PROGRAMMING GEOPHYSICS IN C++

Dave Nichols (foundations, inversion)
Hector Urdaneta (moveout applications)
Hyang Im Oh (missing data examples)
Jon Claerbout (outline, proofreading)
Lisa Laane (foundations)
Martin Karrenbach (moveout, document integration)
Matt Schwab (tutorial and convolution)

Contents

PREFACE

This paper document is an advertisement for an electronic document that lies behind it. We plan to distribute the electronic document on CD-ROM late September or early October 1993.

To run this version of the electronic document two commercial products are required. First is a C++ compiler. This book's code was tested using the Sun C++ compiler version 2.1 and the HP C++ 3.0 compiler. Both compilers are implementations of AT&T's C++ compiler (versions 2.1 and 3.0.1), which seems to be a UNIX industry standard. Hopefully, at a later date, we will be able to include a freeware C++ compiler that will also do the job. Secondly, our code uses the a commercial matrix class library (M++, from Dyad) as a foundation for some of our C++ classes. In the future we hope to use a public domain matrix class library to avoid this commercial dependence. We have no antipathy for commercial products, in fact we like them and appreciate them, but experience shows that our work will be accessible to the greatest number of people if our code depends only on noncommercial software.

We are collaborating with Les Dye who is affiliated with Berea and Stanford's petroleum reservoir simulation group (SUPRI-B), which is headed by Khalid Aziz. These groups are involved in research on object-based mathematical abstractions for reservoir simulation and nonlinear systems. We may base our future library designs on research that results from this collaboration.

A recently introduced C++ language feature known as "templates" allows developers to write one code that can be used for multiple data types. Since our current libraries only handle "float" data and not "complex" data, we hope to use templates to create a more general library later this year.

At mid August 1993 we have only begun to use the results of this C++ project for serious seismological research. We hope to demonstrate its utility further by summer's end.

To test whether your environment will build this C++ document, press this BUTTON to **destroy** all the figures in this document. Then view the document to see if all the figures are gone. Then press this BUTTON2, to rebuild all the figures. You could scan the document to see if all the figures have been regenerated, or press this BUTTON3 to see a statistical analysis of the reproducibility of the illustrations in this document.

Chapter 1

INTRODUCTION

Every few years brings a new computer language. Yet most scientific and engineering computing continues to be done in Fortran, a language that has hardly changed since 1977. Here we explore the hypothesis that a new language, C++, is a worthwhile alternative to Fortran.

C++ is based on the C language. The C language has made remarkable strides in the last several years. Formerly, Fortran was the language that was available on most machines and it had the fewest variations from one machine to another. Today, C replaces Fortran in both universality and uniformity. Now more young engineers and scientists learn C than any other language. C overcomes the memory-allocation limitation of Fortran, and offers new features such as data structures and scope. C however was not designed for numerical analysis and it does not handle matrices as conveniently as does Fortran. There has been little mass movement of scientific code from Fortran to C. We (and many others) have overcome some of Fortran's most glaring deficits by calling C-language subroutines and by using our preprocessors for memory allocation and parameter handling. In the past few years, C has become standardized, and new developments are now centered on the C++ language. C++ shares the advantages of C, but it adds features that permit the addition of new, more complex, data types to the language and allows the author to define all operations possible on the new data types. For instance C++ can handle the concept of arrays *at least* as well as Fortran (whereas C cannot.)

We do not predict that Fortran 90 will die and be replaced by C++, but there are some reasons why this might happen: the growth of C++ has been much more rapid. It already has many more users than Fortran 90. We find C++ compilers on all manner of computers (also the free GNU compiler) whereas our workstation manufacturers still do not offer Fortran 90. New parallel architectures present an opportunity for a malleable language like C++ and a risk for static languages.

C++ was especially designed to enable concepts to be isolated, leading to software that is better debugged and more reusable. A particular problem in geophysical inversion problems (see "Processing versus inversion" (PVI), for example, (Claerbout, 1992)) has been that the Fortran language leads to programs where the physical science is closely interwoven with the numerical analysis in the conjugate-gradient solver. One person (the programmer) must handle all the ingredients of both physics and numerical analysis specializations. Such code is challenging to write and even more challenging to debug! Further, in complicated cases, such as with nonlinear operations or linear operators of very high order, we cannot really be sure the code is debugged, which is disheartening.

Our ultimate goals are:

1. Write real code that is clean enough that we can include it in publications where we now use mathematics and pseudocode.
2. Reduce the "overlap expertise" required to merge scientific application code with numerical analysis code.
3. Write linear operators in such a way that they can be used by people other than the original author, and operators written by different people can easily be combined to produce new operators.

As a first step to meeting these goals, Nichols, Dunbar, and Claerbout (?) took a body of tutorial Fortran-Ratfor code in the geophysical textbook, "Earth soundings analysis: processing versus inversion" (PVI) by Jon Claerbout (1992) and began converting it to C++. This summer, more users/programmers were recruited: The base classes were improved, and many new operators were added. Some examples from Claerbout's book "Basic Earth Interior" (BEI) were converted to C++ form.

1.0.1 The C++ language

To ameliorate the problem of many of the readers of this document knowing little of C and C++, we recommend following learning pathway: Begin from any of the many dozens of C tutorial books concentrating on learning particularly well the ideas of pointer, structure, and typedef. Then switch to the book (?) which concerns itself mainly with the "class" and "inheritance" concepts, avoiding the full minutia of the C++ language.

Going beyond Fortran 77, the first concept needed is that of a structure. For example, in Fortran, if a single symbol could represent the idea of `(n,(x(i), i=1,n)` then that symbol would denote something like a structure. Another structure might be `(n1,n2, ((x(i1,i2), i1=1,n1), i2=1,n2)`. A numerical analyst thinking about solving the multivariate regression $\mathbf{y} \approx \mathbf{Ax}$ might need to allocate several instances of the structure representing $\mathbf{x}$. Clearly, the solver program written by the analyst should work for all possible structures. In Fortran, the application programmer ends out allocating the temporary arrays needed by the numerical analyst, leading to long, error prone calling sequences. In Fortran the work of the applications programmer and the numerical analyst are typically brought together by subroutines. C also uses subroutine libraries, and additionally it makes heavy use of "include files" that define structures thus serving to better isolate science from numerical analysis. C++

extends the structure concept to a "class," which is a structure containing subroutine pointers. C++ has elaborate provisions for sharing and hiding data, for defaulting and overriding functions. It also uses include files to implement a new method of merging the work of the applications programmer to the numerical analyst called "inheritance" or "derivation" whose explanation is beyond the scope of this brief summary but which should become clearer as you progress through this document.

1.1 WHAT IS A SPACE?

The well known word "space" can lead to considerable confusion because there are many entities that are described as spaces, these include continuous spaces, discrete sampled spaces, abstract infinite vector spaces, etc. In our C++ code the "$<$ `type` $>$ `space`" class describes objects that are a discrete sampling of a physical property in some geometric region, e.g. pressure measured on a plane. The word "$<$ `type` $>$" defines the numerical type of the elements of the space, currently `floatspace` is the only supported type, we hope to implement a `complexspace` class soon.

An abstract space in a numerical problem is a collection of values that could be a single physical space (a single instance of the $<$ `type` $>$ `space` class), or an assemblage of physical spaces. In our C++ code such assemblages are called $<$ `type` $>$ `spacearray`, i.e. an array of $<$ `type` $>$ `space` objects. In this paper we will use the word space to refer to the abstract concept and $<$ `type` $>$ `space` or $<$ `type` $>$ `spacearray` when we wish to refer to a particular representation of a space. The $<$ `type` $>$ `spacearray` can be thought of as a generalization of the single $<$ `type` $>$ `space`; in our code a single $<$ `type` $>$ `space` may be automatically converted to a $<$ `type` $>$ `spacearray` containing a single element.

1.1.1 The floatspace class

An object of class `floatspace`, the class to represent a regulary sampled physical space, has two parts. The first part is a multi-dimensional array of numbers representing the sampled data or model. This array contains floating point values. The second part is a set of axes that describe the physical dimensions of region that the samples are taken from.

Every dimension of a `floatspace` has an axis associated with it, which defines an origin, length, and sampling interval (delta) for the axis, as well as a label. Thus, a `floatspace` is meant as a collection of real physical data, rather than an abstract collection of numbers. This is ideal for many scientific applications, as in geophysics.

The publicly accessible member functions that manipulate objects of the class `floatspace` are relatively few, as operators are the main method of transforming a space. The most important member functions are:

- One or more constructors, creates an instance of the class.
- A destructor, destroys an instance of the class.
- norm(), returns the L_2 norm of the space.
- norm2(), returns the square of the L_2 norm.
- dot(), takes the inner product with another (conforming) space.
- Arithmetic operators that add, multiply, etc. by a scalar.
- Arithmetic operators that add, multiply, etc. conforming spaces.
- An indexing operator that accesses the elements of the space.

There are two ways to create a space:

1. The first is to read it from a SEP cube dataset on disk. The SEP cube consists of two parts, an ASCII header file that contains the parameters describing the dataset, and a data file containing the actual data. The parameters are used to initialize the axes of a `floatspace` and the datafile is read to initialize the matrix. A `floatspace` can be constructed by passing it a pointer to an object of class `SepInput`. The SepInput class is a C++ class that encapsulates interaction with a SEP cube dataset (Dulac and Nichols, 1989).
2. The second way to create a space is to create a matrix, using the matrix constructors, and also an Axislist, which is the class that defines the axes for a space. In our applications this method is seldom used, except in generating test cases. The dot-product test routine described in the appendix uses this type of constructor to build spaces containing random numbers.

All the basic arithmetic operations (addition, subtraction, multiplication, and division) are defined for spaces, both by single values and element-by-element by a conformable space. A conformable space is one that has matching shape and axes. If you have one space that is a 10×200 function of space and time it cannot be added to a 10×200 function of velocity and zero-offset-time. Even if two spaces are both 10×200 functions of space and time, they can only be added if the axis starting values and increments match. This high level of error checking is invisible to the user of the class but it helps prevent many errors. Both coding errors and conceptual errors have been detected by these checks.

Assuming space1, space2 and space3 are conformable spaces, the following code fragment demonstrates the manipulation of spaces.

```
space1 = space2 * 2.0 + .001;
space2 = space1 + space3;
```

Note that this code fragment would also be legitimate code for objects that were space-arrays as long as the arrays, and the individual elements were conformable.

The details of the internal structure of a space are hidden from the user and the mathematical operators are provided by the space class. This results in very clean code that is very close to the mathematical representation of the operations required.

Anything else you wish to do to a space is done by an operator. Operators are written in such a way that they have direct access to the private members of a space (the mechanism for doing this in C++ is to make them a "`friend`" class to the space class.) They can directly access the matrix class that is at the heart of a space. This gives operators extra power in manipulating spaces, since operators are designed to be the primary method for transforming spaces.

1.2 OPERATORS

A linear operator on a multidimensional sampled space can be defined as an extended matrix multiply. That is, it is a tensor product from one set of dimensions to another set of dimensions. It can sum along any number of dimensions, eliminating those dimensions. Also, it can expand any number of dimensions, adding new dimensions. Matrix multiplication is a particular case of a linear operator which sums along one dimension, and expands along another.

In most cases we do not handle operators as matrices because their application as a matrix multiply would be unnecessarily inefficient (e.g. discrete Fourier transform) or their storage as a matrix would be unnecessarily large (e.g. partial differential equations). Our operators are not represented as matrices — we define them *procedurally* by a computer subroutine pair. The first subroutine is equivalent to the matrix multiplication,

$$\mathbf{Ax}.$$

The adjoint subroutine computes

$$\mathbf{A}'\mathbf{y},$$

where $\mathbf{A}'$ is the (complex conjugate) transpose matrix. $\mathbf{x}$ is a vector in the model space (range) of the operator and $\mathbf{y}$ is a vector in the data space (domain) of the operator.

When we solve a problem involving the operator both the operator, and its adjoint are often required. Solution methods generally fail if the practitioner provides an inconsistent pair of subroutines. This can be checked by using the dot-product test as described in a later section. The coding of any operator is not finished until a dot-product test has been passed.

In our C++ code the base operator class "`foperator`" defines an operator that maps an input `floatspacearray` to an output `floatspacearray`. The derived class "`fop`" defines an additional mapping that maps a single `floatspace` to a single `floatspace`.

An operator only has three public functions

- The constructor
- The Forward() function that is applied to a model space and returns a data space.
- The Adjoint() function that is applied to a data space to produce a model space.

Our operators are procedural operators. When the operator is constructed (initialized), it is told what axes to apply itself to. At this point, it knows what axes, size, label, etc., it is expecting, and what axes it will output. Then, when the operator is applied to a space, it will automatically find the correct axes to apply itself to. Any extra axes will merely be cycled through, applying the same operator to each index in that dimension. This ability to cycle through extra dimensions is built into the class `fopone` described in a later chapter. Any class that is derived from the `fopone` operator class inherits this ability without the author having to write any code to implement it.

1.2.1 Chains and transposes

We often string together two operators to get a third. Because we generally deal with linear operators that are procedurally defined we cannot regard

$$\mathbf{z} = \mathbf{A}\mathbf{B}\mathbf{x}$$

as

$$\mathbf{z} = (\mathbf{A}\mathbf{B})\mathbf{x}$$

If the operators were matricies we could construct a new matrix that was $\mathbf{AB}$. Since they are defined as procedures we must define the composition of the two operators as a sequence of procedures and we must regard it as

$$\mathbf{z} = \mathbf{A}(\mathbf{B}\mathbf{x}).$$

Our Fortran programs interpret this as meaning that first memory must be allocated for $\mathbf{y} = \mathbf{B}\mathbf{x}$; $\mathbf{y}$ must be erased; $\mathbf{y} = \mathbf{y} + \mathbf{B}\mathbf{x}$ should be computed; $\mathbf{z}$ should be erased; and finally $\mathbf{z} = \mathbf{z} + \mathbf{A}\mathbf{y}$ should be computed. Coding all of this is error prone.

In C++, if you wish to form an operator $\mathbf{C}$ that is $\mathbf{A}$ acting on $\mathbf{B}$, you simply write:

```
fopChain C = A * B
```

The class "`fopChain`" encapsulates the behavior of an operator that is a composite of other operators applied in sequence. When the new operator `C` is applied it will

act as though
```
C.Forward(x) ≡ A.Forward( B.Forward( x ) )
C.Adjoint(y) ≡ B.Adjoint( A.Adjoint( y ) )
```

The application programmer does not need to do anything to ensure that the adjoint of the composite operator is correctly applied. When a chained operator is applied in the adjoint sense it will be applied with each component operator applied adjoint and applied in the reverse order. This removes another possible source of errors in writing new code.

A similar helper class is available to create an operator that is the adjoint of another operator, it is also very simple to use:

```
fopAdjoint B(A);
```

Application of `B.Forward()` will be equivalent to applying `A.Adjoint()`.

1.2.2 Operator arrays

An operator array is a two dimensional array which contains operators. Applying a `foparray` to a `floatspacearray`:

$$\mathbf{u} = \mathbf{A}\mathbf{v}$$

works like normal matrix multiplication. All three of $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{A}$ are two dimensional arrays (of operators or spaces.)

$$\begin{pmatrix} \mathbf{u}_{11} & \cdots & \mathbf{u}_{1n} \\ \vdots & \ddots & \\ \mathbf{u}_{m1} & \cdots & \mathbf{u}_{mn} \end{pmatrix} = \begin{pmatrix} \mathbf{A}_{11} & \cdots & \mathbf{A}_{1k} \\ \vdots & \ddots & \\ \mathbf{A}_{m1} & \cdots & \mathbf{A}_{mk} \end{pmatrix} \begin{pmatrix} \mathbf{v}_{11} & \cdots & \mathbf{v}_{1n} \\ \vdots & \ddots & \\ \mathbf{v}_{k1} & \cdots & \mathbf{v}_{kn} \end{pmatrix}$$

$\mathbf{u}$ is a space array that has the number of rows of $\mathbf{A}$, and the number of columns of $\mathbf{v}$. It is not possible to apply an operator array to an array of unsuitable size without generating a run time error. The rows of $\mathbf{A}$ are applied to the columns of $\mathbf{v}$ and the result is summed. As each element of the operator array is applied to the appropriate element of the space array, run time compatibility checks will take place. e.g. $\mathbf{A}_{11}$ and $\mathbf{v}_{11}$ must be compatible. Also the results to be summed must be compatible, the space that results from $\mathbf{A}_{11}\mathbf{v}_{11}$ must be compatible with the space that results from $\mathbf{A}_{12}\mathbf{v}_{21}$.

In C++, this looks like:

```
floatspacearray u = B.Forward(v);
```

This is almost identical to the syntax for applying a single operator to a single space, except here the variables `u` and `v` are of class `floatspacearray` and the operator `B` is of class `foparray`.

A simple example of using arrays is to generate a damped operator from the regular operator. If we wish to calculate the damped solution to the problem

$$\min_x |\mathbf{Ax} - \mathbf{y}|^2$$

We can augment the problem with a diagonal damping matrix and solve:

$$\min_x \left| \begin{pmatrix} \mathbf{A} \\ \epsilon\mathbf{I} \end{pmatrix} \mathbf{x} - \begin{pmatrix} \mathbf{y} \\ \mathbf{0} \end{pmatrix} \right|^2 \tag{1.1}$$

The new operator, $\mathbf{A_2}$ can be written as

$$\mathbf{A_2} = \begin{pmatrix} \mathbf{A} \\ \epsilon\mathbf{I} \end{pmatrix}$$

When applied to a space, $\mathbf{x}$ it gives a space-array,

$$\mathbf{A_2x} = \begin{pmatrix} \mathbf{Ax} \\ \epsilon\mathbf{Ix} \end{pmatrix}$$

When the adjoint is applied to a space-array it gives a space,

$$\mathbf{A_2'} \begin{pmatrix} \mathbf{y_1} \\ \mathbf{y_2} \end{pmatrix} = (\mathbf{A'}, \epsilon^*\mathbf{I}) \begin{pmatrix} \mathbf{y_1} \\ \mathbf{y_2} \end{pmatrix} = \mathbf{A'y_1} + \epsilon^*\mathbf{Iy_2}$$

All of this is handled automatically by the `foparray` class.

1.3 SOLVERS

In our class library we have a class of objects called "`LSSolver`", these are objects that solve the least-squares minimization problem associated with an operator. There are many classes derived from `LSSolver` that use different methods to solve the problem. The solver classes operate at the most abstract level, they take as arguments objects of class `foperator` and `floatspacearray`. Any class derived from these can be used in the solution algorithm.

The author of the operator class and the author who sets up the particular problem do not have to modify any of the solver code, and the author of a solver routine does not need to assume anything about the operator that will be used other than they must be derived from one of the base operator classes.

This design of the operator and space classes separates the jobs of implementing a particular operator and writing a routine that solves the linear problem. The scientists can concentrate on writing code specific to their domain of expertise and then ask the local scientific computing specialist to write a spiffy solver routine for them. In contrast, when writing in Fortran we typically need to write a new solver routine for each new operator that we implement.

Routines such as solvers and dot-product tests can be written once, and then just called with operators and spaces as arguments. One major task that lies ahead is to implement more types of solver routines. We wish to compare different algorithms for non-Hermitian problems, the use of preconditioners and operators for nonlinear optimizations. Since these routines can be written in terms of abstract operator and space classes we will only have to write these routines once, they will then be available for use on any problem. In an ideal world we would hope to collaborate with members of the computer science department in writing these routines. We would be able to provide many examples of real world problems and a framework in which to use them and they would be able to provide new algorithms for solving the problems.

1.4 ADJOINTS AND THE DOT-PRODUCT TEST

A great many of the calculations we do in science and engineering are really matrix multiplication in disguise. One goal of this book is to unmask the disguise by showing many examples. Second, we will illuminate the meaning of the **adjoint** operator (matrix transpose) in these many examples. Third we show how to write computer code in C++.

Modeling calculations generally use linear operators that predict data from models. We often need to find the inverse of these calculations, i.e., to find models (or make maps) from the data. Logically, the adjoint is the first step and a part of all subsequent steps in this **inversion** process. Surprisingly, in practice the adjoint sometimes does a better job than the inverse! This is because the adjoint operator tolerates imperfections in the data and does not demand that the data provide full information. Once you grasp the relationship between operators in general and their adjoints, you can have the adjoint just as soon as you have learned how to code the modeling operator.

If you will permit me a poet's license with words, I will offer you the following table of examples of **operator**s and their **adjoint**s:

operator	adjoint
matrix multiply	conjugate-transpose matrix multiply
convolution	crosscorrelation
stretching	squeezing
zero padding	truncation
causal integration	anticausal integration
add functions	do integrals
plane-wave superposition	slant stack
superposing on a curve	summing along a curve
upward continuation	downward continuation
diffraction modeling	imaging by migration
hyperbola modeling	CDP stacking
ray tracing	tomography

The left column above is often called "**modeling**," and the adjoint operators on the right are often used in "data **processing**."

When the adjoint operator is *not* an adequate approximation to the inverse, then you apply the techniques of fitting and optimization which require iterative use of the modeling operator and its adjoint.

The adjoint operator is sometimes called the "**back projection**" operator because information propagated in one direction (earth to data) is projected backward (data to earth model). With complex-valued operators the transpose and complex conjugate go together and in Fourier analysis, taking the complex conjugate of $\exp(i\omega t)$ reverses the sense of time. Still assuming poetic license, I will say that adjoint operators *undo* the time and phase shifts of modeling operators. The inverse operator does this too, but it also divides out the color. For example, with linear interpolation high frequencies are smoothed out, so inverse interpolation must restore them. You can imagine the possibilities for noise amplification. That is why adjoints are safer than inverses. But nature determines in each application what is the best operator to use, whether to stop after the adjoint, to go the whole way to the inverse, or to stop part-way.

We will see that computation of the **adjoint** is a straightforward adjunct to the computation itself, and the computed adjoint should be, and generally can be, exact (within machine precision). If the application's operator is computed in an approximate way, we will see that it is natural and best to compute the adjoint with adjoint approximations.

There is a huge gap between the conception of an idea and putting it into practice. During development, things fail far more often than not. Often, when something fails, many tests are needed to track down the cause of failure. Maybe the cause cannot even be found. More insidiously, failure may be below the threshold of detection and poor performance suffered for years. I find the **dot-product test** to be an extremely valuable checkpoint.

Conceptually, the idea of matrix transposition is simply $a'_{ij} = a_{ji}$. In practice, however, we often encounter matrices far too large to fit in the memory of any computer. Sometimes it is also not obvious how to formulate the process at hand as a matrix multiplication. (Examples are differential equations and fast Fourier transforms.) What we find in practice is that an application and its adjoint amounts to two subroutines. The first subroutine amounts to the matrix multiplication $\mathbf{Ax}$. The adjoint subroutine computes $\mathbf{A'y}$, where $\mathbf{A'}$ is the conjugate-transpose matrix. In a later chapter we will be solving huge sets of simultaneous equations. Then both subroutines are required. We are doomed from the start if the practitioner provides an inconsistent pair of subroutines. The dot-product test is a simple test for verifying that the two subroutines are adjoint to each other.

According to texts on linear algebra the "Hilbert Adjoint" operator is defined as the operator $\mathbf{A'}$ that satisfies the following relationship for any $\mathbf{x}$ and $\mathbf{y}$.

$$< \mathbf{y} \ , \ \mathbf{A}(\mathbf{x}) > = < \mathbf{A'}(\mathbf{y}) \ , \ \mathbf{x} > . \tag{1.2}$$

Where $< \mathbf{a}, \mathbf{b} >$ is the inner product of $\mathbf{a}$ and $\mathbf{b}$.

In matrix notation this can be written in the more familiar form

$$\mathbf{y}'(\mathbf{Ax}) \ = \ (\mathbf{A'y})'\mathbf{x}$$

The vector inner product is the adjoint of the first vector dotted with the second vector.

In our C++ classes the inner product is obtained by using the `dot()` member function of a `floatspace` or `floatspacearray`.

For the dot-product test, load the vectors $\mathbf{x}$ and $\mathbf{y}$ with random numbers. Compute the vector $\tilde{\mathbf{y}} = \mathbf{Ax}$ using your program for $\mathbf{A}$, and compute $\tilde{\mathbf{x}} = \mathbf{A'y}$ using your program for $\mathbf{A'}$. Inserting these into equation (1.2) should give you two equal scalars

$$< \mathbf{y}, (\mathbf{Ax}) > \ = \ < \mathbf{y}, \tilde{\mathbf{y}} > \ = \ < \tilde{\mathbf{x}}, \mathbf{x} > \ = \ < (\mathbf{A'y}), \mathbf{x} > \tag{1.3}$$

The left and right sides of this equation will be computationally equal only if the program doing $\mathbf{A'}$ is indeed adjoint to the program doing $\mathbf{A}$ (unless the random numbers do something miraculous).

