these results are summed. However, instead of each product just being a multiplication, it is an `op` being applied to a `space`.

The way to create an `oparray` is to construct it, telling it how big it should be (how many rows and columns). Then, assign pointers to operators into the array. These pointers to operators can be changed within the `oparray`, but in general, changing the operators within an `oparray` indicates that you want a new `oparray`, rather than to reuse an old one.

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

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

DESCRIPTION

Class <type>oparray is an array of operators, which can be applied to a <type>spacearray. Both a <type>oparray and a <type>spacearray have 2 dimensions, 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>ops, but rather pointers to them. Thus, the <type>ops cannot be changed or deleted, or the <type>oparray will behave strangely.

PUBLIC OPERATIONS

<type>oparray(int i=1,int j=1)

Constructs a <type>oparray, with the number of rows and columns specified by i and j.

<type>oparray(<type>op &)

Creates a 1x1 <type>oparray containing the <type>op.

~<type>oparray()

<type>op *& operator()(int i,int j=0)

Returns a pointer to the <type>op at position (i,j) in the <type>oparray. This pointer, being a reference, can be changed, to put a different operator into the <type>oparray. Assigning an operator into the <type>oparray in this manner must be done by assigning the pointer to the <type>op. Generally, the method for creating a <type>oparray is define its size, and then assign the <type>ops into it.

<type>spacearray Forward(<type>spacearray &)
<type>spacearray Conjugate(<type>spacearray &)

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. Conjugate does the same, except with the transpose of the <type>oparray.

PRIVATE VARIABLES

int rows
int cols
<type> ** ops

PRIVATE OPERATION

<type>op *& getop(int,int)

SEE ALSO

<type>space, <type>spacearray, <type>op, <type>oparraychain, <type>oparrayconj

Class `oparrayconj` handles conjugating an `oparraybase`.

NAME
    <type>oparrayconj - Class to conjugate <type>oparrays.

SYNOPSIS
    #include <<type>opat.h>

DESCRIPTION
    <type>oparrayconj is a class, derived from <type>oparraybase, to automatically handle conjugation of a <type>oparray. When a <type>oparrayconj is made from a <type>oparray, it calls the Conjugate of the <type>oparray when Forward is called, and the Forward of the <type>oparray when Conjugate is called. The <type>oparrayconj class can also conjugate other <type>oparrayconj objects, and <type>oparraychain objects.

    Notice that <type>oparrayconj contains only a pointer to the <type>oparray, and has no copy or assignment operations. This means that a <type>oparrayconj must be declared, not used as a temporary. Also, the <type>oparray that is in the <type>oparrayconj must be maintained (not changed or deleted) for the <type>oparrayconj to work.

FRIEND OPERATION
    friend <type>oparrayconj conj(<type>oparraybase &)
        Create a <type>oparrayconj the is the conjugate of the <type>oparraybase.

PUBLIC OPERATIONS
    <type>spacearray Forward(<type>spacearray &)
    <type>spacearray Conjugate(<type>spacearray &)

    <type>oparrayconj(<type>oparraybase &)

PRIVATE VARIABLE
    <type>oparraybase * op

SEE ALSO
    <type>spacearray, <type>op, <type>oparraybase, <type>oparray, <type>oparraychain

The product of `oparrays` is done with class `oparraychain`.

NAME
    <type>oparraychain - Class to chain <type>oparrays.

SYNOPSIS
    #include <<type>opac.h>

DESCRIPTION
    Class <type>oparraychain is class derived from <type>oparraybase which

allows for automatic handling of products of <type>oparrays. Multiplying two <type>oparrays yields a <type>oparraychain which, when the Forward function is called, applies the second <type>oparray to the <type>spacearray, followed by the first <type>oparray. Because all three of <type>oparray, <type>oparraychain, and <type>oparrayconj are derived from <type>oparraybase, multiplying any of the types together will yield a <type>oparraychain. The Conjugate function calls the Conjugate of the <type>oparrays, in reverse order.

One thing that is important to notice is that the <type>oparraychain contains pointer to <type>oparrays, not the <type>oparrays themselves. Thus, the <type>oparrays must be maintained (not changed or deleted), or the <type>oparraychain will not work.

FRIEND OPERATION

friend <type>oparraychain operator*(<type>oparraychain,const <type>oparraychain &)

This function provides for the product of <type>oparraybases. The <type>oparraychain(<type>oparraybase &) constructor is used to change the <type>oparraybases to a <type>oparraychain, and then the two are chained together.

PUBLIC OPERATIONS

<type>spacearray Forward(<type>spacearray &)
<type>spacearray Conjugate(<type>spacearray &)

<type>oparraychain()
<type>oparraychain(<type>oparraybase &)
Constructs a <type>oparraychain containing one <type>oparraybase.

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

PRIVATE VARIABLES

int num
<type>oparraybase **ops

PRIVATE OPERATION

void append(<type>oparraybase &)

SEE ALSO

<type>spacearray, <type>op, <type>oparraybase, <type>oparray,

<type>oparrayconj

The solver routines also work for oparrays, except they must be rewritten for array operators and spaces. First the Hestenes solver. Then the PVI solver.

NAME

<type>arrayhestenes - Function to solve conjugate gradient problem.

SYNOPSIS

#include <<type>ahest.h>

DESCRIPTION

<type>spacearray <type>arrayhestenes(<type>spacearray x0,<type>spacearray & y,<type>oparray & A,int niter)

<type>arrayhestenes runs the Hestenes conjugate gradient solver on the problem

min || A x - y ||^2

given x0 as an initial guess for x. The solver currently disregards the initial guess. The solver runs for niter iterations, then returns its solution.

SEE ALSO

<type>spacearray, <type>oparray, <type>arraybook3, <type>hestenes

NAME

<type>arraybook3 - Function to solve conjugate gradient problem.

SYNOPSIS

#include <<type>abook3.h>

DESCRIPTION

<type>spacearray <type>arraybook3(<type>spacearray x,<type>spacearray & y,<type>oparray &

/fB<type>arraybook3/fP runs a conjugate gradient solver on the problem:

min || A x - y ||^2

given an initial guess for x. The solver runs for niter iterations, then returns its solution.

SEE ALSO

<type>spacearray, <type>oparray, <type>book3, <type>arrayhestenes

REFERENCES

Dyad Software Corporation, Renton, Washington. **M++ Class Library**, 1991.

Nichols, D., Dunbar, G., and Claerbout, J. F., 1992, The C++ language in physical science: SEP–**75**, 455–472.