Using the C++ classes we can write a single dot-product test routine that all new operators can be tested with. We only need to write a main program that supplies the operator, and a definition of the data space, as arguments. The test routine will generate spaces of random numbers and check that the dot product test condition is satisfied. The ability to construct chains and arrays of operators allows us to make the following statement; if all the individual operators in the chain or array pass the dot-product test then the composite operator will always pass the dot-product test. Thus a huge family of coding errors possible in Fortran are not possible in C++.

Do not be alarmed if the operator you have defined has **truncation** errors. Such errors in the definition of the original operator should be identically matched by truncation errors in the adjoint operator. If your code passes the **dot-product test**, then you really have coded the adjoint operator. In that case, you can take advantage of the standard methods of mathematics to obtain inverse operators.

We can speak of a continuous function $f(t)$ or a discrete one f_t. For continuous functions we use integration, and for discrete ones we use summation. In formal mathematics the dot-product test *defines* the adjoint operator, so the definition if the inner product is fundamental to the definition of the adjoint operator. The input or the output or both can be given either on a continuum or in a discrete domain. So the dot-product test $< \mathbf{y}, \tilde{\mathbf{y}} >=< \tilde{\mathbf{x}}, \mathbf{x} >$ could have an integration on one side of the equal sign and a summation on the other. Linear-operator theory is rich with concepts, but we will not develop them in this book on its applications and C++ implementation.

In mathematics the word "**adjoint**" has three meanings. One of them, the so-called **Hilbert** adjoint, is the one generally found in physics and engineering and it is the one used in this book. In matrix algebra there is a different matrix, called the **adjugate** matrix. It is a matrix whose elements are signed cofactors (minor determinants). For invertible matrices, this matrix is the determinant times the inverse matrix. It is computable without ever using division, so potentially the adjugate can be useful in applications where an inverse matrix cannot. Unfortunately, the adjugate matrix is sometimes called the adjoint matrix particularly in the older literature. For that reason we often find the usage "conjugate-transpose", "transpose", or "conjugate" in the literature where Hilbert adjoint is what is intended.

Chapter 2

Using and understanding an operator

[1]This chapter introduces you to designing a new operator for SEP's C++ linear operator (CLOP) library. You do not need to know these details if you merely want to apply an already existing operator. This document does not expect the reader to be proficient in C++, but it will not explain C++.

The chapter is based on my experiences when implementing a simple convolution operator. When I coded this operator, I had previously used neither C++ nor SEP's CLOP library. I hope that people with a similar slim background will be able to build simple geophysical operator following this outline.

2.1 DIRECTORY STRUCTURE

The directory containing the 1-D convolution operator contains a set of short files. The `paper.tex` file holds the words you are reading. This directory takes advantage of certain text-stripping scripts, which allow me to include in this `paper.tex` file certain essential lines from my source files (e.g. the prototype constructor calls).

The `Cakefile` maintains the files of the directory. It is very similar to the Unix `Makefile`. Every chapter in this book is supported by a directory with a `Cakefile`. Since most of these `Cakefiles` have basic default rules in common, it is best to implement these rules in a canonical file and to include it into all the `Cakefiles`. The `Cakefile` in this directory includes four canonical files:

`SEP.clop.defs` contains the definition of convenient macros for compiling and linking C++ routines.

`SEP.idoc.rules` holds rules for removing and recomputing reproducible figures in interactive documents (Claerbout and Karrenbach, 1993).

[1]Matthias Schwab

`SEP.clop.rules` supplements the `SEP.idoc.rules` with C++ specific rules.

`SEP.obj.rules` lists rules for compiling source files according to their filename's suffix.

In this directory, the following files contain C++ code: `fopcontran.h` [2], `fopcontran.cc`, `Contran.cc`, and `Dottest.cc`. The header file `fopcontran.h` defines the class. It includes whatever an application program needs in order to use the class. The file `fopcontran.cc` implements the constructors and members of the class. In particular, the operator's scientific task is coded here. Finally, the test routine `Contran.cc` handles input and output of data, builds an object of class `fopContran` and applies this instance of the operator to some data to make a figure. Periodically, we destroy and rebuild all the figures in the book to test changes in the code.

The file `Dottest` conducts the dot-product test (see chapter 1.4) for the convolution operator. At this point, I will not explain the mathematical meaning of the dot-product test, but I want to point out that every linear operator and its adjoint will pass the dot-product test if they are correctly coded. Every newly implemented linear operator should be checked by an appropriate `Dottest` routine.

If you see any other files in this directory, they are probably intermediate outputs of the test procedures. If so, they should go away when you type `cake clean`.

2.2 A PHYSICAL OPERATOR CLASS

Unlike data in numerical analysis, physical data always involve physical dimensions. Without physical dimensions, a physical interpretation of the computational result is impossible. Furthermore, keeping track of dimensions prevents errors.

In CLOP, physical dimensions are packaged into a set of axis values. Each axis is described by an axis name, an original offset, the sample rate, and the number of samples. Generally, each CLOP operator has to be supplied with two sets of axes information: one set describes the "input space"; the other set describes the "output space". On the other hand, each CLOP data set contains, besides the data itself, a corresponding set of axes descriptions. The operator's input axes information has to be matched by any data the operator is applied to. The axes description of the output data is derived from the operator's output axes information.

In CLOP's C++ code, the task of the operator implementation is split in a way which is unusual for someone accustomed to traditional programming. The linear operator is implemented as a "class" rather than a subroutine. A class is a fundamental feature of the C++ language which generalizes the C concept of a "structure". A class usually contains a sequence of data members of various types, a set of functions manipulating these members, and a set of restrictions on the access to its members.

[2] Think of `fopcontran` as "float operator for convolution". Conventionally, the operator class and its files are named identically except that the class name starts the unique name part with a capital letter like in `fopContran`.

Generally, the CLOP class definition of an operator defines the abstract structure of this operator. For example, the convolution class implements convolution independently of what filter coefficients the user may choose, or what data the user may want to apply the operator to. In a specific application program like `Contran.cc`, I create an instance of the operator by "constructing" an object of type `fopContran` with a specific filter (e.g. the difference filter (1,-0.5)). I can then apply this filter operator to any compatible input data and it will return the corresponding output. Figure 2.1 demonstrates this by displaying the input vector, the filter coefficients, and the output vector of `Contran.cc`.

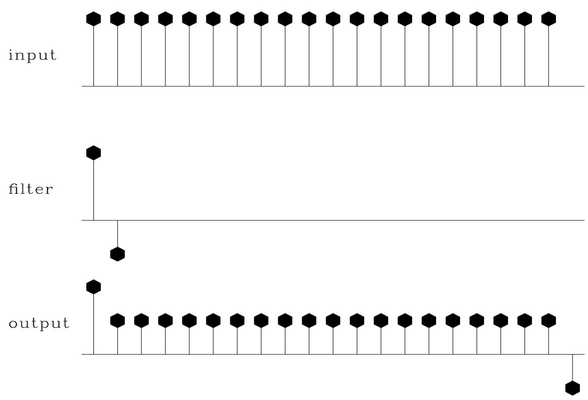

Figure 2.1: Demonstration of convolution. From top to bottom: (1) input; (2) filter; (3) output of `fopcontran()`. underst-contranin1 [R]

By first constructing the operator object before applying it, the CLOP library enforces a clear distinction between the input parameters (filter coefficients), which instantiate the operator object, the input data, which the operator is applied to, and the operator's output. In traditional programming the input and output are often indiscriminately clumped together in a single subroutine argument string.

2.3 THE TEST PROGRAM

The test program `Contran.cc` accomplishes three distinct tasks. It reads in the filter and the input data, it then constructs the operator object using the filter coefficients and the axis information, and finally it applies the object operator to the given input data.

Input and output

In the test program `Contran.cc`, the bulk of statements deals with the reading and writing of data. The input-output operations of the test program contain site specific code. (If you have this document's CD-ROM version, it will contain all required site specific code.) In all other chapters of this document, we will concentrate on the operator class and we will avoid discussing the test routine. However, for the sake of completeness, I show this site specific code in this chapter.

In the past twelve years the Stanford Exploration Project has developed a seismic processing environment SEPlib (Dellinger and Tálas, 1992), for which Dulac and Nichols (1989) developed a set of C++ classes which read and write SEPlib files.

In `Contran.cc` an object of class `SepInput` (man pages `sepinput` on page 459) is used to handle the communication with the SEP header files of the input data. The function `Sepinit()` is called using the standard C variables `argc, argv` and the SEPlib argument `SOURCE`[3]. `Sepinit()` returns a pointer to a `SepInput` object named `sepInput`. This `sepInput` object is used to read the parameter `adj` and `filter` from the command line. Furthermore, the object `sepInput` contains a reference to the input data which is indicated on the command line.

```
/* Contran.cc:  Contran filter=myfilter.H  adj=0 < in.H > out.H  */
#include <iostream.h>
#include <SEPinit.h>
#include <fopcontran.h>
#include <farray.h>
#include <Processnew.h>
main(int argc,char **argv) {            // site specific input
    SepInput * sepInput = SEPinit(argc,argv,SOURCE);
    floatspace input(sepInput);
    int adj  = 0;             sepInput->Getch("adj",adj);
    int inv  = 0;             sepInput->Getch("inv",inv);
    int niter= 1;             sepInput->Getch("niter",niter);
    char filname[100];  if (!(sepInput->Getch("filter",filname)))
                          seperr("filter must be specified.\n");
    SepInput * sepinfilter = new SepInput(filname);
    floatspace filtsp(sepinfilter);

    fopContran mycontran(filtsp,0);     // make contranolve operator
    floatspacearray output = Processnew( input, mycontran, adj, inv, niter );
                                        // apply operator to input data

    SepOutput *sepout = new SepOutput( "stdout", output(0,0).sepData() );
    delete sepout;                      // site specific output
    UnRef(sepinfilter);
    UnRef(sepInput);
    return 0;
}
```

The name of the SEPlib header file associated with the filter coefficients is taken as an argument to obtain a new pointer to an object of type `SepInput` called `sepinfilter`, which references the file containing the filter coefficients. Then an object `filtsp` of class `floatspace` is constructed from `sepinfilter`. The `floatspace` class is designed to hold arrays of data and associated axis information. The definition of `floatspace` is not site specific (see man pages `space` on page 464).

Creating the filter operator

Having created the necessary objects for reading data, the test program creates an object `mycontran` of the convolution class `fopContran`.

[3] `SOURCE` refers to the directory location of a SEPlib file.

```
fopContran mycontran(filtsp,0);
```

As arguments it takes a reference to the `floatspace` object `filtsp` containing the filter information and a flag indicating not to check the axis information when applying the operator.

Applying the filter operator

The operator `mycontran` is applied to the test data `input` by the line:

```
floatspacearray output = Processnew( input,mycontran,adj,inv,niter );
```

The flag `adj=0` indicates that `mycontran`'s forward method (convolution) is the operation I want to apply. The flag `inv` signals if the given test data `input` and the designated operator are posing an inversion problem. If `inv==1`, `Processnew` invokes an iterative solver and `niter` sets the number of iterations. Since in `Contran.cc` `inv` is set to zero, the operator is simply applied to the test data `input` (see chapter 5).

`Processnew` invokes the necessary member functions of the operator class `fopContran` to compute the convolved data `output`. Since all the application specific code is buried within the operator class, `Processnew` can apply any operator object of CLOP to any compatible input data.

The `UnRef()` statements at the end of `Contran.cc` free the resources used by the various objects. Deleting the object `sepout` causes its destructor to write the filtered output data to the standard output.

2.4 THE CONVOLUTION CLASS

2.4.1 The file fopcontran.h

The file `fopcontran.h` defines the operator "class" for convolution. Fortran users can think of the constructors in `fopcontran.h` as a list of alternative call syntaxes to create a convolution operator.

```
// fopcontran.h
#ifndef _OP_CONTRAN_F_
#define _OP_CONTRAN_F_

#include <fopone.h>
#include <Axislist.h>
#include <Cube/sepinput.h>

class fopContran : public fopone {
  public:
       // construct operator from an array; apply to axis axisname;
       // sample interval of input data must be delta;
    fopContran( floatArray & filt,const char* axisname, float delta );
       // sample interval of input data is irrelevant;
```

```
      fopContran( floatArray & filt,const char* axisname );
         // construct operator from floatspace;
         // if usedelta==0 then the sample rate of data is irrelevant;
      fopContran(const floatspace & filtsp, int usedelta = 1);
    private:
      void apply(floatArray &,floatArray &,int,
                  const Axislist &,const Axislist &) const;
      void forwardwildtrans (Axislist &,const Axislist &) const;
      void adjointwildtrans (Axislist &,const Axislist &) const;
      floatArray filter;
  };

  #endif
```

The '`ifndef`' flag at the beginning of `fopcontran.h` prevents multiple definition of the operator `fopContran`.

The class `fopContran` is defined as a subclass of class `fopone`[4], where `fopone` is a class which knows about data and function members common to all linear operator. The `fopone` class for example, includes the functions which manipulate the axes when the operator is applied (see man pages `opone` on page 438). Since `fopContran` inherits these members from `fopone`, the code that remains to be written can exclusively concentrate on what sets `fopContran` apart from all other linear operator in the CLOP library.

All the functions publicly available to the user are constructors. These constructors are used to initialize an object. Since C++ distinguishes the type of arguments in a constructor call, several constructors can be defined using different sets of arguments. When an instance of a class is constructed by an user, C++ chooses the appropriate constructor depending on the argument types.

The remaining members are private and can only be used within the class. All the members are functions, except the data member `filter`. The scientifically interesting method of a class is `apply()`, since it, as we shall see, contains the code implementing the physical operator.

2.4.2 The file fopcontran.cc

The `fopcontran.cc` file contains the implementation of the different member functions of the class defined in `fopcontran.h`. The public members of `fopcontran.cc` are constructors and are used to instantiate a filter operator. The private members implement the functionality of a convolution operator. Only the operator object has access to private members.

```
// fopcontran.cc
#include <fopcontran.h>
#include <Axis.h>

fopContran::fopContran( floatArray & filt,const char * axisname) {
    Axis inax(0,0,0,axisname,1,1,1);      // Make an axis to be matched
    inaxis = (Axislist) inax;
    outaxis = inaxis;
```

[4] Note, that its header file is included.

```
        filter = filt;
    }
    fopContran::fopContran(floatArray & filt,const char *axisname, float delta) {
        Axis inax(0,delta,0,axisname,1,0,1);  // Make an axis to be matched
        inaxis  = (Axislist) inax;
        outaxis = inaxis;
        filter = filt;
    }
    fopContran::fopContran(const floatspace & filtsp, int usedelta) {
        Axislist ax = getinfo(filtsp);        // get the axis list.
        filter = getdata(filtsp);             // get the filter coefficients
        for( int i=1; i< ax.ndim ; i++ ){     // check input is 1D
            if( ax.list[i].length != 1 ){
               cerr << "floatContran needs 1-D Space as input.\n";
               exit(1);
        }   }
        // Make an axis to be matched
        Axis inax(0, ax.list[0].delta, 0,ax.list[0].name,1,!usedelta,1);
        inaxis = (Axislist) inax;
        outaxis = inaxis;
    }
    void fopContran::forwardwildtrans(Axislist & out,const Axislist & in ) const {
        out.list[0]=in.list[0];
        out.list[0].length += ( ::rows(filter) - 1 );
    }
    void fopContran::adjointwildtrans(Axislist & out,const Axislist & in ) const {
        out.list[0]=in.list[0];
        out.list[0].length -= ( ::rows(filter) - 1);
    }
    void fopContran::apply(floatArray & x,floatArray & y,int adj,
                           const Axislist & xal, const Axislist & yal) const {
        int nb = (::rows(filter)); int nx = xal.list[0].length;
        for (int ib=0; ib < nb ; ib++) {
        for (int ix=0; ix < nx ; ix++) {
            if (adj)
               x(ix)    += y(ib+ix) * filter(ib);
            else
               y(ib+ix) += x(ix)    * filter(ib);
    }   }}
```

The constructors

The `fopcontran.cc` file comprises three constructors. Every constructor 'instantiates' a filter operator when given a set of filter coefficients. However, these constructors distinguish themselves by the format they expect their arguments to be in.

The first public constructor's argument `filt` is a reference[5] to an object of type `floatArray` which holds the filter coefficients. The object `floatArray` is an M++ class (?). The constructor's second argument `axisname` designates which axis of the possibly multidimensional input the one-dimensional filter is applied to.

In its first line of code, the constructor instantiates an object `inax` of type `Axis` (see man page on page 413). The set of zeros defines the number of samples, the sample rate, and the offset of the `Axis inax`. The set of ones declare these values as

[5] A C++ "reference" is a C pointer except that code using this pointer does need not dereference it, so such code resembles more Fortran than C. A reference of type `mytype` is declared by a line as `mytype & referencename`.

being irrelevant for later consistency checks.

Casting the `Axis` object `inax` to a variable `inaxis` of type `Axislist` (see man pages `Axislist` on page 415), the program initializes a second, identical `Axislist` object `outaxis`. Since both objects are of type `Axislist`, the equal sign is overloaded according to the rules of the class `Axislist`. Both `inaxis` and `outaxis` are member functions of `fopContran`, which inherited them from `fopone`. Later when an object of class `fopContran` is applied to a data set the axislist `inaxis`, the axislist of the input are compared for compatibility. This ensures that the operator is only applied to an input it is designed for. The `outaxis` information is used to create the axis information of the filtered output data. If the adjoint operator is applied, the roles of `inaxis` and `outaxis` reverse; the `outaxis` is used for the compatibility checks and `inaxis` is used to create the `output` axis.

In the last statement of the constructor, the reference to the array of filter coefficients is assigned to the private `fopContran` member `filter`.

The only difference between the first and second constructor is the existence or non-existence of the axis increment `delta` in the constructor argument list. If a `delta` does appear in the argument list, the second constructor is invoked. The instantiation of `inax` assigns the delta value to `inax` and declares it as relevant.

The first argument to the last `fopContran` constructor is a `floatspace`[6] holding the filter. The second constructor argument is an integer flag which signals if the filter's sample rate is relevant or not.

The functions `getinfo()` and `getdata()` are defined within the base class `foperator`. `foperator` is defined as 'friend' of the class `floatspace`, which gives it access to the axis information and data. Since `fopContran` is a derived from `foperator`, it inherits and can use these functions.

The `getinfo()` function exploits the header information associated with the filter. It returns an object `ax` of type `Axislist`. The `getdata()` function reads the filter coefficients and returns a `floatArray` object which I again assign to the previously defined `fopcontran` member `filter`.

Having checked that the supplied filter is one dimensional, the constructor instantiates an object `inax` of type `Axis`. The flag `usedelta` signals if the sample rate of the filter is relevant for later argument checking or if it is to be ignored. Then `inax` is cast to `Axislist` object and assigned to `fopcontran`'s inherited members `inaxis` and `outaxis`.

In summary, all constructors assign a `floatArray` of filter coefficients to the `fopcontran` member `filter` and the relevant axis information to its members `inaxis` and `outaxis`.

[6] As I showed in the discussion of the main program `Contran.cc`, a `floatspace` object can be generated from a given SEPlib header file.

The private members

The first two private member function `forwardwildtrans` and `adjointwildtrans` take care of the wildcarded axes descriptors. As mentioned earlier, the operator's member functions `inaxis` and `outaxis` serve two tasks: one is used for a compatibility check of the input data, and the other one is used to construct the axis information of the output data. However, a convolution operator should be applicable to any input data independent of the data's axis origin or number of samples. Wildcarding irrelevant axis descriptors in the instantiation of `inaxis` and `outaxis` prevents a compatibility check (see constructors above). The axis descriptors of the output data are created according to the `forwardwildtrans` and `adjointwildtrans` members of the convolution operator `fopContran`.

Except for the number of samples, both wildcard functions set the output axes information equal the input axes information. In the case of convolution (`forwardwildtrans`) with a filter of length n, the output data length increases by $n-1$. In the adjoint case the number of sample decreases by the same amount. The wildcarding functions supply axis values for the wildcarded descriptors of the output data. The axis information of the member `inaxis` facilitates the axis information for the descriptors which were not wildcarded.

The scientifically interesting member function of `fopcontran` is `apply`, which holds the code for convolution.

All CLOP operators which are derived from `fopone` inherit an `apply` routine which copies the input to the output. This `apply` is meant to be overwritten by an operator specific function. However, the name `apply` and its argument list are standardized in CLOP. The data, which the operator is applied to and which it computes, is referenced as `floatArray` and the corresponding axes information as `Axislist`. The integer argument `adj` designates if the operator or its adjoint is to be computed. Note that the axislists are defined as `const` to prevent `apply` from changing them accidentally.

In numerical analysis, a linear operator is often visualized as a matrix multiplication. However, in many application programs it is inefficient to implement the matrix explicitly. For example, convolution of a long trace with a short filter is equivalent to multiplication with an extremely sparse matrix (see equation (2.1) in the 'Standard Documentation' chapter). As an alternative to storing and multiplying a huge banded matrix, convolution routines usually store only the filter coefficients and apply them "on the fly".

Since private members of `fopcontran` are accessible everywhere within the same object, coefficients of the member `filter` can be freely referred to. The function `rows()` is invoked using the scope resolution operator `::`. It thereby refers to the globally defined M++ function (?) `rows` and ignores any possible `fopcontran` internal function called `rows`.

The syntax for arithmetic operations and array indexing in `apply` resembles Ratfor and is not standard C or C++ notation. This syntax is the main contribution of the

M++ classes to CLOP.

2.5 SKELETON

CLOP is designed to avoid the duplication of code. The code implementing a new operator is exclusively concerned with characteristics which set the operator apart from the base class. In general, each operator possesses a distinct `apply` method and appropriate constructors. So what can we learn from this example chapter? This example chapter outlines the overall structure common to the implementation of every CLOP operator. Additionally, many details may familiarize the reader with the tools available in the CLOP environment.

2.6 STANDARD DOCUMENTATION

Many users may want to apply a CLOP operator without being burdened by unnecessary information. What does such a user need to know in order to use the convolution operator class? A user should be told about the physical meaning of the operator, how the operator was implemented in the `apply` method of the operator class, and how he can construct an object of the convolution class `fopContran`.

The first subsection of this chapter is a brief physical introduction to the operator. That section also includes a meaningful example with a figure. I copied an appropriate paragraph about convolution from PVI (Claerbout, 1992).

The second subsection of this chapter will show you the `apply` method of the operator.

The last section lists the syntax of the different constructor expressions for the convolution operator. It seems useful to list the different arguments similar to SEP's traditional self-documentation.

2.6.1 The convolution operator

Matrix multiplication with a matrix of the form

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} b_1 & 0 & 0 & 0 & 0 \\ b_2 & b_1 & 0 & 0 & 0 \\ b_3 & b_2 & b_1 & 0 & 0 \\ 0 & b_3 & b_2 & b_1 & 0 \\ 0 & 0 & b_3 & b_2 & b_1 \\ 0 & 0 & 0 & b_3 & b_2 \\ 0 & 0 & 0 & 0 & b_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} \tag{2.1}$$

is called convolution. The operation $\mathbf{Bx}$ convolves b_t with x_t, whereas the operation $\mathbf{B}'\mathbf{y}$ crosscorrelates b_t with y_t.

The corresponding pseudo code is:

```
do ix = 1, nx {
do ib = 1, nb {
        iy = ib + ix - 1
        if operator itself (convolution)
                y(iy) = y(iy) + b(ib) × x(ix)
        if adjoint (correlation)
                x(ix) = x(ix) + b(ib) × y(iy)
}}
```

Notice that the "bottom line" in the program is that x and y are simply interchanged.

Equation (2.1) could be rewritten as

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} x_1 & 0 & 0 \\ x_2 & x_1 & 0 \\ x_3 & x_2 & x_1 \\ x_4 & x_3 & x_2 \\ x_5 & x_4 & x_3 \\ 0 & x_5 & x_4 \\ 0 & 0 & x_5 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{2.2}$$

which we abbreviate by $\mathbf{y} = \mathbf{X}\mathbf{b}$. So we can choose between $\mathbf{y} = \mathbf{X}\mathbf{b}$ and $\mathbf{y} = \mathbf{B}\mathbf{x}$. In one case the output $\mathbf{y}$ is dual to the filter $\mathbf{b}$, and in the other case the output $\mathbf{y}$ is dual to the input $\mathbf{x}$. In applications, sometimes we will solve for $\mathbf{b}$ and sometimes for $\mathbf{x}$; so sometimes we will use equation (2.2) and sometimes (2.1).

The class `fopcontran` (page 341) can be used for either equation (2.1) or equation (2.2), because the calling program can swap the $\mathbf{x}$ and $\mathbf{b}$ variables. The name `fopcontran()` denotes convolution with "transpose" and with "transient" end effects. Figure 2.1 shows a simple example for convolution using a short two point filter.

The convolution kernel

The convolution is implemented in `fopcontran.cc`'s `apply` member `fopcontran` (page 341).

```
void fopContran::apply(floatArray & x,floatArray & y,int adj,
                    const Axislist & xal, const Axislist & yal) const {
    int nb = (::rows(filter)); int nx = xal.list[0].length;
    for (int ib=0; ib < nb ; ib++) {
    for (int ix=0; ix < nx ; ix++) {
        if (adj)
           x(ix)    += y(ib+ix) * filter(ib);
        else
           y(ib+ix) += x(ix)    * filter(ib);
}   }}
```

The constructors

The header `fopcontran.h` (page 342) defines the constructors.

```
class fopContran : public fopone {
  public:
       // construct operator from an array; apply to axis axisname;
       // sample interval of input data must be delta;
    fopContran( floatArray & filt,const char* axisname, float delta );
       // sample interval of input data is irrelevant;
    fopContran( floatArray & filt,const char* axisname );
       // construct operator from floatspace;
       // if usedelta==0 then the sample rate of data is irrelevant;
    fopContran(const floatspace & filtsp, int usedelta = 1);
  }
```

The different arguments in the various constructors refer to:

`filt` the filter coefficients

`axisname` the axis the 1D filter is applied to

`delta` the sample rate the input data has to match

`usedelta` the flag indicating if the sample rate of the filter has to be matched by the input data.

REFERENCES

Claerbout, J. F., and Karrenbach, M., 1993, How to use cake with interactive documents: SEP–**77**, 427–444.

Claerbout, J. F., 1992, Earth Soundings Analysis: Processing versus Inversion: Blackwell Scientific Publications.

Dellinger, J., and Tálas, S., 1992, A tour of SEPlib for new users: SEP–**73**, 461–502.

Dulac, J. C., and Nichols, D., 1989, Object oriented programming for seismic data: SEP–**61**, 303–318.

Chapter 3

Elementary Science Operators

Many calculations we do in Geophysics, as well as in other disciplines of science and engineering, are linear operations and can be expressed as matrix multiplications. Many practitioners do not think of these calculations as linear operations or matrix multiplications, since these operators are usually derived and programmed as efficient special purpose programs, instead as a general matrix multiplication.

In this chapter, we show the object oriented implementation of some elementary operators and their adjoints. We start by discussing the obviously linear matrix multiplication operator and consequently look at increasingly more complex operators.

matrix multiply	conjugate-transpose matrix multiply
zero padding	truncation
causal integration	anticausal integration
convolution	crosscorrelation
stretching (moveout)	squeezing

3.1 MATRIX MULTIPLICATION

[1] The prototype of a linear operation is matrix multiplication. Multiplication of a matrix $\mathbf{B}$ times a vector $\mathbf{x}$ is computed by $y_i = \sum_j b_{ij} x_j$. The adjoint operation to the multiplication of the matrix $\mathbf{B}$ is multiplication with $\mathbf{B}$'s transpose: $\tilde{x}_j = \sum_i b_{ij} y_i$.

fopMatmul::apply

```
void fopMatmul::apply( floatArray & x, floatArray & y, int adj,
                       const Axislist & xal, const  Axislist & yal) const {
    int m = ::rows(x);
    int n = ::rows(matrix);
    for( int i=0; i<n; i++ )
    for( int j=0; j<m; j++ )
       if (adj)
```

[1] Matthias Schwab

```
            x(j) +=  matrix(i,j) * y(i);
         else
            y(i) +=  matrix(i,j) * x(j);
   }
```

The CLOP library class `fopMatmul` is used to build matrix multiplication operators. The `fopMatmul` class contains the `apply` method which codes the mathematical relations concerning the forward and adjoint of matrix multiplication. Note the interchange of the input and output variables in the bottom `if` statement: this symmetry is characteristic for forward and adjoint operator pairs. In general, each input variable `x(i)` which contributes to a certain output variable `y(j)` in the forward operation will receive an equally weighted contribution from this variable `y(j)` in the adjoint operation.

In order to carry out a matrix multiplication, a user first creates a `fopMatmul` operator by invoking one of the two constructors listed in the `fopmatmul` header file. Having two constructors allows users to use matrix multiplication whether they have named coordinate axes and labels or simply a matrix and a vector.

fopMatmul::header

```
class fopMatmul : public fopone {
  public:
    fopMatmul( const floatspace &mat );
    fopMatmul( const Axis &inaxis, const Axis &outaxis, const floatArray &mat);
  }
```

After creation of a matrix multiplication operator, the application program can apply the operator to any conformable vector by invoking the `forward()` method of `fopMatmul`. The function `fopMatmul.forward()` accepts the input vector as argument and returns the multiplied vector. Analogously, the application program can apply the adjoint operator by invoking the `adjoint()` method.

3.2 CAUSAL INTEGRATION

[2] Causal integration is defined as

$$y(t) = \int_{-\infty}^{t} x(t)\ dt \tag{3.1}$$

[2]Hector Urdaneta

Sampling the time axis gives a matrix equation which we should call causal summation, but we often call it causal integration.

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \end{bmatrix} \quad = \quad \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{bmatrix} \tag{3.2}$$

(In some applications the 1 on the diagonal is replaced by 1/2.) Causal integration is the simplest prototype of a recursive operator. The coding is trickier than the first derivative operator. Notice when you compute y_5 that it is the sum of 6 terms, but that this sum is more quickly computed as $y_5 = y_4 + x_5$. Thus equation (3.2) is more efficiently thought of as the recursion

$$y_t \quad = \quad y_{t-1} + x_t \qquad \text{for increasing } t \tag{3.3}$$

(which may also be regarded as a numerical representation of the differential equation $dy/dt = x$.)

When it comes time to think about the adjoint, however, it is easier to think of equation (3.2) than of (3.3). Let the matrix of equation (3.2) be called $\mathbf{C}$. Transposing to get $\mathbf{C}'$ and applying it to $\mathbf{y}$ gives us something back in the space of $\mathbf{x}$, namely $\tilde{\mathbf{x}} = \mathbf{C}'\mathbf{y}$. From it we see that the adjoint calculation, if done recursively, needs to be done backwards like

$$\tilde{x}_{t-1} \quad = \quad \tilde{x}_t + y_{t-1} \qquad \text{for decreasing } t \tag{3.4}$$

We can sum up by saying that the adjoint of causal integration is anticausal integration.

The header `fopcausint.h` (page 345) defines the causal integrator constructor.

```
class fopcausint : public fopone {
  public:

      // construct
    fopcausint( const Axis & causintaxis );

  }
```

The code for anticausal integration is not obvious from the code for integration and the adjoint coding tricks we learned earlier. To understand the adjoint, you need to inspect the detailed form of the expression $\tilde{\mathbf{x}} = \mathbf{C}'\mathbf{y}$ and take care to get the ends correct.

The class method `fopcausint::apply` `fopcausint` (page 346) shows the code for the forward and adjoint operation.

```
void fopcausint::apply(floatArray & x, floatArray & y, int adj,
                        const Axislist & xlist, const Axislist & ylist) const {

    int n = xlist.list[0].length;
    if (!adj) {
        y(0)   = x(0);
        for( int i=1;   i<n; i++) {   y(i) = y(i-1) + x(i); }
    } else {
        x(n-1) = y(n-1);
        for( int i=n-1; i>0; i--) { x(i-1) = x(i)   + y(i-1); }
    }
}
```

A beautiful example is shown in Figure 3.1.

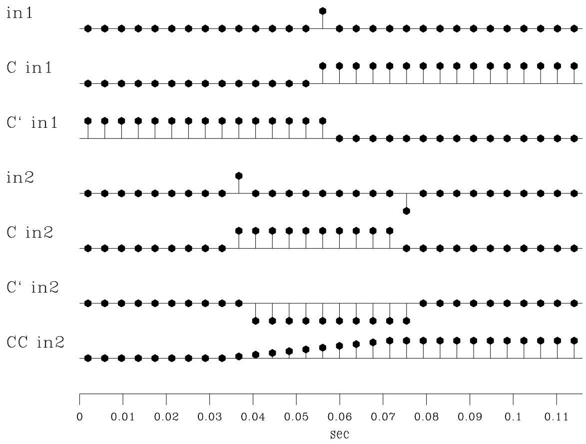

Figure 3.1: `in1` is an input pulse. `C in1` is its causal integral. `C' in1` is the anticausal integral of the pulse. `in2` is a separated doublet. Its causal integration is a box and its anticausal integration is the negative. `CC in2` is the double causal integral of `in2`. How can an equilateral triangle be built? elementary-cllp [R]

3.3 FIRST DERIVATIVE

[3] Given a sampled signal, its time derivative can be estimated by convolution with the filter $(1, -1)/\Delta t$. This example arises so frequently that I display the matrix multiply below:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} \tag{3.5}$$

The filter impulse response is seen in any column in the middle of the matrix, namely $(1, -1)$. In the transposed matrix the filter impulse response is time reversed

[3]Hector Urdaneta

to $(-1, 1)$. So, mathematically, we can say that the adjoint of the time derivative operation is the negative time derivative. This corresponds also to the fact that the complex conjugate of $-i\omega$ is $i\omega$. We can also speak of the adjoint of the boundary conditions: we might say the adjoint of "no boundary condition" is "specified value" boundary conditions.

An example of the first derivative operator is shown in Figure 3.2.

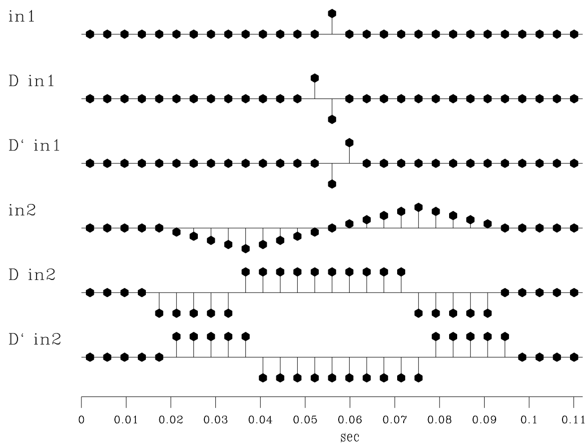

Figure 3.2: `in1` is an input pulse. `D in1` is its first derivative. `D' in1` is the adjoint of the first derivative operator applied of the pulse. `in2` is a double triangle function. Its first derivative `D in2` is made by three boxes and its adjoint `D' in2` is the negative. elementary-dllp [R]

The header file `fopdiff.h` (page 347) defines the `fopDiff()` class, where the first derivative operator is implemented. The class method `fopDiff::apply fopdiff` (page 347) performs the first derivative operation in either the forward or adjoint direction.

```
class fopDiff : public fopone {
  public:

      // construct
    fopDiff(const Axis& diffaxis );

  }
```

```
void fopDiff::apply(floatArray & x, floatArray & y, int adj,
                     const Axislist & xlist, const Axislist & ylist) const {

    for (int i=0; i<ylist.list[0].length; i++) {
        if (!adj) {
            y(i) = x(i+1) - x(i);
        } else {
            x(i+1) = x(i+1) + y(i);
            x(i  ) = x(i  ) - y(i);
        }
    }
}
```

3.4 CONVOLUTION

[4] In chapter 2 we discussed the implementation of convolution with transient end effects in great detail. However, we restricted the discussion to a straightforward convolution. In practise, filtering does often not consist of just convolving the data with a filter, but it involves shifting the data to some preferred time alignment, and truncating the output to the length of the input.

fopContrunc constructors

```
fopContrunc::fopContrunc( floatspace & x, int shift, int cut, int usedelta){
        Axislist ax = getinfo(x); num=3;
        ops    = new foperator*[num];
        ops[2] =                new fopContran( x,          usedelta ) ;
        ops[1] =                new fopAdvance( ax[0].name, shift    ) ;
        ops[0] = new fopAdjoint(new fopPad(     ax[0].name, 0, cut   ));
}
fopContrunc::fopContrunc(floatArray& x, int shift, int cut, const char* axname ){
        Axis ax(0,0,0,axname,1,1,1); num=3;
        ops    = new foperator*[num];
        ops[2] =                new fopContran( x, ax.name         ) ;
        ops[1] =                new fopAdvance(    ax.name, shift  ) ;
        ops[0] = new fopAdjoint(new fopPad(        ax.name, 0, cut ));
}
```

Chaining the convolution operator `fopContran` (see chapter 2) and the shift operator `fopAdvance` with the padding operator `fopPad` (see chapter 4), CLOP achieves the desired, more compliant convolution. The operator class `fopChain`, which implements such concatenations of existing CLOP operators, reuses the tested code of the composite operators `fopContran`, `fopAdvance`, and `fopPad`. Note, that the `fopPad` operator truncates a data set if the parameter `cut` is negative. The `forward()` method of the `fopChain` operator calls the `forward()` functions of its composite operators. The `adjoint()` operator of the `fopChain` operator is calling the adjoint functions of the chained operators, but in the reversed order.

fopContrunc header

```
class fopContrunc : public fopChain {
  public:
        fopContrunc( floatspace & x, int shift,int cut,int usedelta);
        fopContrunc( floatArray & x, int shift,int cut,const char* axname );
};
```

There exists a multitude of variations of convolution operators. One potentially interesting class for filter design and spectral estimation, are convolution schemes which do not assume zero data beyond their given data interval. One way to design such a filter is to truncate the output to an interval smaller than the input signal. Such a filter is easily implemented by changing the truncation operator in the operator chain described above. An alternative filter which avoids a zero assumption outside of the given input interval is a “circular” filter.

[4] Matthias Schwab

Figure 3.3 shows a simple test of different convolution operators and their end effects.

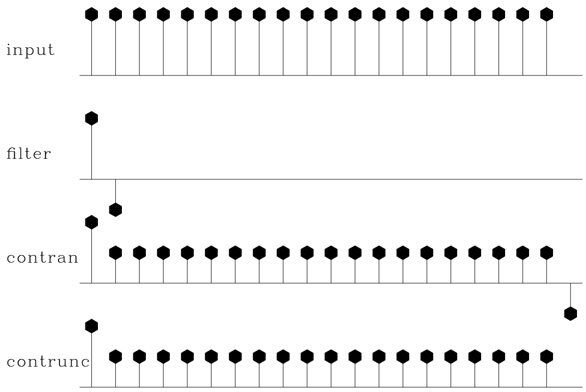

Figure 3.3: Demonstration of convolution. From top to bottom: (1) input; (2) filter; (3) output of `fopcontran()`; (4) output of `fopcontrunc()`; elementary-conv [R]

3.5 A GENERAL MOVEOUT OPERATOR

[5] Many basic geophysical processes have at their core a squeezing/stretching operation. The familiar processes **migration**, **CDP stack**, **velocity analysis** and **slant stack** all utilize such an operator. In this chapter we lay the foundation and design a basic trace squeezing/stretching class, from which more complicated operators in chapter 6 and 7 are derived. We illuminate the workings of such a general operator by applying it to a simple process, namely normal moveout (NMO). For example NMO stretches traces of CDP gathers according to a given velocity. In 2-D, each trace in a time-offset panel is transformed into an equivalent moveout corrected trace.

The general moveout class `fopMO` accomplishes this task. It is based on the notion that it operates on a panel that is specified by two axes and a moveout function is defined that maps each input trace into an output trace. The input panel will have a time coordinate t and a location x, the output trace will have a time coordinate τ and a location x, and the mapping function will be of the form $t(x) = t(\tau, x)$.

The moveout transformation for one trace $\mathbf{N}$ is representable as a square matrix (when the time axes are of the same length). The matrix $\mathbf{N}$ is a (τ, t)-plane containing all zeros except an interpolation operator centered along the trajectory defined by the mapping function. The dots in the matrix below are zeros. The input signal d_t is put into the vector $\mathbf{d}$. The output vector for this particular operator $\mathbf{m}$—i.e., the moved out signal—is simply $(d_6, d_6, d_6, d_7, d_7, d_8, d_8, d_9, d_{10}, 0)$. In real life examples

[5] Martin Karrenbach

the subscript goes up to about one thousand instead of merely to ten.

$$\mathbf{m} = \mathbf{N}\mathbf{d} = \begin{bmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \\ m_5 \\ m_6 \\ m_7 \\ m_8 \\ m_9 \\ m_{10} \end{bmatrix} = \begin{bmatrix} . & . & . & . & . & 1 & . & . & . & . \\ . & . & . & . & . & 1 & . & . & . & . \\ . & . & . & . & . & 1 & . & . & . & . \\ . & . & . & . & . & . & 1 & . & . & . \\ . & . & . & . & . & . & 1 & . & . & . \\ . & . & . & . & . & . & . & 1 & . & . \\ . & . & . & . & . & . & . & 1 & . & . \\ . & . & . & . & . & . & . & . & 1 & . \\ . & . & . & . & . & . & . & . & . & 1 \\ . & . & . & . & . & . & . & . & . & . \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \\ d_8 \\ d_9 \\ d_{10} \end{bmatrix} \tag{3.6}$$

You can think of the matrix as having a horizontal t-axis and a vertical τ-axis. The 1's in the matrix are arranged on the trajectory of the mapping function. $t = t(\tau)$.

The class definition for `fopMO` defines a constructor, an `apply()` function and two virtual functions, `mapping()` and `weight()`.

```
class fopMO : public fopone {
  public:

    // Constructors for fopMO: parameters are:
    // Time axis,Offset axis, slowness
    // or slowness array with as many samples as the time axis.
    fopMO(Axis & t ,Axis & x );
    fopMO(Axis & ti ,Axis & xi, Axis & to ,Axis & xo );

    ~fopMO();

    // define moveout curve
    virtual float mapping( float tau, float x ) const ;

    // define weighting function for data points
    virtual float weight ( float tau, float x ) const ;

    void apply(floatArray &,floatArray &,int,
                const Axislist &,const Axislist &) const;

        Axislist inlist;
        Axislist outlist;
};
```

The heart of the stretching operation is implemented in `fopMO:apply()`. The code loops over the second input axis placing each sample in the input trace value into the appropriate output sample.

```
void fopMO::apply(floatArray & in,floatArray & out,int adj,
                  const Axislist & inlist,const Axislist & outlist) const {
    for( int  i=0;  i<inlist[1].length; i++)  { float x   = inlist[1].value(i);
    for( int iz=0; iz<inlist[0].length; iz++) { float tau = inlist[0].value(iz);

        int it = outlist[0].index( mapping(tau,x) );
        if (it < outlist[0].length && it >= 0) {
           if ( adj==0 ){ out(it,i) +=  in(iz,i) * weight(tau,x);} // modeling
           else         {  in(iz,i) += out(it,i) * weight(tau,x);} // processing
        }
    }}
}
```

The computation of the location is specified by a virtual member function `mapping()`. This function is redefined in a specific derived operator. The `mapping` function returns a floating point number which must be converted into an index in the input trace. This is achieved by using the `index()` member function of the `Axis` class (see man page 413).

3.5.1 The mapping function

The basic stretching operator allows specification of a mapping function `fopMO::mapping()`. The mapping functions takes an input sample location (τ, x) as arguments and computes the output location in time (t) for the same x coordinate. Thus it is a one-dimensional mapping between input and output space. The mapping function is a virtual function, that can be overwritten by derived operators. In chapter 6 and 7 we extensively use this feature to specify a variety of mapping functions, but still rely on all other properties of the basic class.

```
float fopMO::mapping(float tau, float x) const {
   float  t = tau;
   return t;
}
```

The mapping function can be as complicated as you would like. By default, the `fopMO::mapping` is the identity matrix and will map a point in the input to the same location in the output, thus basically copying the input space onto the output space.

3.5.2 Weighting during the mapping

The basic stretching operator also allows specificaton of a weighting function `fopMO::weight()`. When the mapping is performed in the operator `fopMO` a weight can be multiplied onto the mapped data points. Weighting functions can be used to make an operator more unitary, or implement cut-off regions. If weights are applied the 1's in the matrix representation described above would become some time and offset dependent weight function. By default the `fopMO` operator will apply a weight of 1 to all data points (meaning essentially no weight is applied at all.)

```
float fopMO::weight(float tau, float x) const {
   float  dataweight = 1.;
   return dataweight;
}
```

Any derived class can override the virtual `weight()` function to implement its own desired weighting. Weighting functions can be used to specify data cut-offs like applying mute functions or aperture contraints to the data. Such applications are of considerable practical importance.

3.5.3 The fopMAP class

The class `fopMAP` (`fopmap.h` (page 352)) is derived from `fopMO`. It has the additional property that it stores internally some data required to calculate the mapping function. The data stored is an array of slowness values, one for each output time sample. The constructors for `fopMAP` allow the user to supply either a single slowness value or an array of different slowness values.

The `fopMAP` class does not define a mapping function, this means that it inherits `fopMO`'s mapping function which is usually redefined in the derived classes. Each class defines a mapping function that describes how to use the slowness data to calculate a coordinate transformation.

```
class fopMAP : public fopMO {
  public:
    fopMAP(Axis & t, Axis & x, float slow);
    fopMAP(Axis & t, Axis & x, float *slowarray);

    fopMAP(Axis & ti, Axis & xi, Axis & to, Axis & xo, float slow);
    fopMAP(Axis & ti, Axis & xi, Axis & to, Axis & xo, float *slowarray);

    float *slow;                /* mapping knows about slowness */

};
```

Many moveout operators can be derived from `fopMAP` using this interface. As a matter of fact, **all** operators in chapter 6 and 7 are derived from the basic class `fopMAP`. We do not show here an application of the basic operators `fopMO` and `fopMAP`, but rather point you to those chapters full of applications. In chapter 6 derived operators are described, that implement **normal moveout, linear moveout, kirchhoff moveout** and **anelliptic moveout**. Those operators are derived from `fopMAP` and merely specify their own new mapping function. Similarly the more comlicated composite operators, **velocity analysis**, **slant stack** and **Kirchhoff migration**, in chapter 7 are based on the derived operators introduced in chapter 6. We have a good reason for having a description of such general operators in this chapter. In particular we want to show you how careful design of a few basic operators avoids code duplication and leads to a simplified implementation. Thus it creates a foundation that allows rapid design and testing of new operators and new interpolation schemes.

Chapter 4

Base and Building Block Operators

[1] This chapter is an introduction to the basic framework and building blocks of operators. It serves as an overview for programmers intending to apply already existing operators or to combine them to form new operators. Using and understanding operators is discussed in chapter 2.

This document assumes a working knowledge of the C++ programming language, in particular the concepts of inheritance, class and construction.

4.1 BASIC OPERATOR HIERARCHY

Operators are arranged in the form of an inheritance hierarchy. This design eliminates the need to repeate code and simplifies the writing of new operators. At the top of the hierarchy are base classes which provide useful functions for derived operators and allow code to be written for abstract operators. New operators may be derived from any point in the hierarchy. Figure 4.1 depicts the dependencies between the operators discussed in this chapter.

For an introduction to operators and their functions, refer to the section on operators in chapter 1.

4.1.1 Base class foperator

At the top of the hierarchy is the `foperator`. It provides abstract virtual definitions of functions `Forward` and `Adjoint` operating from `floatspacearray` to `floatspacearray`. Derived classes must define these functions.

`foperator` provides functions which allow access to the `floatArray` and `Axislist` of a `floatspace`, and the `rows` and `cols` of a `floatspacearray`.

[1]Lisa Laane

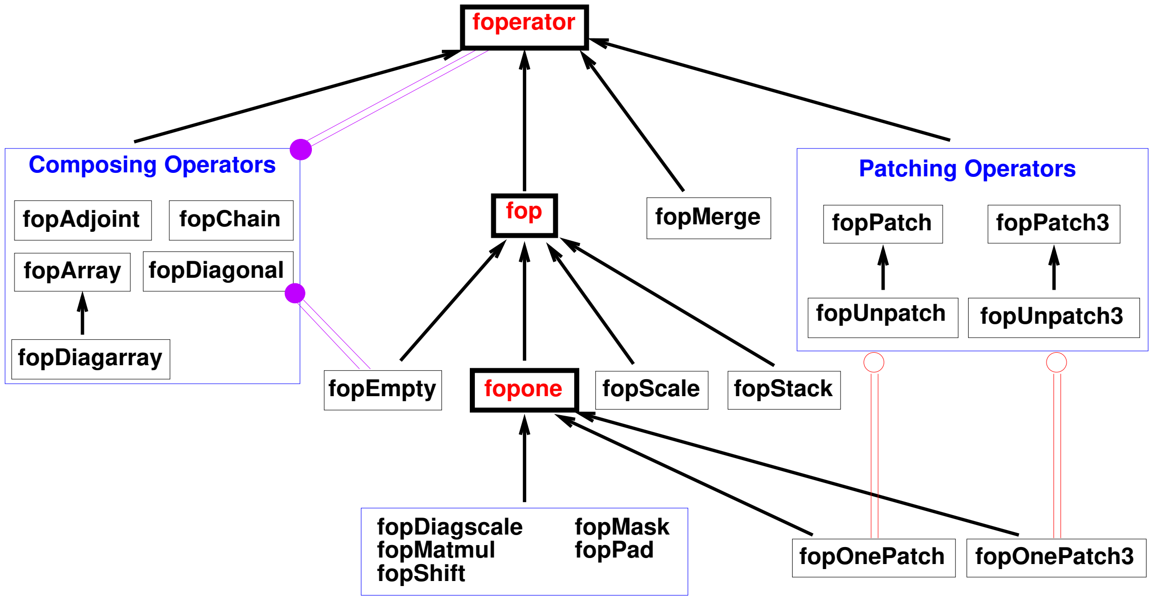

Figure 4.1: Graph of dependencies. Class relationships are depicted according to the Booch Method: a derived class points to its parent class with a single line and arrow, a hollow dot and double lines connect a class with the class it uses, and a solid dot and double lines connect a class with the class it contains. (basics-graph) [NR]

The `foperator` base class is also responsible for keeping track of the number of references to a specific `foperator`. This is hidden from the user for most applications and may usually be ignored. It is most important to note that `deref()` should be used for `foperators` instead of `delete`. See man pages `operator` p. 432.

See `foperator.h` below.

```
class foperator {
 public:
  foperator() {references=1;}

  virtual floatspacearray Forward(const floatspacearray &) const =0;
  virtual floatspacearray Adjoint(const floatspacearray &) const =0;

  virtual ~foperator() {references--;}

  void ref() {references++;}
  void deref() {references--;
                if (!(references)) delete this;}

  protected:
    int rows(const floatspacearray & x) const { return x.rows; }
    int cols(const floatspacearray & x) const { return x.cols; }

    floatArray & getdata(floatspace & s) const { return s.array; }
    const floatArray & getdata(const floatspace & s) const { return s.array; }
    Axislist & getinfo(floatspace & s) const { return s.axislist; }
    const Axislist & getinfo(const floatspace & s) const { return s.axislist; }

 }
```

4.1.2 Base class fop

Directly derived from `foperator` is class `fop`. `fop` defines new virtual functions for `Forward` and `Adjoint`, this time from a single `floatspace` to a single `floatspace`. It serves as the base class for operators which operate between `floatspaces`. Any derived class must define `Forward` and `Adjoint` between spaces, as well as a constructor.

`Forward` and `Adjoint` from `floatspacearray` to `floatspacearray` are defined to cycle through the `floatspaces` of the input `floatspacearray` and return the result of applying the operator to each `floatspace` individually. See man pages op p. 421.

Many simple operators are derived from `fop`, but `fopone` (described in the next section) provides additional helper functions which usually make it the preferred class for derivation.

See `fop.h` below.

```
class fop : public foperator {

  public:
    floatspacearray Forward(const floatspacearray &) const;
    floatspacearray Adjoint(const floatspacearray &) const;

    virtual floatspace Forward(const floatspace &) const = 0;
    virtual floatspace Adjoint(const floatspace &) const = 0;

};
```

4.1.3 Base class fopone

The majority of operators are derived from `fopone`, which is derived from `fop`. `fopone` includes several functions to help handle the work shared by most operators. When called `Forward` or `Adjoint`, it creates the output space, arranges the input space (through transposes) so that the axes are in the right order for calculating, and cycles over any extraneous axes. For derived operators, the user does not rewrite the `Forward` and `Adjoint` operators. Instead, an `apply()` function is written that is passed the input and output arrays. For more detail on the functions used by `fopone` see man pages opone p. 438 and chapter 2 on using and understanding an operator. See `fopone.h` below.

```
class fopone : public fop {
  public:
    fopone();

    // Construct fopone given input Axislist and
    // output Axislist.
    fopone( const Axislist &, const Axislist & );

    floatspace Forward( const floatspace & ) const;
    floatspace Adjoint( const floatspace & ) const;

    floatspacearray Forward( const floatspacearray & x ) const
      { return fop::Forward( x ); }
```

```
    floatspacearray Adjoint( const floatspacearray & x ) const
      { return fop::Adjoint( x ); }

    virtual Axislist forwardinfo( const Axislist & ) const;
    virtual Axislist adjointinfo( const Axislist & ) const;

  }
```

4.2 COMPOSING OPERATORS

Several operators derived from `foperator` are available to compose other operators and apply them from `floatspacearray` to `floatspacearray`. Using composing operators allows users to combine and easily create more powerful and complex operators.

Since the composing operators are `foperator`s themselves, they may be used in any combination with themselves and each other to create arrays of arrays, adjoints of chains, arrays of chains of chains, and so on.

It should be noted that the composing operators can assign an operator to be contained from either an operator or a pointer to one. In most cases, the operator to be contained can simply be passed, but when it may go out of scope *before* the composing operator, it should be constructed by calling `new` on a pointer to it, and the *pointer* should be passed. This case is likely to arise when composing operators are constructed within other operators and not in the main routine. See the man pages for `opAdjoint` p. 422, `fopChain` p. 425, and `fopArray` p. 423 for more information. See also the introduction to chains, adjoints and arrays in chapter 1.

4.2.1 The Adjoint operator

The simplest composing operator is `fopAdjoint`. When applied `Forward`, it will call the `Adjoint` of the `foperator` from which it is built. When applied `Adjoint`, it will perform the `Forward` of the original `foperator`.

The constructor takes the operator or a pointer to the operator of which the adjoint operator is desired. `fopAdjoint` can also be created as follows:

```
fopAdjoint A = !B;
```

where **A** is the the adjoint of `foperator` **B**. See `fopadjoint.h` below.

```
class fopAdjoint : public foperator {
  public:
    fopAdjoint( foperator &);
    fopAdjoint( foperator *);

    fopAdjoint & operator=(const fopAdjoint &);

    friend fopAdjoint operator!(foperator &);

    ~fopAdjoint();
```

```
    floatspacearray Forward(const floatspacearray &) const;
    floatspacearray Adjoint(const floatspacearray &) const;
  }
```

4.2.2 Chains of operators

`fopChain` is used to join two `foperators` to be applied in sequence to a `floatspacearray`. When applied `Forward`, `fopChain` applies the second `foperator` and then the first. When applied `Adjoint`, the operators are applied in the opposite order and adjoint. Derived classes may contain more than two `foperators` in the chain.

Chains are constructed from the two `foperators` or pointers to them. Chains can also be created by using the following syntax:

```
fopChain C = A * B;
```

where **A** and **B** are `foperators` and **C** is a chain of **A** acting on **B**. When using this syntax, any number of `foperators` may be joined using `*`.

See `fopchain.h` below.

```
class fopChain : public foperator {
 public:
    fopChain() {};
    fopChain(const fopChain &);
    fopChain(foperator &,foperator &);
    fopChain(foperator *, foperator *);

    fopChain & operator=(const fopChain &);
    friend fopChain operator*(foperator &, foperator &);

    ~fopChain();

    floatspacearray Forward(const floatspacearray &) const;
    floatspacearray Adjoint(const floatspacearray &) const;
  }
```

4.2.3 Arrays of operators

`fopArray` is used to create two-dimensional arrays of `foperators` to be applied to `floatspacearrays` (which are also up to two-dimensional) in a fashion resembling matrix multiply. For `Forward`, the rows of the `fopArray` are multiplied by the columns of the `floatspacearray`, and this is summed along. `Adjoint` does the same, except the transpose of the `fopArray` is used.

`fopArrays` are constructed by specifying the number of rows and columns in the array. An individual `foperator` can then be assigned with the `set` function. `set` takes the coordinates of the position in the array and either a `foperator` or a pointer to one. Every position in the array must be assigned an operator. Individual operators may be viewed or applied (but not changed) using an indexing operator.

See `foparray.h` below.

```
class fopArray : public foperator {
  public:
    fopArray() {rows=0; cols=0; ops=0;}
    fopArray(const fopArray &);
    fopArray & operator=(const fopArray &);
    ~fopArray();

    fopArray(int i, int j=1);

    // index a specific operator; cannot change return value
    const foperator & operator() (int i, int j=1) const;

    // set to operator
    void set(int, int, foperator &);
    void set(int, int, foperator *);

    void setrow( int, foperator &);
    void setrow( int, foperator *);

    void setcol( int, foperator &);
    void setcol( int, foperator *);

    floatspacearray Forward(const floatspacearray &) const;
    floatspacearray Adjoint(const floatspacearray &) const;
  }
```

Diagonal arrays of operators

For convenience, class `fopDiagarray` is provided to create diagonal arrays of `foperators`.

Derived from `fopArray` it provides the same functions as its parent class. It is constructed from the length of a side of the square array and a `foperator` to be assigned to the diagonal positions. All other positions are set to the "zero" operator, `fopEmpty` (see the discussion of `fopEmpty` in Section 4.3.2). Any position of the array can be reassigned to another `foperator`.

One should note that this is *not* an optimized version of `fopArray`, and only serves as an aid to easily set the diagonal positions to the same operator and all other positions to "zero". See the following section on `fopDiagonal`. See `fopdiagarray.h` below.

```
class fopDiagarray : public fopArray {
 public:
  fopDiagarray(int, foperator &);
  fopDiagarray(int, foperator *);
};
```

4.2.4 Optimized diagonal arrays of operators

The operator `fopDiagonal` is the optimized version of the diagonal array of operators. It behaves like a `fopArray`, but is restricted to contain non-"zero" operators *only* along the diagonal.

fopDiagonal operators can be constructed from the number of rows (or columns) of the array, or that number and an operator to insert in every position in the diagonal. The class provides functions to index and set the diagonal positions of the array by specifying the row number (which is the same as the column number). See fopdiagonal.h below.

```
class fopDiagonal : public foperator {
  public:
    fopDiagonal( int );
    fopDiagonal( int, foperator & op );
    fopDiagonal( int, foperator * op );
    fopDiagonal( const fopDiagonal &);

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

    void set( int, foperator & op );
    void set( int, foperator * op );
    const foperator & operator()( int ) const;

    floatspacearray Forward( const floatspacearray & ) const;
    floatspacearray Adjoint( const floatspacearray & ) const;
  }
```

4.3 UTILITY OPERATORS

The following is a list of simple operators that are basic to many applications and can easily be used by programmers.

More complete descriptions are provided in the chapter 9 Appendix man pages.

4.3.1 Operators derived from foperator

Any operator derived from `foperator` defines `Forward` and `Adjoint` from `floatspacearray` to `floatspacearray`.

```
floatspacearray Forward( const & floatspacearray ) const
floatspacearray Adjoint( const & floatspacearray ) const
```

The merge operator

The `Forward` operator merges a vector of `floatspaces` (in the form of a `floatspacearray`) into a `floatspace` (returned as a 1x1 `floatspacearray`) that has one more dimension than the individual spaces in the input. Note that the individual spaces *must* have the same dimensions. The `Adjoint` operator slices a `floatspacearray` containing a single `floatspace` at every sample along the given `Axis` to form a vector `floatspacearray` of spaces with one fewer dimension than the input space.

The constructor takes an `Axis` that describes the dimension to be created or eliminated. See `fopmerge.h` below.

```
class fopMerge : public foperator {

    public:
        fopMerge(const Axis &);

        floatspacearray Forward(const floatspacearray &) const;
        floatspacearray Adjoint(const floatspacearray &) const;

    }
```

Figure 4.2 shows a `floatspacearray` of four `floatspaces` (each of two dimensions) being merged into a single three-dimensional `floatspace`.

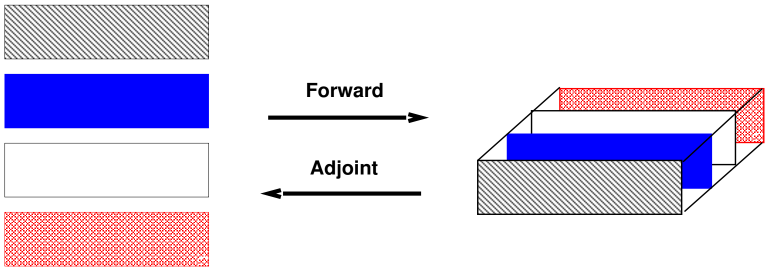

Figure 4.2: Left frame = input floatspacearray of 2-D spaces; Right frame = merged 3-D floatspace (basics-merge) [NR]

The patch operator

The operator `fopPatch` can chop a space into subspaces (which may overlap) called patches. The `Adjoint` adds them back again. Because of possible overlap, however, weighting functions are needed to restore the original space (see the following section on the unpatch operator).

`Forward` takes a 1x1 `floatspacearray` and returns a one-dimensional `floatspacearray` of patches. `Adjoint` takes a one-dimensional `floatspacearray` of patches and returns a 1x1 `floatspacearray` of the patches put together into a single space. `fopPatch` uses `fopOnepatch` for each individual patch that is extracted from or inserted in the larger space. See the description of `fopOnepatch` in Section 4.3.3.

`fopPatch` is constructed from the number and size of patches along each `Axis`, and the `Axislist` of the single large space in the intput `floatspacearray`. See `foppatch.h` below.

`fopPatch` operates on `floatspacearrays` containing two-dimensional `floatspaces` and patches. `fopPatch3` is the equivalent operator for three-dimensional patching. See `foppatch3.h` below.

```
class fopPatch : public foperator {
    friend class fopUnpatch;
  public:
    fopPatch();
    fopPatch(const fopPatch &);
    fopPatch( int num0,int num1,int size0,int size1,const Axislist& );
```

```
    floatspacearray Forward( const floatspacearray & ) const;
    floatspacearray Adjoint( const floatspacearray & ) const;
  }


class fopPatch3 : public foperator {
    friend class fopUnpatch3;
  public:
    fopPatch3();
    fopPatch3(const fopPatch3 &);
    fopPatch3( int, int, int, int, int, int, const Axislist & );

    floatspacearray Forward( const floatspacearray & ) const;
    floatspacearray Adjoint( const floatspacearray & ) const;
  }
```

The unpatch operator

Since patches may overlap, the `Adjoint` of `fopPatch` is not the same as the inverse. `fopUnpatch` is derived from `fopPatch` to serve as the inverse. It uses weighting functions so that the `Adjoint` of `fopUnpatch` is the inverse of `Forward` of `fopPatch`, and vice versa. Note that the `Adjoint` is the preferred method for reassembling patches.

`fopUnpatch` is based on the following equation:

$$\tilde{\mathbf{d}} = [\mathbf{W}_{wall}\ \mathbf{P}'\ \mathbf{W}_{wind}\ \mathbf{P}\ ]\mathbf{d} \tag{4.1}$$

where $\mathbf{d}$ is the original data, $\tilde{\mathbf{d}}$ is the reconstructed data, $\mathbf{P}$ is the `Forward` of `fopPatch`, and $\mathbf{P}'$ is the `Adjoint` of `fopPatch`. $\mathbf{W}_{wind}$ is an arbitrary weighting function for a window (patch). $\mathbf{W}_{wall}$ is the weighting function for the wall (full space) that is calculated from $\mathbf{W}_{wind}$ to undo the effects of overlapping. Since $\mathbf{W}_{wall}$ is constructed from $\mathbf{W}_{wind}$, *any* weighting function may be chosen for $\mathbf{W}_{wind}$.

`Adjoint` for `fopUnpatch` applies the following to a `floatspacearray` of patches:

$$\mathbf{W}_{wall}\ \mathbf{P}'\ \mathbf{W}_{wind}$$

and `Forward` applies the following to a `floatspacearray` containing a single space:

$$\mathbf{W}_{wind}\ \mathbf{P}\ \mathbf{W}_{wall}$$

`fopUnpatch` may be constructed the same way as `fopPatch`. It may also be constructed from an object of type `fopPatch`. See `fopunpatch.h` below.

`fopUnpatch3` is the equivalent operator for three-dimensional unpatching. See `fopunpatch3.h` below.

```
class fopUnpatch : public fopPatch {
  public:
    fopUnpatch( const fopPatch & );
    fopUnpatch( int, int, int, int, const Axislist & );
```

```
    floatspacearray Adjoint( const floatspacearray & ) const;
    floatspacearray Forward( const floatspacearray & ) const;
 }
```

```
class fopUnpatch3 : public fopPatch3 {
  public:
    fopUnpatch3( const fopPatch3 & );
    fopUnpatch3( int, int, int, int, int, int, const Axislist & );
    floatspacearray Adjoint( const floatspacearray & ) const;
    floatspacearray Forward( const floatspacearray & ) const;
  }
```

The following exmple creates and applies an operator, **op**. It first makes a 9x1 `floatspacearray` of patches of size 4x7 each. It then applies a 9x9 diagonal array of scaling operators to scale the 4th patch by 3.0 and all others by 0.5. Finally, it reassembles the patches (with weighting).

```
fopPatch myPatch( 3, 3, 4, 7, mySpaceAxisList );
fopScale myScale(0.5);
fopScale myScale2(3.0);
fopDiagonal myDiagonal( 3*3 , myScale );
myDiagonal.set( 4, myScale2 );
fopUnpatch myUnpatch( myPatch );

fopChain op = !myUnpatch * myDiagonal * myPatch;

floatspacearray result = op.Forward( mySpacearray );
```

Figure 4.3 depicts the single `floatspace` in the `floatspacearray` before and after application of the operator.

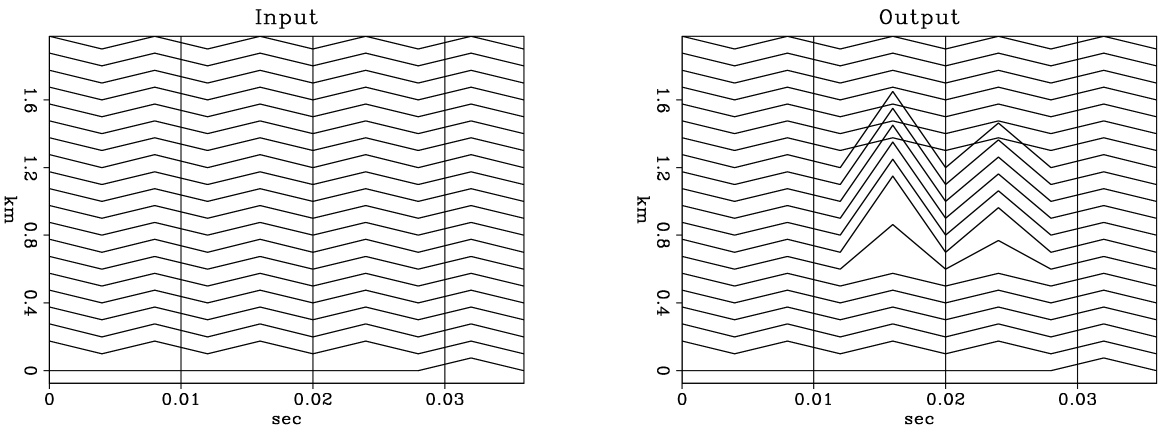

Figure 4.3: Left frame = input floatspace, right frame = output floatspace basics-patchout [R]

4.3.2 Operators derived from fop

`Forward` and `Adjoint` for operators derived from `fop` operate between single `floatspaces`. When called on a `floatspacearray` they cycle through all the `floatspaces` of the array.

```
floatspace Forward( const & floatspace ) const
floatspace Adjoint( const & floatspace ) const

floatspacearray Forward( const & floatspacearray ) const
floatspacearray Adjoint( const & floatspacearray ) const
```

The scale operator

`fopScale` multiplies a `floatspace` by a single float.

It is constructed from the float by which the space is to be multiplied. `Forward` multiplies by the float, and `Adjoint` also multiplies by the complex conjugate of the float. See `fopscale.h` below.

```
class fopScale : public fop {
  public:
    // Create a scale operator that multiplies by the argument
    fopScale(float);

    floatspace Forward(const floatspace &) const;
    floatspace Adjoint(const floatspace &) const;

    floatspacearray Forward(const floatspacearray & x) const
      { return fop::Forward(x); }
    floatspacearray Adjoint(const floatspacearray & x) const
      { return fop::Adjoint(x); }

  }
```

The “empty” operator

`fopEmpty` serves as a “zero” operator. When applied to a `floatspace` either `Forward` or `Adjoint`, it returns a “zero” `floatspace`, which acts like a “zero” when added to, subtracted from, or multiplied by another `floatspace`. See man pages `space` p. 464 and `Axis` p. 413.

`fopEmpty` has no constructor and is created by a simple declaration. See `fopempty.h` below.

```
class fopEmpty : public fop {
 public:
    floatspace Forward(const floatspace &) const;
    floatspace Adjoint(const floatspace &) const;

    floatspacearray Forward(const floatspacearray & x) const
      { return fop::Forward(x); }
    floatspacearray Adjoint(const floatspacearray & x) const
      { return fop::Adjoint(x); }
```

```
};
```

The stack operator

`fopStack` applies a reduction or expansion along one axis. When applied `Forward`, it adds an extra dimension to the space, and 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 optionally 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. See `fopstack.h` below.

```
class fopStack : public fop {
  public:
    // supply axis to be stacked over
    // and optionally the position to insert it
    fopStack(const Axis & inax , int pos = -1 );

    floatspacearray Forward(const floatspacearray & x) const;
    floatspacearray Adjoint(const floatspacearray & x) const;

  }
```

Figure 4.4 shows a two-dimensional `floatspace` being spread into a three-dimensional `floatspace` in the `Forward` direction. The `Adjoint` operator sums over the third axis to create a two-dimensional `floatspace`.

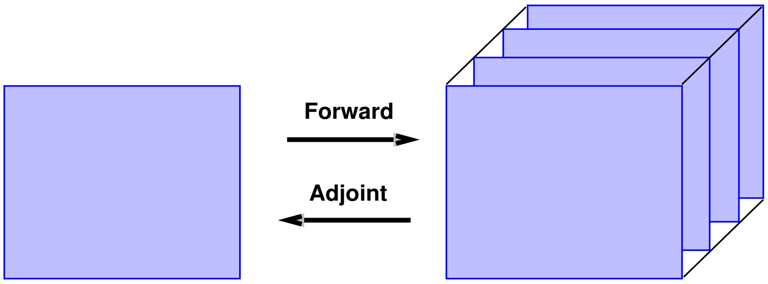

Figure 4.4: Left frame = input 2-D floatspace, right frame = replicated 3-D floatspace (basics-stack) [NR]

4.3.3 Operators derived from fopone

Since `fopone` is derived from `fop`, `fopone` operators can apply `Forward` and `Adjoint` from `floatspace` to `floatspace` or from `floatspacearray` to `floatspacearray`. If applied to a `floatspacearray`, they will apply the operator to each space in the array seperately.

The diagonal scale operator

`fopDiagscale` scales each element of a `floatspace` by a constant.

It is constructed either from a single `floatspace` of constants or two `floatspaces`, `sp1` and `sp2` of constants. In the second case, the input `floatspace` is multiplied by the square root of `sp1/sp2`. See `fopdiagscale.h` below.

```
class fopDiagScale : public fopone {
  public:
    // Construct a fopDiagScale that scales each
    // element of a space by a constant.
    fopDiagScale( const floatspace & scalars );
    fopDiagScale( floatspace & sp1, floatspace & sp2 );

  }
```

The mask operator

`fopMask` applies a mask to a `floatspace`, meaning it sets some of the elements to zero.

The constructors take the axis to mask along and the positions along this axis to set to zero. `Forward` and `Adjoint` perform the same function. See `fopmask.h` below.

```
class fopMask : public fopone {
  public:
    // Construct from Axis to apply to,
    // and an array of int, used as a mask.
    fopMask(const Axis &,const int *);

    // Construct from the Axis and a floatspace used
    // to mask.
    fopMask(const Axis &,const floatspace & x);
    ~fopMask();
  }
```

The one patch operator

`fopOnepatch` creates a single two-dimensional patch from a two-dimensional `floatspace` when applied `Forward`. When applied `Adjoint` it puts a patch in a larger `floatspace` that contains zeros elsewhere.

`fopOnepatch3` does the same for three dimensions.

One patches are constructed in many ways, always with specifications for the full space first and a patch second. The specific constructors are described in `foponepatch` and `foponepatch3`.

```
class fopOnepatch : public fopone {
  public:
    // construct from axislist of the full space and patch origin
```

```
    // and length along first axis and second axis
    fopOnepatch( const Axislist & spacelist,
                 float origin0, int len0,
                 float origin1, int len1 );

    // construct from origin and
    // length along each axis of full space
    // and axislist of the patch
    fopOnepatch( float origin0, int len0, float origin1, int len1,
                 const Axislist & patchlist );

    // construct from full space axislist and patch corner coordinates
    fopOnepatch( const Axislist & spacelist,
                 float i0, float j0, float i1, float j1);

    // construct from full space and patch axislists
    fopOnepatch( const Axislist & spacelist, const Axislist & patchlist );

  }
```

```
class fopOnepatch3 : public fopone {
  public:
    // Construct from axislist of the  full  space  and  patch
    // origin and length along each axis.
    fopOnepatch3( const Axislist & spacelist,
                  float origin0, int len0,
                  float origin1, int len1,
                  float origin2, int len2 );

    // Construct from full space origin and length
    // along each axis
    // and axislist of the patch
    fopOnepatch3( float origin0, int len0,
                  float origin1, int len1,
                  float origin2, int len2,
                  const Axislist & patchlist );

    // construct from full space axislist and patch corner coordinates
    fopOnepatch3( const Axislist & spacelist,
                  float i0, float j0, float k0,
                  float i1, float j1, float k1 );

    // construct from full space and patch axislists
    fopOnepatch3( const Axislist& spacelist,const Axislist& patchlist );

  }
```

Figure 4.5 shows the application of `fopOnepatch` `Forward` and `Adjoint` to first make a patch from the original `floatspace` and then put the patch back in a `floatspace` of the original size.

See the discussion of patch and unpatch in Section 4.3.1.

The pad operator

`fopPad` pads or truncates along an `Axis` of the `floatspace`. Surrounding a dataset by zeros (zero padding) is adjoint to throwing away the extended data (truncation). Let us see why this is so. Set a signal in a vector $\mathbf{x}$, and then make a longer vector $\mathbf{y}$ by adding some zeros at the end of $\mathbf{x}$. This zero padding can be regarded as the

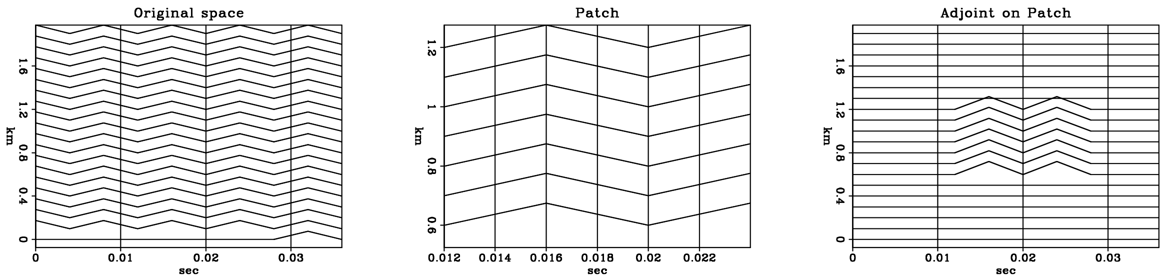

Figure 4.5: Left frame = original space, middle frame = fopOnepatch applied Forward on the original space. A single patch of size 5x6, right frame = fopOnepatch applied Adjoint on the patch basics-onepatchout [R]

matrix multiplication

$$\mathbf{y} \quad = \quad \begin{bmatrix} \mathbf{I} \\ \mathbf{0} \end{bmatrix} \mathbf{x} \tag{4.2}$$

The matrix is simply an identity matrix $\mathbf{I}$ above a zero matrix $\mathbf{0}$. To find the transpose to zero padding, we now transpose the matrix and do another matrix multiply:

$$\tilde{\mathbf{x}} \quad = \quad \begin{bmatrix} \mathbf{I} & \mathbf{0} \end{bmatrix} \mathbf{y} \tag{4.3}$$

So the transpose operation to zero padding data is simply *truncating* the data back to its original length.

The constructor for `fopPad` may either take the `Axis` to pad or its name, followed by the number of samples to pad to the front and to the back. For `Forward`, positive numbers specify the number to pad and negative numbers the number to truncate. `Adjoint` does the opposite.

See `foppad.h` below.

```
class fopPad : public fopone {
  public:
    fopPad(const Axis &, int, int);
    fopPad(const char *, int, int);

  }
```

The shift operator

`fopShift` shifts data in the `floatspace` along an `Axis`.

The constructors take either the `Axis` or the name of the `Axis` to shift along, and the distance by which to shift (positive values advance and negative values retard the origin). See `fopshift.h` below.

```
class fopShift : public fopone {
  public:
    fopShift(const Axis & inaxis,int shift);
    fopShift(const char* axname,int shift);
```

```
    }
```

¿

Chapter 5

Solving least-squares inverse problems

[1] The adjoint operator does not recover the input to the forward operator unless the operator happens to be unitary. If we want a better reconstruction than that given by the adjoint, we must use some form of inverse operator. Given a data vector $\mathbf{y}$ and an operator $\mathbf{A}$, we wish to find a model $\mathbf{x}$ such that $\mathbf{A}(\mathbf{x}) = \mathbf{y}$.

If the operator $\mathbf{A}$, is a matrix we could solve the problem in the following way,

$$\begin{aligned} \mathbf{Ax} &= \mathbf{y} \\ \mathbf{A}'\mathbf{Ax} &= \mathbf{A}'\mathbf{y} \\ \mathbf{x} &= (\mathbf{A}'\mathbf{A})^{-1}\mathbf{A}'\mathbf{y} \end{aligned}$$

If $\mathbf{A}$ is not represented as a matrix but it is implemented as an operator we can still solve this problem but we need another operator that is the "inverse" to $\mathbf{A}'\mathbf{A}$, i.e. for any $\mathbf{x}$ we require an operator $\mathbf{B}$ such that $\mathbf{B}(\mathbf{A}'(\mathbf{A}(\mathbf{x}))) = \mathbf{x}$ for all $\mathbf{x}$. The operator $\mathbf{B}$ can then be regarded as the inverse of $\mathbf{A}'\mathbf{A}$.

This assumes that the inverse of $(\mathbf{A}'\mathbf{A})$ exists. In many cases the matrix is singular or near enough to singular that this method will not work. We have to deal with problems that may be overdetermined, underdetermined and/or inconsistent. In these cases it is often best to choose to solve a different problem, the "least-squares" problem. We try to find a model that minimizes the L_2 norm of the difference between the predicted data and the observed data.

$$\min_{\mathbf{x}} ||\mathbf{Ax} - \mathbf{y}||^2$$

5.1 LEAST SQUARES SOLVERS

These concepts are implemented in solver objects. All solver objects are derived from the abstract base class `LSSolver`. The class header `lssolver.h` (page 370) shows the

[1] Dave Nichols

public definitions. An instance of this class (a solver object) is constructed from an operator that maps a model space to a data space.

```
class LSSolver {

public:
    LSSolver( foperator& op );
    virtual ~LSSolver();

    virtual floatspacearray Solve(  const floatspacearray & rhs )=0;

}
```

It has one member function, `Solve`. This function takes as its argument a `floatspacearray` that is the right hand side of the least-squares problem and returns a `floatspacearray` that is the solution to the least squares problem. It is a pure virtual function, this means that it is not defined in this class. A class that is derived from this base must implement the function.

5.1.1 Iterative solvers

In our work we have operators that have a forward and adjoint operation defined, so we are interested in solution methods that only require these operations. There is a whole class of algorithms that use iterative methods to approximate the solution to the least squares problem. Many iterative methods can be implemented using the following set of operations,

- Application of the forward operator to an instance of the model space.
- Application of the adjoint operator to an instance of the data space.
- Scaling and addition of members of the model space
- Scaling and addition of members of the data space
- Inner products of instances of the model space
- Inner products of instances of the data space

All of the operations are defined for *any* of our operator and space classes. We can write solver classes that use these algorithms and they will be usable with any operator and space in our class library.

The first class derived from `LSSolver` is `IterSolver`. This defines the class of solvers that solve a least-squares problem by an iterative algorithm. What distinguishes these solution methods is that they start from an initial estimate of the solution $\mathbf{x}_0$ and produce a sequence of solutions that are better approximations to the true solution (we hope).

The constructor for the iterative solver has an optional integer parameter that is the maximum number of iterations to perform. The class adds a new version of the

Solve function, one that takes both a right hand side and an initial solution. The Solve function defined in the LSSolver class is implemented by calling the new solve function with an initial solution that is all zeros.

The header itersolver.h (page 371) shows that there are many more public functions for this class. The most important is the new Solve function. Note that it too is a pure virtual function, this means that we must derive classes from IterSolver that actually implement the new function.

```
class IterSolver : public LSSolver {

public:

IterSolver(foperator & fop , int maxit=10 ) :
        LSSolver( fop ), save_x_iter(0), save_resid(0),
        maxiter(maxit), numiter(0) , repstream(nil){};

virtual floatspacearray Solve(const floatspacearray & rhs,
                              const floatspacearray& init) = 0; //pure virtual

floatspacearray Solve(  const floatspacearray & rhs );

void ReportOn( ostream & rep )                  // report on this stream
        {repstream = &rep; }

void SetSaveResidStep( int save_step );         // set residual save interval
void SetSaveIterStep( int save_step );          // set iteration save interval

int GetIterCount(){ return numiter; }           // number of iterations used

floatspacearray GetResiduals( );                // retrieve all residuals
floatspacearray GetOneResid( int i );           // retrieve one residual

floatspacearray GetIterations();                // retrieve all iterations
floatspacearray GetOneIter( int i );            // retrieve one iteration

~IterSolver();

}
```

The rest of the functions are utility functions common to all iterative solvers. They allow the user to save the sequence of iterations and/or residuals, to retrieve the intermediate results and to control reporting of the progress of the algorithm.

5.1.2 A conjugate gradient solver

The first "concrete" solver class (one with all the functions defined) is NormCGSolver. It is derived from IterSolver and it implements the Solve routine by using the "conjugate gradient" method on the normal equations associated with the least squares problem. This class implements the algorithm describe by Claerbout in PVI (Claerbout, 1992). Although this is not a numerically well behaved algorithm, it is implemented here to provide direct comparisons with solutions illustrated in PVI. The header file normcgsolver.h (page 372) shows that all that is defined is, a constructor, and the solve routine.

```
class NormCGSolver : public IterSolver {
public:
```

```
    NormCGSolver( foperator & fop, int maxit=10 ) : IterSolver( fop, maxit ){};

    floatspacearray Solve( const floatspacearray & rhs,
                           const floatspacearray & initial);

    floatspacearray Solve( const floatspacearray & rhs )
         { return IterSolver::Solve( rhs ); }

};
```

The implementation of the Solve routine is shown below. Note that at the end of each iteration it calls the functions `report()`, `save_one_iter()`, `save_one_resid()`, these functions are defined in the `IterSolver` class and are inherited by all classes derived from it.

```
#include <normcgsolver.h>

floatspacearray NormCGSolver::Solve( const floatspacearray & rhs,
                                    const floatspacearray & init  ) {
    floatspacearray x,grad,cgrad,resid;
    double alfa,beta,determ;
    double cgnorm,csnorm,cgxr,csxr,cgxcs;

    floatspacearray step =  init.empty();
    floatspacearray cstep = rhs.empty();

    rhs_norm = rhs.norm();
    x = init; resid = rhs - op->Forward(x);

    numiter = 0;
    do{
        grad = op->Adjoint(resid);
        cgrad = op->Forward(grad);
        if (numiter==0) {
            alfa = cgrad.dot(resid) / cgrad.norm2();
            beta = 0;
        } else {
            cgnorm = cgrad.norm2(); csnorm = cstep.norm2();
            cgxcs = cgrad.dot(cstep);
            cgxr = cgrad.dot(resid); csxr = cstep.dot(resid);
            determ = cgnorm * csnorm - cgxcs * cgxcs + 1.0E-15;
            alfa = (csnorm * cgxr - cgxcs * csxr) / determ;
            beta = (-cgxcs * cgxr + cgnorm * csxr) / determ;
        }
        step = alfa * grad + beta * step;
        cstep = alfa * cgrad + beta * cstep;
        x += step; resid -= cstep;

        // gather statistics, report, save iterations etc.
        resid_norm = resid.norm();
        report(); save_one_iter( x ); save_one_resid( resid );

    }while( (numiter++ < maxiter) && ( resid_norm/rhs_norm > 1e-8 ) );

    return x;
}
```

5.1.3 A different conjugate direction solver

The class `HestSolver` implements the algorithm describes by Hestenes (?). Again all we define is a constructor and a `Solve` function. The implementation of this algorithm is shown in hestsolver (page 373). This algorithm is said to be more stable than the simple conjugate-gradient method on normal equations described earlier.

```
#include <hestsolver.h>

floatspacearray HestSolver::Solve( const floatspacearray & rhs,
                                   const floatspacearray & init  ) {
    floatspacearray x,step,cstep,grad,resid;     // work space
    double gamma,gamma1,alpha,beta;

    rhs_norm = rhs.norm();                        // initial norm
    x = init; resid = rhs - op->Forward(x);      // initial residual

    grad = op->Adjoint(resid);
    step = grad;  gamma1 = step.norm2();

    numiter=0;
    do{
        cstep = op->Forward(step);
        alpha = gamma1 / cstep.norm2();
        x += step * alpha;
        cstep *= alpha; resid -= cstep;
        grad = op->Adjoint(resid);
        gamma = grad.norm2();
        beta = gamma / gamma1;
        gamma1 = gamma;
        step *= beta; step += grad;

        // gather statistics, report, save iterations etc.
        resid_norm = resid.norm();
        report(); save_one_iter( x ); save_one_resid( resid );

    }while( (numiter++ <maxiter) && ( resid_norm / rhs_norm > 1e-8 ) );

    return x;
}
```

These classes represent a first attempt to define least-squares solver objects. An inspection of the Solve function for the two classes shows that they have a lot of code in common. This usually means that the classes have been poorly designed and that some of this functionality belongs in the parent class. Another limitation is that it is difficult to fit non-linear solvers into this class heirarchy. These classes do not represent the final stages of our code development and they will probably change significantly before we are satisfied.

5.2 USING SOLVER OBJECTS

The program SolvNmo (page 374) illustrates the use of a solver object. The bulk of the code deals with reading and writing SEPlib data (our local data format). The core of the code constructs an NMO operator (op), it then constructs a solver from that operator (solver) and invokes the solver on the input data. The result of invoking the solver is written to the output. Note that the solver is instructed to report its progress on the stream associated with stderr.

```
/* SolvNmo vel=1.5 <infile.H >outfile.H
 * Applies inverse NMO operator to dataset, given velocity vel.
 */
#include <SEPinit.h>
```

```
#include <fopnmo.h>
#include <hestsolver.h>

main(int argc,char **argv) {
    SepInput * sepInput = SEPinit(argc,argv,SOURCE);// Seplib initialization

    int niter=5; sepInput->Getch("niter",niter);    // command line arguments
    float vel=1.5; sepInput->Getch("vel",vel);

    floatspace input(sepInput);                     // make input space
    Axislist inaxlist = input.getaxislist();        // retrive axis info

    fopNMO op(inaxlist[0],inaxlist[1],1/vel);       // make NMO operator

    HestSolver solver( op, niter );                 // construct solver object
    solver.ReportOn(cerr);                          // report progress on stderr
    floatspacearray output = solver.Solve( input ); // solve the problem

     // write to output
    SepOutput *sepout = new SepOutput("stdout", output(0,0).sepData() );
    delete sepout;

    UnRef(sepInput); return 0; // cleanup
}
```

This program produces results identical to those shown in the section on NMO and inversion 6.2.

5.3 PRECONDITIONED INVERSE PROBLEMS

Many iterative algorithms converge very slowly, an inversion for a 100×100 model has 10,000 free parameters. To guarantee complete solution we might need 10,000 iterations. This is impractical, in most cases we can only afford a few iterations. One way to make more effective use of the iterations that we can afford is to use preconditioners in our iterative methods. The aim in using a preconditioner is to solve a related problem that has faster convergence properties and then to convert the solution of the related problem to the solution of our problem.

Given a problem associated with the operator $\mathbf{A}$ we seek an operator $\mathbf{P}$ such that a problem involving the operator $\mathbf{AP}$ has better convergence properties that $\mathbf{A}$.

The original problem

$$\mathbf{A}(\mathbf{x}) = \mathbf{y}$$

is transformed to

$$\mathbf{AP}(\tilde{\mathbf{x}}) = \mathbf{y}$$

and

$$\mathbf{P}(\tilde{\mathbf{x}}) = \mathbf{x}$$

We solve the preconditioned problem to obtain $\tilde{\mathbf{x}}$ and then apply the preconditioner one more to obtain $\mathbf{x}$.

What makes a system converge faster? Mathematically we seek a preconditioner that "clusters the eigenvalues" of the system. Ideally the operator $\mathbf{A}\,\mathbf{P}$ would be unitary, then application of the adjoint of $\mathbf{AP}$ would solve the preconditioned problem, this would mean that $\mathbf{P}'$ was actually $(\mathbf{A}'\mathbf{A})^{-1}$. This suggests that an approximation to $(\mathbf{A}'\mathbf{A})^{-1/2}$ might make a good preconditioner. Some people use the inverse of the diagonal of $(\mathbf{A}'\mathbf{A})^{-1/2}$ as a preconditioner. The most effective preconditioners are usually problem specific, they rely on knowledge of the structure of the problem to design a good preconditioner.

In our C++ classes the use of preconditioners is a trivial process. Once we have a preconditioner we can convert an operator into a preconditioned operator by combining the two in a chain. We then solve the problem associated with the chained operator and apply the preconditioner to the result to obtain the final solution. We do *not* need to modify any of the solver classes to solve the preconditioned problem. Here is a code fragment for solving a system using a preconditioner.

```
foperator A( ... ) ;  // the operator
foperator P( ... ) ;  // the preconditioner

fopChain op( A, P );  // the preconditioned operator

floatspacearray rhs;  // the right hand side

HestSolver solver( op ); // a solver for the preconditioned operator

floatspacearray x1 = solver.Solve( rhs );  // solve preconditioned problem

floatspacearray x = P.Forward( x1 );  // apply preconditioner
```

5.4 THE "Process" UTILITY

In many of the sections of this document we write a new operator and we need a small driver program to test the operator and demonstrate its application. In all these programs we give the use the choice of applying the operator in a forward or adjoint manner or solving an inverse problem involving either the forward or adjoint operator. This is so common that we have written a small utility routine to simplify the process. This routine, `Process`, is passed a `foperator`, a `floatspacearray` to apply it to and flags to indicate how it should be used. It returns a `floatspacearray`. The declaration of the routine is shown in `Process.h`. The `adj` flag selects either the forward or adjoint operation. If the `inv` flag is zero, it just applies the operator and returns the result. If the `inv` flag is 1, it will solve an inverse problems using the `HestSolver` class.

```
#ifndef PROCESS_H
#define PROCESS_H

// This Utility function applies an operator to an input space in one
// of several ways. It returns a floatspacearray as the result.
```

```
#include <foperator.h>
#include <floatspacea.h>

extern floatspacearray Process(
        const floatspacearray& input,   // The input space
        foperator& op,                  // The operator to be applied
        int adj,                        // adj=0 use forward op, adj=1 adjoint
        int inv=0,                      // inv=1 solve inverse problem
        int maxiter=2 );                // maximum number of iterations

#endif
```

Chapter 6

Derived moveout operators

[1]In this chapter we describe the implementation of various types of moveout operations, based on the general moveout operator introduced in 3.5. We will see that the coding task is fairly simple, since all we have to do is define our specific moveout operator as a class derived from the general moveout `fopMO` class (see page 350), so that it will inherit all its properties. The virtual function, `float fopMO::mapping()`, creates a dynamic binding between the two classes, telling the general moveout how to perform the squeezing/stretching operation.

We will discuss in the next sections the linear moveout operator, the normal moveout, the anelliptic moveout and the Kirchhoff moveout. Figure 6.1 shows an inheritance sketch of these three classes and its respective mapping equations. Arrows indicate derivation.

6.1 LINEAR MOVEOUT

[2]To do linear moveout correction (LMO), we need to time-shift data within each trace. Shifting data requires us to interpolate it. The easiest interpolation method is the nearest-neighbor method. For a given offset, we shift the data using the equation $\tau = t - px$. Given the location τ of the desired value, we backsolve for an integer. In the CLOP library this is done by calling the function `index()` (see man page 413), a member of the `Axis` class. This gives the nearest neighbor. The adjoint operation copies τ space back to t space. Figure 6.2 shows a CMP gather shifted upwards using LMO.

The mapping function for the linear moveout operation is given in `foplmo` (page 379).

```
class fopLMO : public fopMAP {
  public:

    float mapping( float, float ) const;
```

[1]Dave Nichols, Martin Karrenbach, Hector Urdaneta
[2]Hector Urdaneta

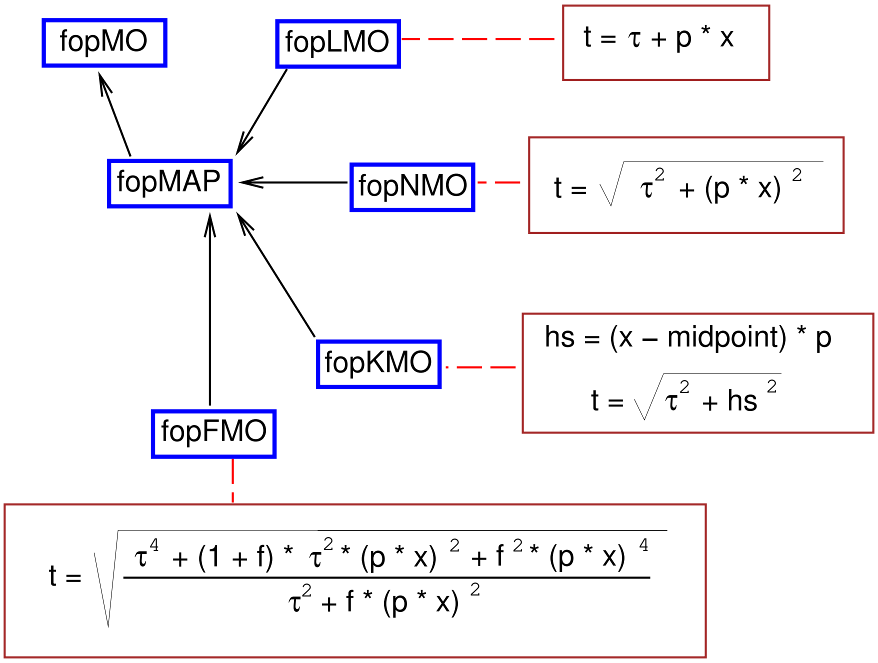

Figure 6.1: Graph of dependencies and definition of the mapping equations. Moveout-mapping [NR]

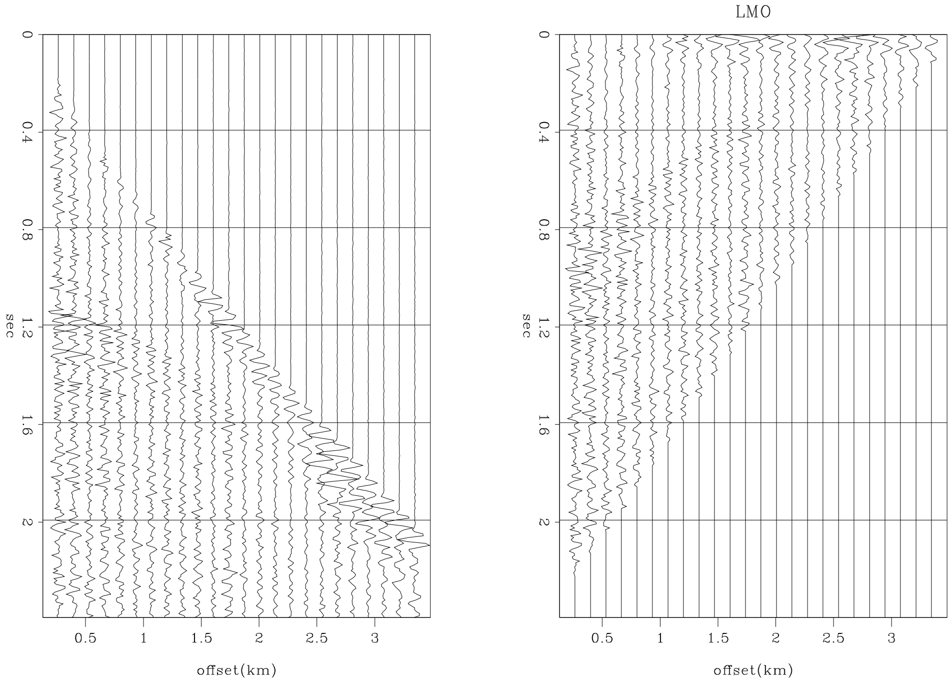

Figure 6.2: A common midpoint gather from the Gulf of Mexico before (left) and after (right) linear moveout at water velocity. Moveout-wglmo [R]

```
    // Constructors for LMO: parameters are:
    // Time axis, offset axis, slowness value
    // or array
    fopLMO( Axis &, Axis &, float );
    fopLMO( Axis &, Axis &, float a[] );

};


float fopLMO::mapping( float tau, float x ) const {

        float slowness = slow[fopMO::inlist[0].index( tau )];
        float t = tau + x * slowness;
        return t;
};
```

6.2 NORMAL MOVEOUT

[3] The operator `fopNMO` implements a moveout operation defined by the hyperbolic traveltime curve,

$$t^2 \;=\; \tau^2 \;+\; \frac{x^2}{v^2} \tag{6.1}$$

where τ is traveltime depth. This equation gives either time from a surface source to a receiver at depth τ, or it gives time to a surface receiver from an image source at depth τ.

A seismic trace is a signal $d(t)$ recorded at some constant x. We can convert the trace to a "vertical propagation" signal $m(\tau) = d(t)$ by stretching t to τ. This process is called "normal moveout correction" (NMO).

Typically we have many traces at different x distances each of which theoretically produces the same hypothetical zero-offset trace.

These concepts are implemented in the class `fopNMO`. The header `fopnmo.h` (page 379) shows the public definitions. The operator is constructed from two Axis objects that specify the time and offset axes and a slowness (1/velocity) value. The slowness can either be a constant slowness or an array of slownesses, one for each zero offset time sample.

```
class fopNMO : public fopMAP {

  public:
    float mapping( float tau, float x ) const;
    fopNMO( Axis & t, Axis & x, float slowness    );
    fopNMO( Axis & t, Axis & x, float slowarray[]  );
    fopNMO( Axis & t, Axis & x, Axis & to, Axis & xo, float slowness    );
    fopNMO( Axis & t, Axis & x, Axis & to, Axis & xo, float slowarray[]  );

};
```

[3] Dave Nichols, Hector Urdaneta, Martin Karrenbach

The `fopNMO` class that implements NMO is derived from the class `fopMAP` (page 352). The parent class handles the actual stretching of the trace, in the derived class we merely need to supply a virtual mapping function that implements the definition of the NMO transform.

Following the usual convention the forward operator is a modeling operation that models data at finite offset from data at zero-offset time. The adjoint operation takes data at finite offset and applies NMO.

The following figures illustrate, modeling, adjoint processing and inverse processing for NMO.

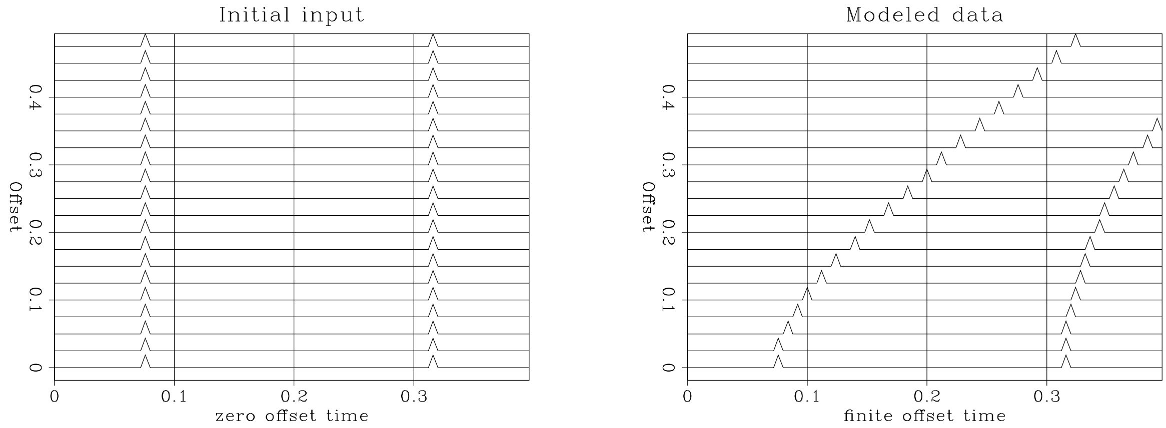

Figure 6.3: Left frame is input data, two events that are constant on all traces, the time coordinate is t_0, zero offset time. Right frames shows the NMO operator applied forward, the time coordinate is now t, time at each fixed offset. Moveout-nmoout [R]

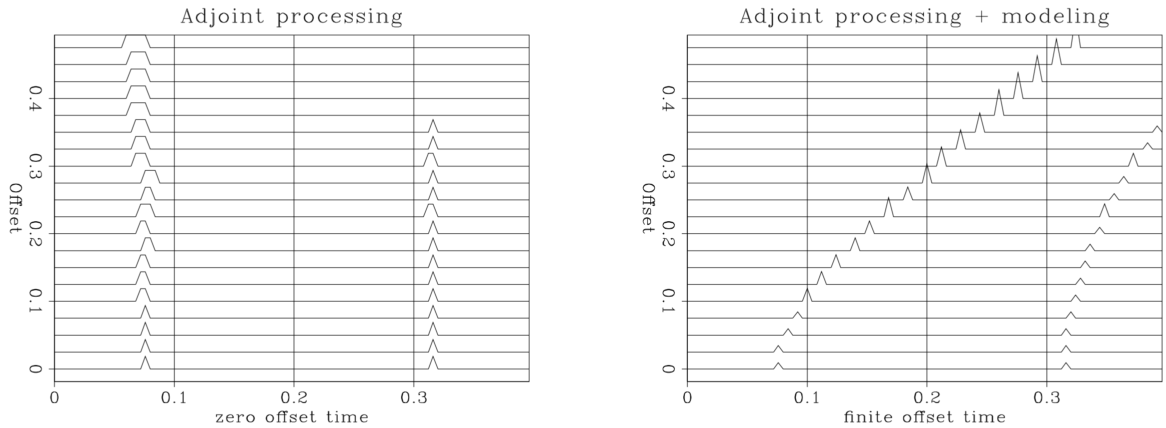

Figure 6.4: Left frame shows adjoint processing applied to the output of modeling, the time coordinate is t_0, zero-offset time. Right frame shows modeling applied to the adjoint processing, the original data *is not* recovered. Moveout-nmoadj [R]

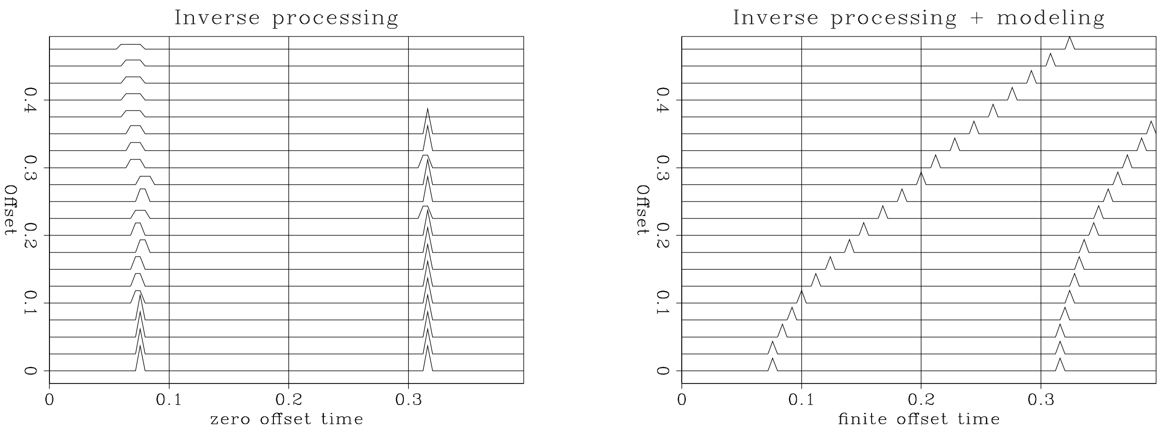

Figure 6.5: Left frame shows inverse processing applied to the previous figure, the time coordinate is t_0, zero-offset time. This result does not reproduce 6.3 because some of the original model is in the null space of the operator . Right frame shows modeling applied to the inverse processing, the original data *is* recovered even though the input to this process did not match 6.3. Moveout-nmoinv [R]

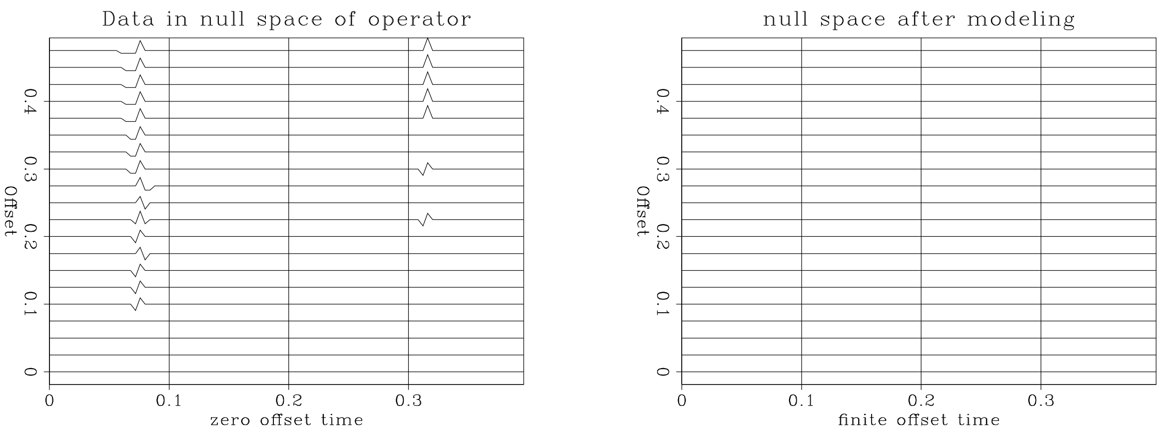

Figure 6.6: Some data in null space of NMO operator, this is the difference between the input data and the result of the inversion. The right hand frame is the result of applying the forward operator to the data in the null space. The result is zero (as expected). Moveout-nmonull [R]

6.3 ANELLIPTIC MOVEOUT

[4]Normal moveout is based on a horizontal layer model with isotropic velocities in each layer. Nature however can be more complex than that simple model. Each layer can be constituted of many more layers of very small thickness. The resulting velocity then will be angle dependent (anisotropic).

Dix equation is derived from an isotropic layered model and can be extended to anisotropic layers. An anisotropic moveout equation is derived by Dellinger et al. (1993), which adds one more parameter f to the conventional NMO computation. The first-anelliptic processing-NMO equation is:

$$t^2 = \frac{\tau^4 + (f+1)(\tau\ s\ x)^2 + f^2\ (s\ x)^4}{\tau^2 + f(s\ x)^2} \tag{6.2}$$

where f is a parameter governing anellipticity of the moveout curve. τ is the travel time for the zero offset ray, s denotes the slowness. A practical implementation can be again derived from the moveout operator `fopMAP` by specifying a new anisotropic moveout curve. The moveout then is given as the mapping function in `fopfmo` (page 382).

```
float fopFMO::mapping( float tau, float x ) const {
        float slowness = slow[ fopMO::inlist[0].index( tau )];
        float sx = x * slowness; float sx2 = sx*sx   ; float sx4 = sx2*sx2;
                                 float z2  = tau*tau ; float z4  = z2*z2;
        float t  = sqrt( (z4 + (1.+ffn)*z2 *sx2 + ffn*ffn*sx4)/(z2+ffn*sx2) );
        return t;
    };
```

The anisotropic moveout operator can be called using the following constructors:

```
class fopFMO : public fopMAP {
    public:
    float mapping( float tau, float x ) const;
    fopFMO( Axis & t, Axis & x, float slowness, float ff);
    fopFMO( Axis & t, Axis & x, float slowarray[]  ,float ff );
    fopFMO( Axis & t, Axis & x, Axis & to, Axis & xo, float slowness , float ff);
    fopFMO( Axis & t, Axis & x, Axis & to, Axis & xo, float slowarray[] , float ff);
    }
```

The following Figures 6.7 and 6.8 illustrate, modeling, adjoint processing and inverse processing for anelliptic moveout. Especially on the first event in the moved out gather we can see a clear deviation from the normal moveout curve at far offsets.

[4]Martin Karrenbach

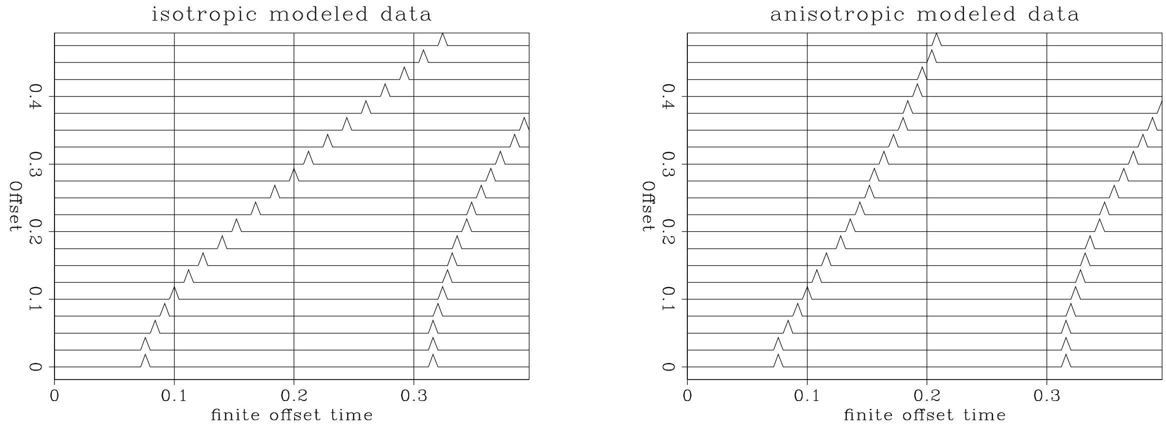

Figure 6.7: Again the input data are two events that are constant on all traces, the time coordinate is t_0, zero offset time. The left frame shows the isotropic NMO, while the right frame shows the FMO operator applied to the data, the time coordinate is now t, time at each fixed offset. Moveout-fmoout [R]

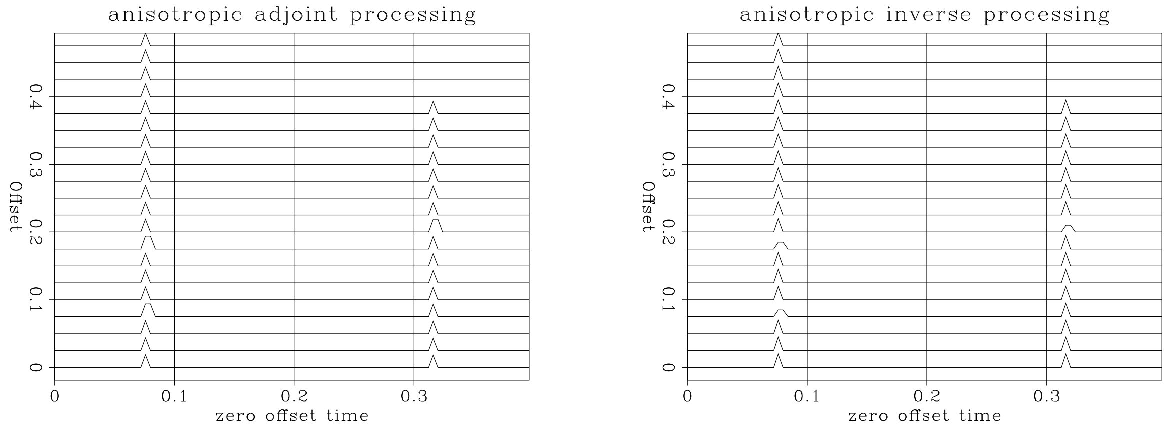

Figure 6.8: Left frame shows adjoint processing applied to the output of modeling, the time coordinate is t_0, zero offset time. The right frame shows the application of inverse processing to the same data. The differences between adjoint and inverse processing are relatively small for the chosen parameter setting. Moveout-fmoadjinv [R]

REFERENCES

Dellinger, J., Muir, F., and Karrenbach, M., 1993, Anelliptic approximations for TI media: Journal of Seismic Exploration, **2**, 23–40.

6.4 KIRCHHOFF MOVEOUT

[5]This operator implements summing along a trajectory in midpoint-offset space. Modeling or migration of seismic data after stack can be carried out by summing along hyperbolic trajectories. This integral technique (Kirchhoff Modeling/Migration) requires the knowlegde of the trace position in relation to the midpoint (CMP) to be migrated. The moveout curve is given by:

$$t = \sqrt{\frac{\tau^2 + (x-m)^2}{v^2}} \tag{6.3}$$

where t and τ are two-way travel times with v as half the medium velocity. m is the migrated midpoint position and x denotes the spatial coordinate of the trace to be summed in. We show here only the basic Kirchoff moveout class. Later examples of poststack migration and modeling will be derived from it. The Kirchoff moveout operator is derived here from the operator `fopMAP`, since it has a mapping function which uses a slowness model parameter.

```
float fopKMO::mapping( float tau, float x ) const {

        float twiceslow = slow[ fopMO::inlist[0].index( tau )];

        float hs =  ( x - midpoint ) * twiceslow;
        float  t =  sqrt( tau * tau  +  hs * hs );

        return t;
    };
```

The operator is constructed according to

```
class fopKMO : public fopMAP {

    public:

    float mapping( float tau, float x ) const;

    fopKMO(Axis & t ,Axis & x ,float slowness , float m   );

    fopKMO(Axis & t ,Axis & x ,float *slowarray  , float m );

    float midpoint ;   /* midpoint */

};
```

Figure 6.9 shows the application of the forward operator to a CMP gather that has horizontal events.

[5]Martin Karrenbach

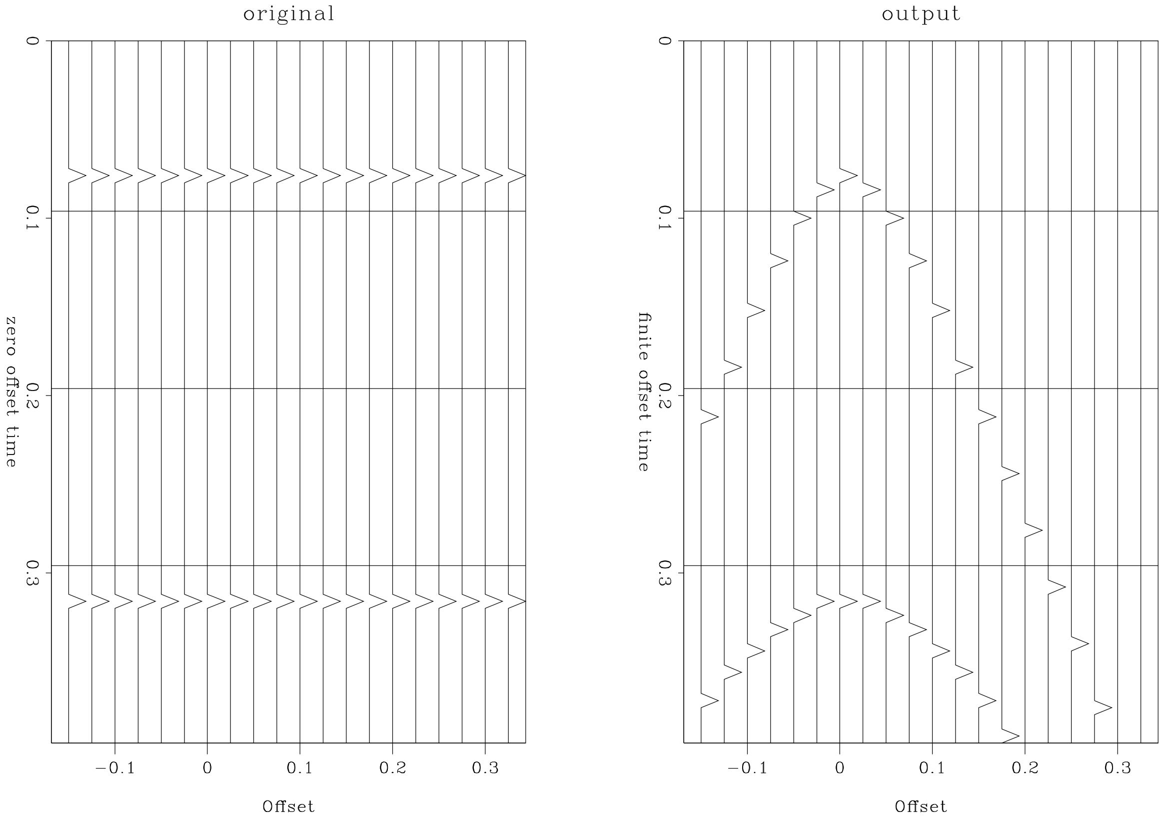

Figure 6.9: A gather is moved out with the Kirchhoff moveout equation. The CMP midpoint location coincides with the top of the hyperbola. Moveout-kmo [R]

Chapter 7

Stacking operators

Many geophysical algorithms can be cast in the form of a summation over some trajectory in a 2-D space. Familiar examples are NMO-stack, Kirchhoff migration and LMO-stack. In an earlier chapter 6 we defined moveout operators that performed NMO, LMO etc. All that we need to implement the desired algorithms is to combine these moveout operators with a stacking operator that sums over one dimension. For example NMO-stack is NMO followed by summation over the offset dimension. In our C++ classes all of these stacking operators can be coded using a chain of a moveout operator and a stacking operator `fopStack` (see man page 447).

There is no "official" definition of which operator of an operator pair is the operator itself and which is the adjoint. We like to think of the modeling operation itself as *the* operator. In this case the forward operation for NMO-stack is to begin from a model that is the stacked zero-offset trace and spray this trace to all offsets. The adjoint operation is the common-midpoint stacking procedure. On the other hand, the industry machinery keeps churning away at many processes that have well-known names, so people often think of one of them as *the* operator. Industrial data-processing operators are typically adjoints to modeling operators. In this document we always try to define the forward operator as the modeling operator.

7.1 LINEAR MOVEOUT AND STACKING

[1]The linear moveout and stacking of a data space is done by the `fopLMOstk` operator. This operator is constructed as a chain of two operators: `fopLMO` and `fopStack`. The adjoint method takes a CMP gather as input, does a linear moveout of the data and finally stacks it. The header file `foplmostk.h` (page 388) shows how the class is define and `foplmostk` (page 388) implements the constructors. Figure 7.1 shows an LMO stacked CMP gather.

```
class fopLMOstk : public fopChain {
  public:
```

[1]Hector Urdaneta

```
    // Contructors for fopNMOstk
    // Take time Axis, offset Axis, and slowness (1./vel)
    // parameter, as either a value or an array of
    // values.
    fopLMOstk(Axis &, Axis &, float);
    fopLMOstk(Axis &, Axis &, float a[]);

};



fopLMOstk::fopLMOstk(Axis & t, Axis & x, float s) {

num=2;
ops = new foperator*[2];
ops[0] = new fopLMO( t, x, s );
ops[1] = new fopStack( x, 1);

}
fopLMOstk::fopLMOstk(Axis & t, Axis & x, float s[]) {

num=2;
ops = new foperator*[2];
ops[0] = new fopLMO( t, x, s );
ops[1] = new fopStack( x, 1);

}
```

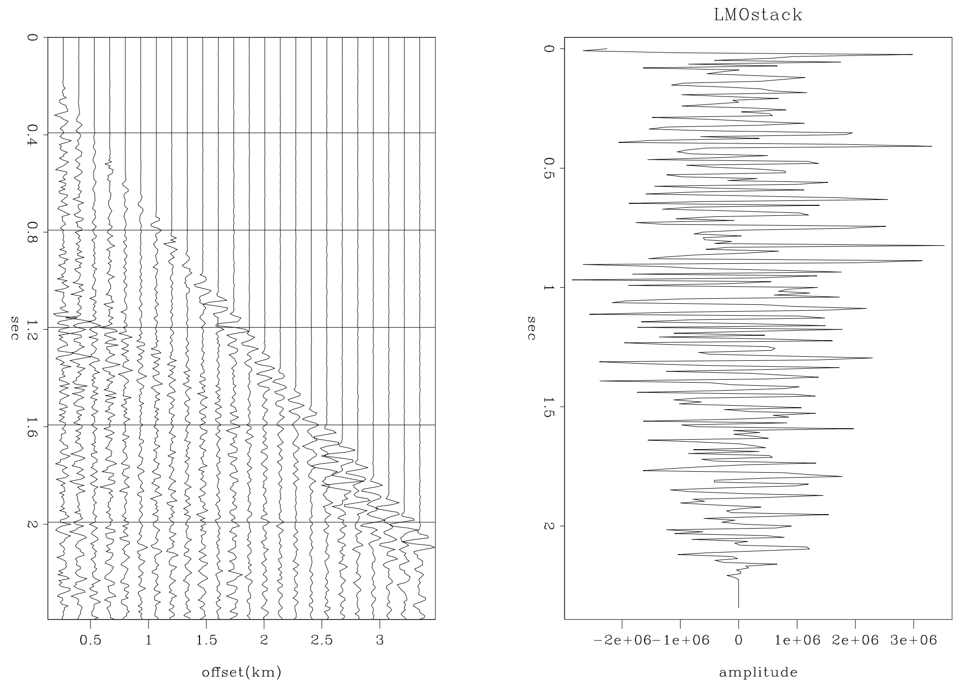

Figure 7.1: A common midpoint gather before (left) and after (right) LMO stacking at water velocity. stacking-wglmostk [R]

7.2 NMO AND STACK

[2] NMO stack is the process of applying normal moveout to all the traces in a CMP gather and then summing the traces. We implement NMO stack as a composition of two previously defined operators, the NMO operator "fopNMO" and the stacking operator "fopStack". To avoid the nuisance of having to create two operators each time we want to perform NMO and stack we create a new class, derived from the fopchain class, that builds a chain of the two operators for us. The code for this is almost exacty the same as the code for the LMO stack class (page 388) except that the fopLMO operator is replaced by a `fopNMO` operator (page 379). Note that, as usual, the forward operation is the modeling operation. The adjoint is the familiar CMP stack.

Let $\mathbf{A}$ denote NMO-stack modeling (modeling from one zero offset stack trace to a full gather with hyperbolic moveout), and let the NMO-stack be defined by invoking the adjoint of the operator. Figure 7.2 shows a model trace $\mathbf{x}$ on the left and the result of applying the forward operator $\mathbf{A}\,\mathbf{x}$ on the right, this is our synthetic CMP gather, $\mathbf{y}$.

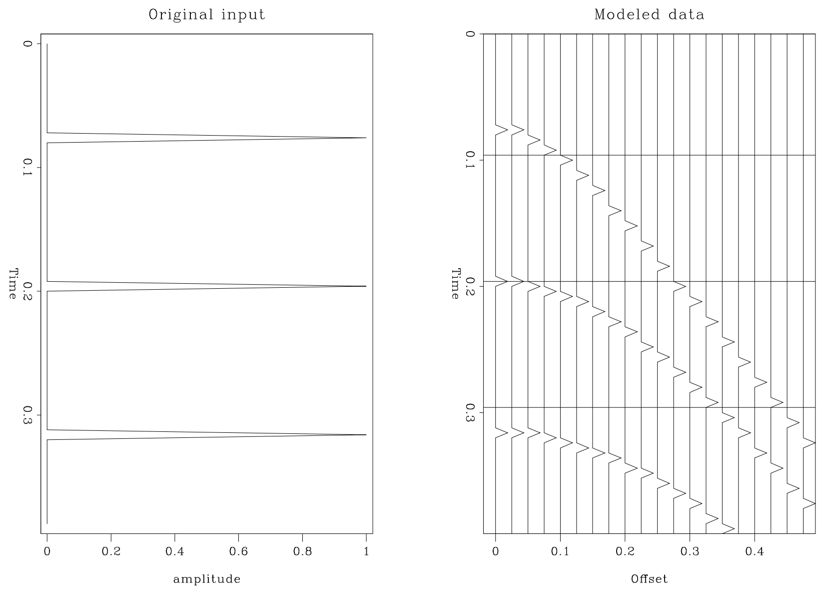

Figure 7.2: Left frame = input, right frame = modeling. stacking-stackout [R]

Figure 7.3 shows the result of adjoint processing on the synthetic CMP gather. The left frame is $\mathbf{A}'\mathbf{y} = \mathbf{A}'\mathbf{A}\mathbf{x}$; we have not recovered our input model, $\mathbf{x}$, because the NMO-stack operator is not unitary. The right frame shows the result of modeling the processed trace $\mathbf{A}\,\mathbf{A}'\,\mathbf{y}$.

We can use a least-squares inverse solver to improve on the adjoint processing. This procedure minimizes $||\mathbf{A}\,\mathbf{x} - \mathbf{y}||^2$. Figure 7.4 shows the true model and several

[2]Dave Nichols

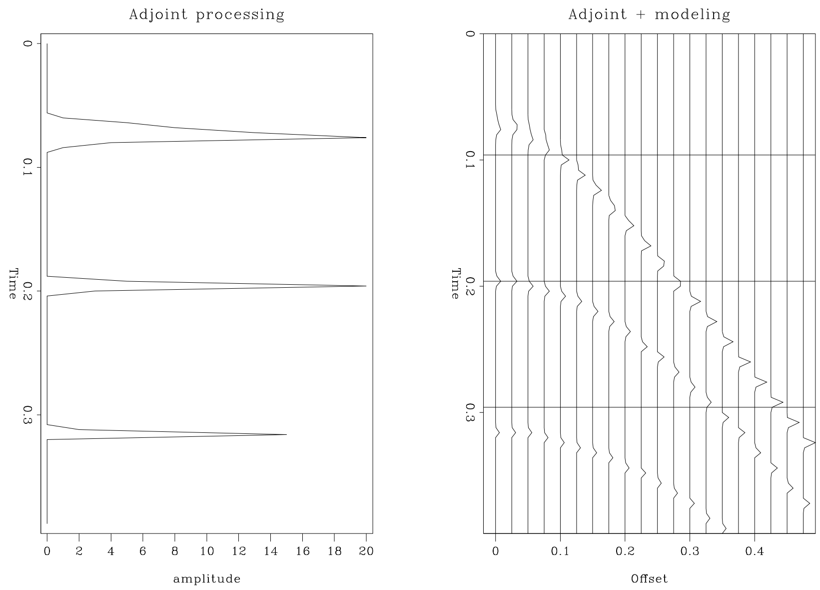

Figure 7.3: Left frame = adjoint processing, right frame = adjoint processing + modeling. stacking-stackadj [R]

iterations of an iterative inverse procedure. After ten iterations the exact result has been recovered.

In Figure 7.5 the result of modeling the inverted result is shown, it completeley recovers the input to the inversion process. It is interesting to contrast this result with the inversion of the NMO operator in section 6.2. In that inversion we observed a large null space in the operator and the original model was not recovered. One major reason that this does not happen with NMO-stack is that we have reduced the number of variables in the model by a factor of twenty (the number of offsets we stack over).

7.3 KIRCHHOFF OPERATOR

[3]The Kirchhoff poststack operator (Kirchhoff moveout + Stack over offset) is implemented as a composition of two previously defined operators, the KMO operator `fopKMO` and the stacking operator `fopStack`. To avoid the nuisance of having to create two operators each time we want to perform KMO and Stack, we derive a new class `fopKMOstksingle`, from class `fopChain`. For convenience we create an intermediate class `fopKMOstksingle`, that handles only the moveout and stack for a single midpoint gather (or cube). More complicated output geometries, such as post-stack 2-D panels or 3-D cubes, produce their operators by chaining the single gather operator.

```
class fopKMOstksingle : public fopChain {
```

[3]Martin Karrenbach

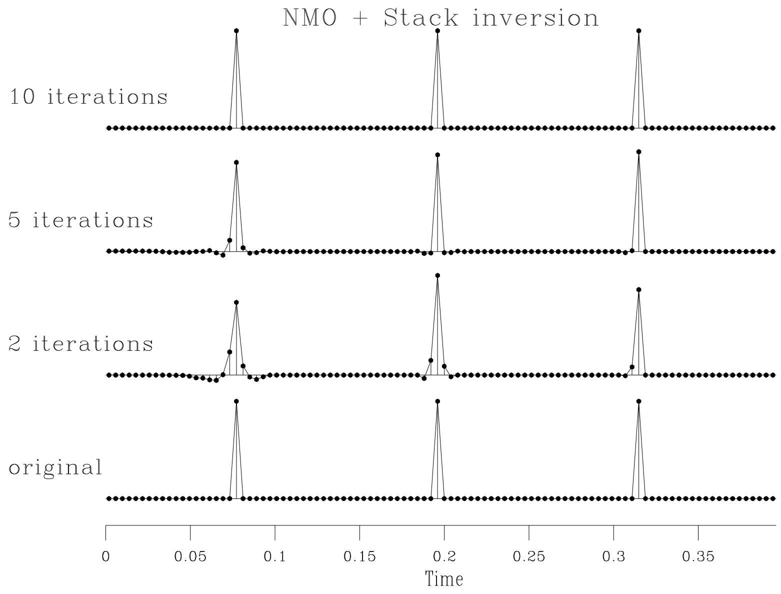

Figure 7.4: Iterations of inverse processing stacking-stackiters [R]

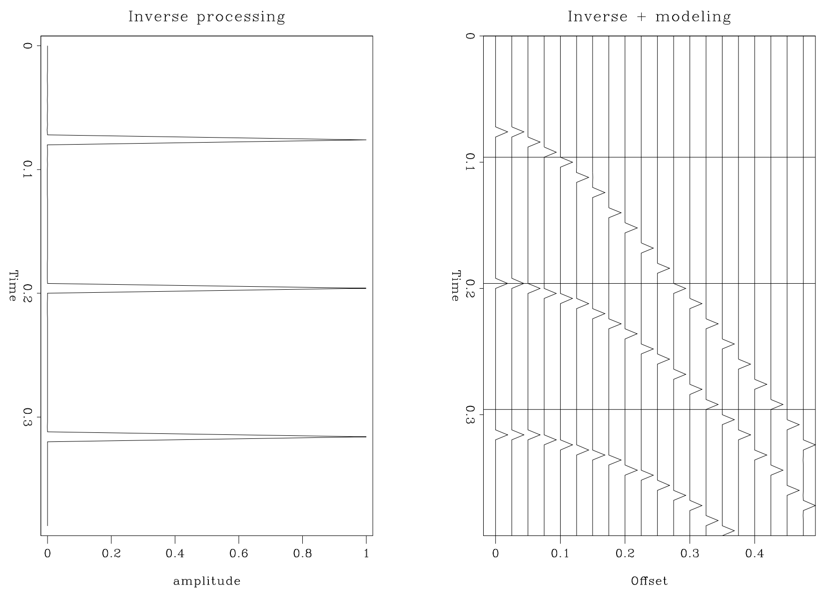

Figure 7.5: Left frame = inverse processing, right frame = inverse processing + modeling stacking-stackinv [R]

```
  public:

    // Contructors for fopKMOstk
    // Take time Axis, offset Axis, midpoint Axis and slowness (1./vel)
    // parameter, as either a value or an array
    // values.
    fopKMOstksingle( Axis &, Axis &, float m, float a);
    fopKMOstksingle( Axis &, Axis &, float m, float a[]);

};
```

All that is required to implement this new class is to write a constructor that builds the two separate operators and stores them internally in the `fopChain` class. Application of the two operators forward and adjoint is handled by the parent class so they can inherit its behavior. We do not need to reimplement the code.

```
#include <fopkmostksingle.h>
#include <fopkmo.h>
#include <fopstack.h>

// Operator to do KMO and stack (as adjoint) on a single trace.

fopKMOstksingle::fopKMOstksingle(Axis & t,Axis & x, float m, float s) {

  num = 2;
  ops = new foperator*[num];
  ops[0] = new fopKMO( t, x, s ,m );
  ops[1] = new fopStack( x, 1);
}

fopKMOstksingle::fopKMOstksingle(Axis & t,Axis & x, float m ,float s[]) {

  num = 2;
  ops = new foperator*[num];
  ops[0] = new fopKMO( t, x, s ,m );
  ops[1] = new fopStack( x, 1);
}
```

Let $\mathrm{KMOSTK_s}$ denote Kirchhoff modeling from one zero-offset trace to a full gather with hyperbolic moveout. By invoking the adjoint the operator will sum all moved-out traces into a single stacked trace. Thus `fopKMOstksingle` performs the following operation on some given input data set:

$$\mathrm{KMOSTK_s} = \mathrm{STACK}^t \quad \mathrm{KMO} \tag{7.1}$$

What we consider the normal Kirchhoff stack operator is here the adjoint composite operator $\mathrm{KMOSTK_S}^t$.

7.3.1 Poststack Kirchhoff Operator

The previously introduced class `fopKMOstksingle` can be used for building the more complicated operator which models or migrates a whole poststack panel. Again we chain operators which loop over the midpoint axes and sum the data.

The poststack Kirchhoff operator needs three axes to work on: the time axis, the unmigrated midpoint axis and the migrated midpoint axis. We construct it in the familiar manner:

```
class fopKMOstk : public fopChain {
  public:

    // Contructors for fopKMOstk
    // Take time Axis, offset Axis, midpoint Axis and slowness (1./vel)
    // parameter, as either a value or an array
    // values.
    fopKMOstk( Axis &, Axis &, Axis &, float a);
    fopKMOstk( Axis &, Axis &, Axis &, float a[]);

};
```

The actual work is done in the following operator chain:

$$\text{KIRCH} = \text{KMOSTK}_s \quad \text{MERGE}^t \tag{7.2}$$

The forward input space is the "time–migrated midpoint" space. We introduce a new dimension the "unmigrated midpoint" axis, which will contain our modeling results.

An operator array is constructed from the input space, with each subspace containing one midpoint. Each of those subspaces is moved out and stacked using `fopKMOstksingle`. The final step is the stack over migrated midpoint axis to obtain the output section. The output is "time–unmigrated midpoint" space. The forward operation thus implements a Kirchhoff modeling process. The adjoint operation is to start with "time" and "unmigrated midpoint" space and applying the adjoint operator chain, ending up with the "time" and "migrated midpoint" space, a migrated dataset.

```
fopKMOstk::fopKMOstk(Axis & t,Axis & m, Axis & x, float s) {

   // make an array of kmostack operators
   fopArray * kmostkops = new fopArray( 1, m.length );

   for ( int i=0; i<m.length; i++ ){
       fopKMOstksingle * kmostack = new fopKMOstksingle( t, x, m.value(i), s );
       kmostkops->set( 0, i, kmostack );
   }
   fopMerge   * mergeop   = new fopMerge( m );
   fopAdjoint * makepanel = new fopAdjoint( mergeop );

   num = 2;
   ops = new foperator*[num];
   ops[0] = kmostkops;
   ops[1] = makepanel;
}
fopKMOstk::fopKMOstk(Axis & t,Axis & x, Axis & m ,float s[]) {

   // make an array of kmostack operators
   fopArray * kmostkops = new fopArray( 1, m.length );

   for ( int i=0; i<m.length; i++ ){
       fopKMOstksingle * kmostack = new fopKMOstksingle( t, x, m.value(i), s );
       kmostkops->set( 0, i, kmostack );
   }
   fopMerge   * mergeop   = new fopMerge( m );
   fopAdjoint * makepanel = new fopAdjoint( mergeop );

   num = 2;
   ops = new foperator*[num];
   ops[0] = kmostkops;
   ops[1] = makepanel;
}
```

The following Figures 7.6 and 7.7 show results of Kirchhoff modeling and migration using the chained operators.

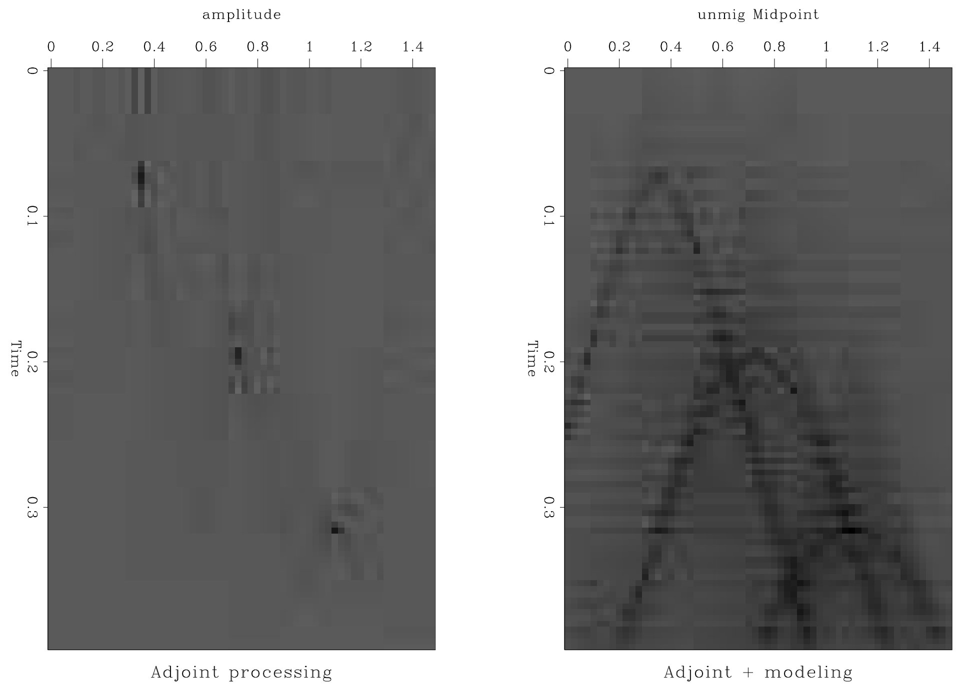

Figure 7.6: Left frame = adjoint processing, right frame = adjoint processing + modeling. stacking-stackadjp [R]

7.4 VELOCITY ANALYSIS

[4] An important transformation in exploration geophysics takes the data as a function of shot-receiver offset and transforms it to data as a function of apparent velocity. Data is summed along hyperbolas of many velocities. This important industrial process is the adjoint to another that may be easier to grasp: data is synthesized by a superposition of many hyperbolas. The hyperbolas have various asymptotes (velocities) and various tops (apexes).

The construction of the velocity analysis operator, is done by chaining the NMO stack operator (see section 7.2) and the merge operator (see man pages 437). The adjoint operator takes a set of linearly increasing slownesses values that are put into an axis. The adjoint operator applies first, for each slowness value, the NMO stack operator, which in turn, does the normal moveout and the stacking of the data. When this process is finished, we end up having one-dimensional data sets for each individual velocity. The merge operator finishes the job, putting all these one-dimensional spaces into a two-dimensional space where the new axis is a slowness axis.

[4] Hector Urdaneta

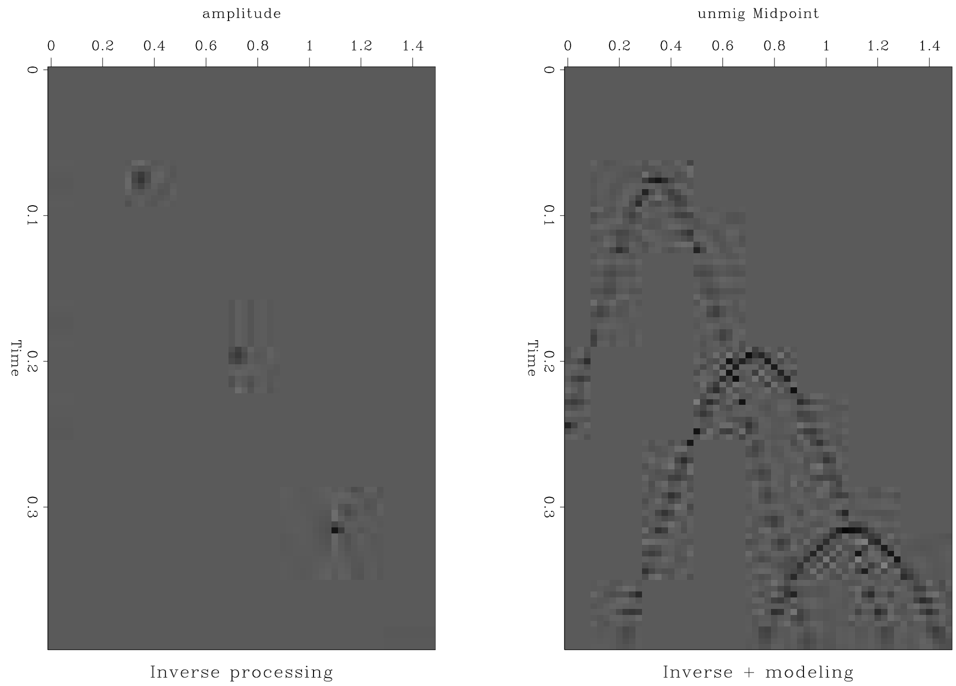

Figure 7.7: Left frame = inverse processing, right frame = inverse processing (10 iteration) + modeling. stacking-stackinvp [R]

A graph of inheritances for the velan operator is shown in Figure 7.8. Notice how you can also construct a velocity analysis operator, for the LMO operator (see next section), in the same way as for the NMO operator.

```
class fopVelan : public fopChain {
  public:
        //    time       offset    slowness  mode 0=slowness 1=vel
    fopVelan(Axis & t,  Axis & x,  Axis& s, int mode=0   );
};


fopVelan::fopVelan(Axis & t, Axis & x, Axis& s, int mode ) {

int len = s.length;

// make a ( 1, len ) array of nmostack operators
fopArray * nmostkops = new fopArray(  1, len );

for( int i=0; i<len; i++ ){
    float slow ;
    if( mode == 0 ){
       slow = s.value(i);
    }else{
       slow = 1./s.value(i);
    }
    // build a single nmostack operator
    fopNMOstk * stkop = new fopNMOstk(t, x, slow );
    // put it in the array
    nmostkops->set( 0, i, stkop );
}

 // make a Merge operator
fopMerge * mergeop = new fopMerge( s );
```

```
 // make its adjoint
fopAdjoint * adjmerge = new fopAdjoint( mergeop );

// make a chain of the two
num=2;
ops = new foperator*[2];
ops[0] = nmostkops;
ops[1] = adjmerge;

}
```

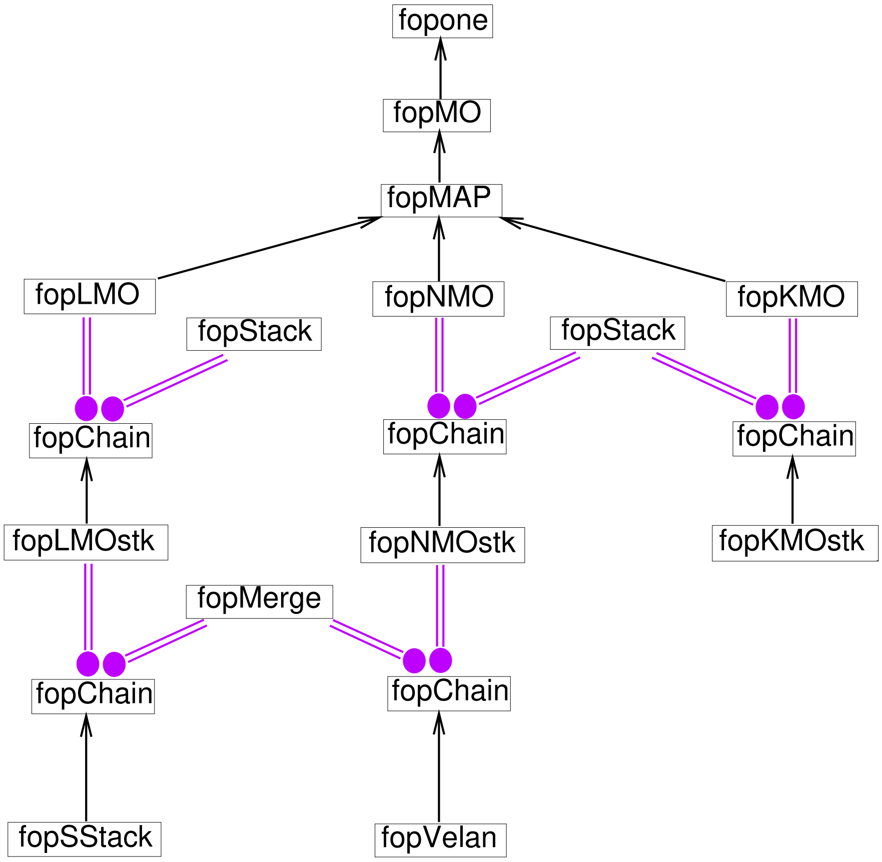

Figure 7.8: Graph of dependencies for the velocity analysis operator. Class relationships are depicted according to the Booch Method: an arrow indicates inheritance and double lines with a solid dot connect a class with the class it contains $\boxed{\text{stacking-dependencies}}$ [NR]

The constructors for the `fopVelan` class are defined in the header file `fopvelan.h` (page 395). `fopvelan` (page 396) shows the coding of the `fopVelan` constructor.

An example of applying the `fopVelan` operator to field data is shown in Figure 7.9.

Figures 7.9 to 7.12 show examples of modeling, processing and inverse processing for the velocity analysis operator.

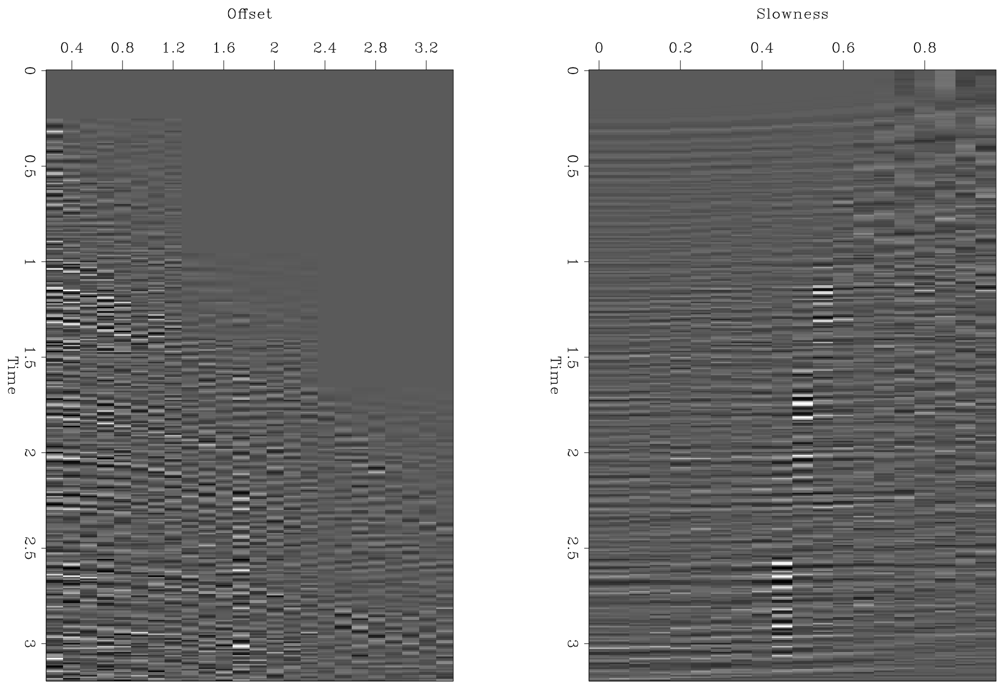

Figure 7.9: Transformation of data as a function of offset (left) to data as a function of slowness (right) using the `fopVelan` operator. stacking-mutvel [R]

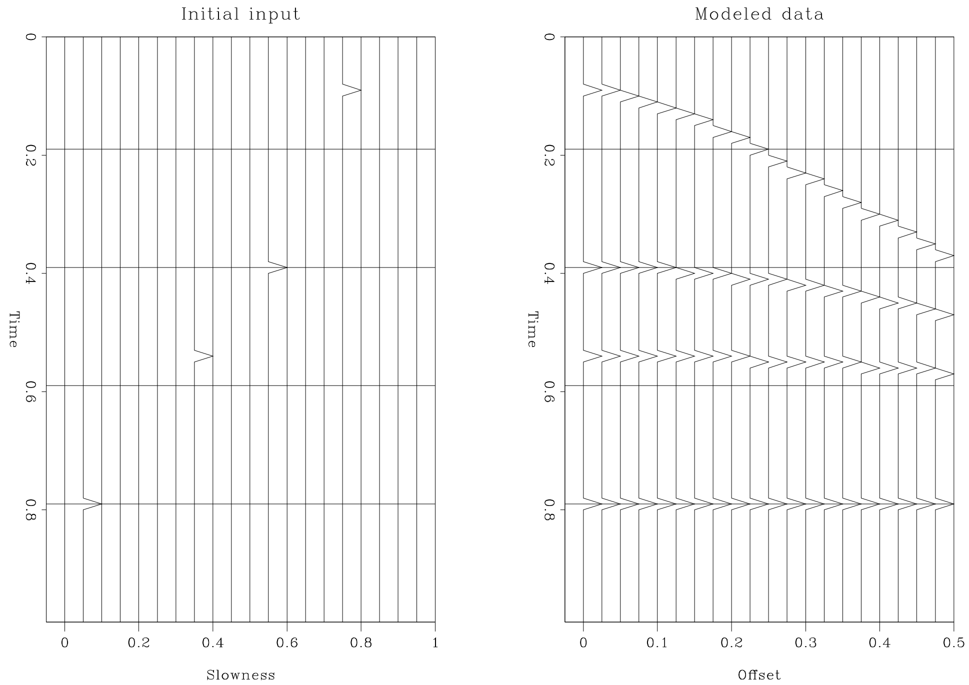

Figure 7.10: The left frame shows three events in a time-slowness space. The right frame shows the spreading of this events across the offset axis, made by the velocity analysis operator applied forwards. stacking-velanmod [R]

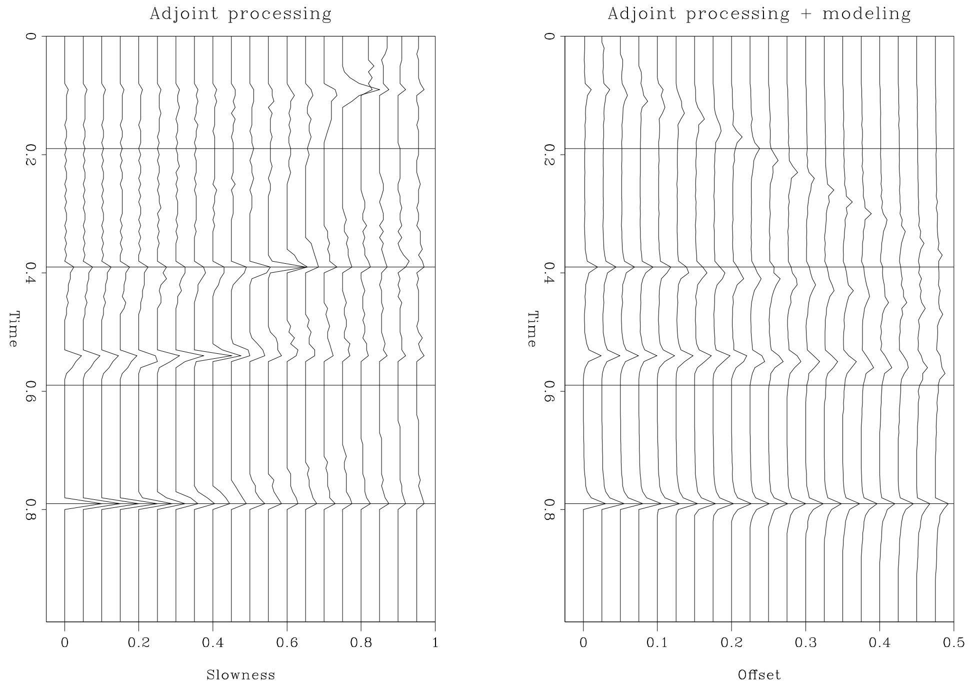

Figure 7.11: The left frame shows the result for the adjoint operator applied to the modeled data. The right frame shows the modeling operation applied to the adjoint processing data. The adjoint operation does not recover the modeled data. stacking-velanadj [R]

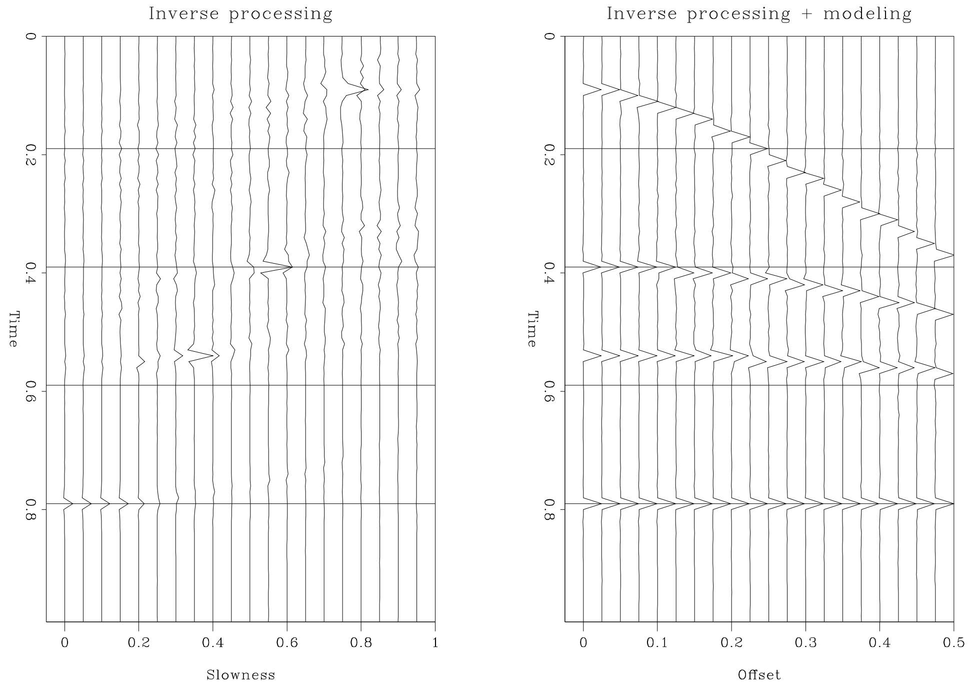

Figure 7.12: The left frame shows the inverse operator applied to the modeled data. The right frame shows modeling applied to the inverse processing, where even though the input does not reproduce Figure 7.10, the modeled data is recovered. stacking-velaninv [R]

7.5 SLANT STACK AND PRECONDITIONED INVERSION

[5] The slant stack process, LMO and stack for a range of slowesses, is implemented in much the same way as the velocity analysis operator. It is a composition of an array of LMO-stack operators (class `fopLMOstk`) and a merge operator (class `fopMerge`).

```
class fopSStack : public fopChain {
  public:
    //        time        offset    slowness
    fopSStack(Axis & t,  Axis & x,  Axis& s  );
};
```

All that is required to implement this new class is to write a constructor that builds the two operators and chains them together. I will not show this code here because it is identical to the code for constructing the `fopVelan` operator, except that it uses the `fopLMOstk` instead of the `fopNMOstk` operator internally (see `fopvelan` 395).

The forward operation of the slant stack operator models data from slowness-time space to offset-time space. In all the following figures the right hand plot is the result of applying the modeling operator to the left hand plot. Figure 7.13 shows the input, slowness-time, model on the left, it has three spikes in the slowness domain. The forward operator maps each spike to a linear event in offset-time space.

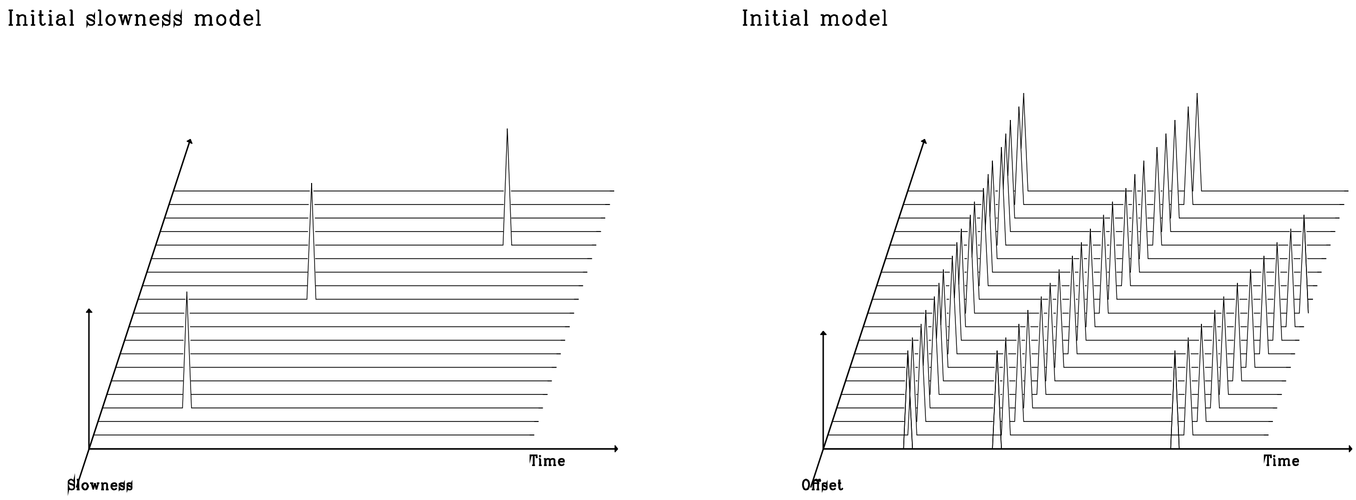

Figure 7.13: Left frame = input, right frame = modeling. stacking-out.1 [R]

Adjoint processing of the modeled data is shown in figure 7.14, the three spikes have not been recovered, instead the familiar "cross" pattern is seen. The incomplete reconstruction is caused by two effects the absence of the "rho" filter and the finite aperture of the modeled data.

[5] Dave Nichols

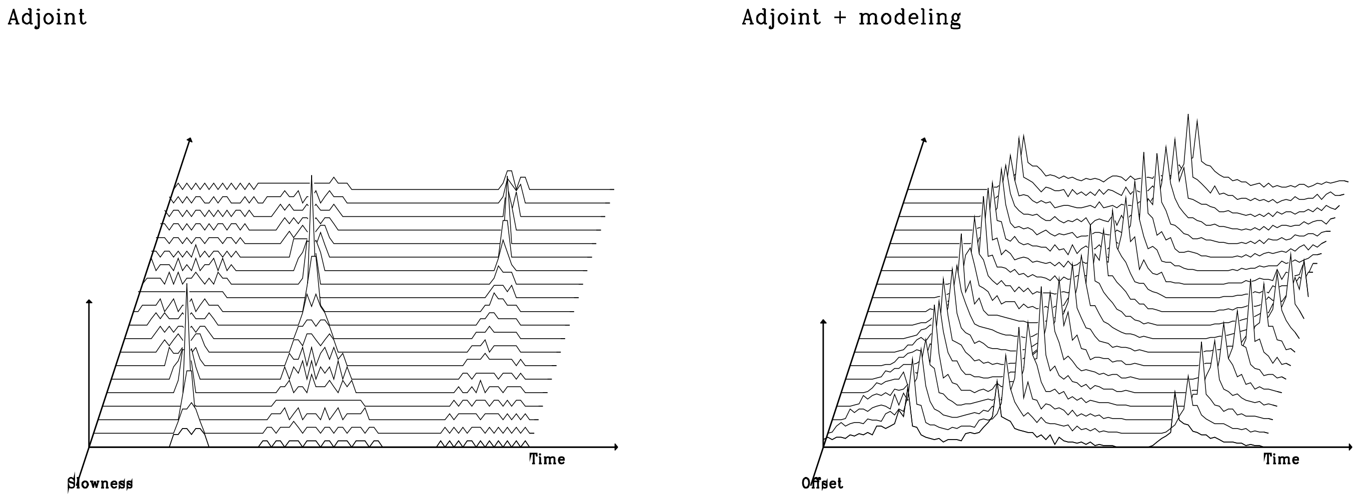

Figure 7.14: Left frame = adjoint processing, right frame = adjoint processing + modeling. stacking-adj.1 [R]

Theory tells us that if the slant stack is performed by integration in an infinite medium (rather than summation in a finite medium) then the inverse of the forward operator is the adjoint operator plus the "rho" filter, the rho filter scales each frequency component by its magnitude. The lack of this frequency scaling is clearly seen when the adjoint processed data is modeled back to the offset-time space. Instead of the sharp linear event that was input a much lower frequency event is seen.

The "cross" in the result of adjoint processing is caused by the finite aperture of the input data. The slope of the two arms of the cross correspond to the minimum and maximum offset in the input. If the input data had infinite extent we would expect the cross to be absent.

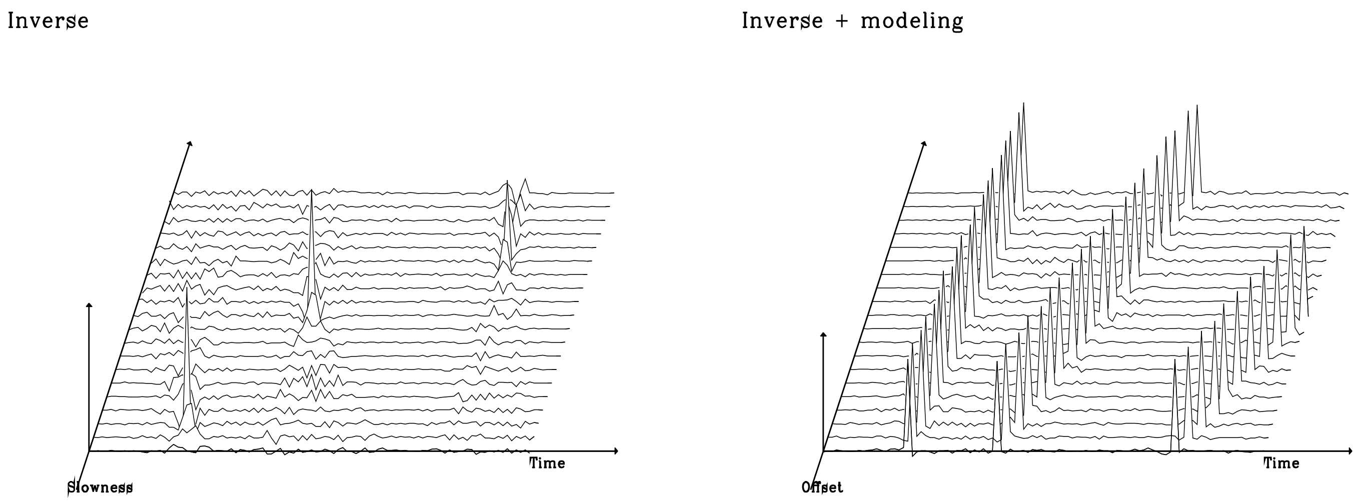

Figure 7.15: Left frame = inverse processing, right frame = inverse processing + modeling. stacking-inv.1 [R]

Using a Hestenes solver to perform inverse processing removes almost all of these artifacts. Figure 7.15 shows the result of inverse processing after ten iterations. The result is much closer to the original model. When this data is modeled back to the offset-time domain it is a closer match to the input data. The iterative solver reported the following statistics from the inversion steps.

```
iteration: 0   residual norm: 5.40984   norm ratio: 0.757529
iteration: 1   residual norm: 3.60264   norm ratio: 0.504471
iteration: 2   residual norm: 2.54561   norm ratio: 0.356457
iteration: 4   residual norm: 1.5167    norm ratio: 0.212381
iteration: 6   residual norm: 1.06988   norm ratio: 0.149813
iteration: 8   residual norm: 0.858584  norm ratio: 0.120226
iteration: 10  residual norm: 0.700457  norm ratio: 0.0980836
```

The first step (which is essentially a scaled adjoint) produces a model that is a poor match to the input data. After 10 iterations there is still a residual with a norm that is 10% of the original but the fit to the data is much improved. Note that a 10% ratio in the norm means that all but 1% of the energy in the input has been modeled.

In an attempt to improve this result I decided to use the rho filter as a preconditioner for the solver. Figure 7.16 shows the result of using the adjoint operator with a rho filter. This still does not reconstruct the original data because of the finite aperture and finite sampling of the input. The slowness-time data still has the cross artifact in it because the adjoint has not corrected fot the finite aperture. The modeled output on the right now has a higher frequency content than the input rather than a lower one. This is because the rho filter has boosted the high frequencies too much, for finite aperture data the filter should be less powerful.

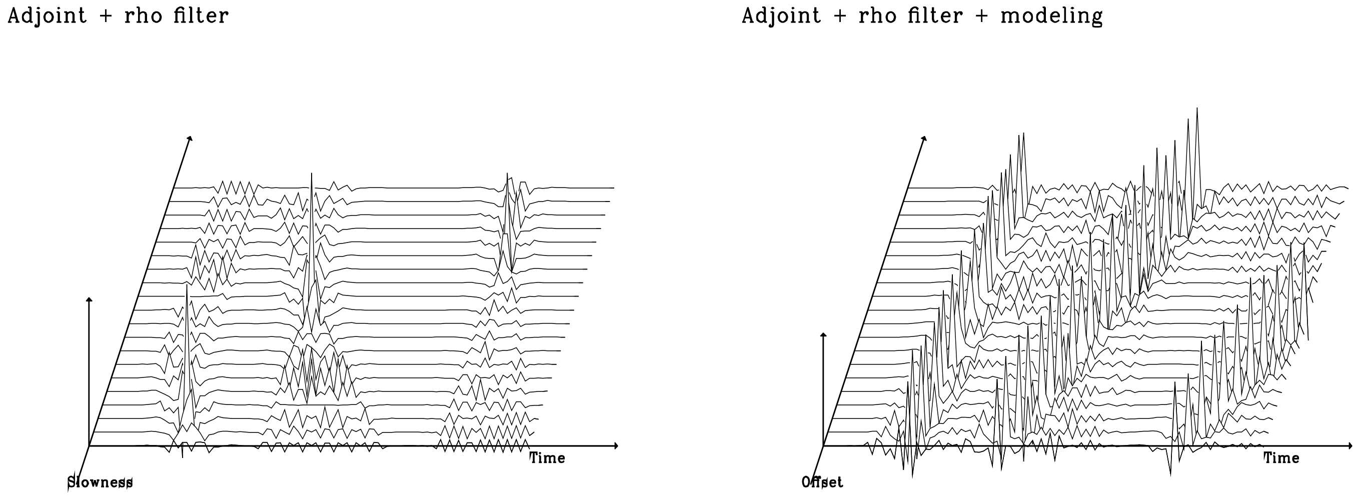

Figure 7.16: Left frame = adjoint processing + rho filter, right frame = adjoint processing + rho filter + modeling. stacking-adjrho.1 [R]

The rho filter can be regared as an estimate of $(A'A)^{-1}$ for the slant stack operator. As discussed in the chapter on solvers (chapter 5). We can use the square root of this operator as a preconditioner. The square root of the rho filter is a operator that scales each frequency by the square root of its magnitude. I use an operator of the class `fopContrunc` to implement this operator. The filter is an approximate time domain representation of the half-rho filter.

Figure 7.17 shows the result of solving the preconditioned inverse operator, it is a little worse than the non preconditioned result! The iterative solver reported the following statistics.

```
iteration: 0   residual norm: 3.70267   norm ratio: 0.518477
iteration: 1   residual norm: 2.78889   norm ratio: 0.390522
iteration: 2   residual norm: 2.44061   norm ratio: 0.341753
iteration: 4   residual norm: 2.04424   norm ratio: 0.286251
iteration: 6   residual norm: 1.79228   norm ratio: 0.25097
iteration: 8   residual norm: 1.6462    norm ratio: 0.230514
iteration: 10  residual norm: 1.53393   norm ratio: 0.214793
```

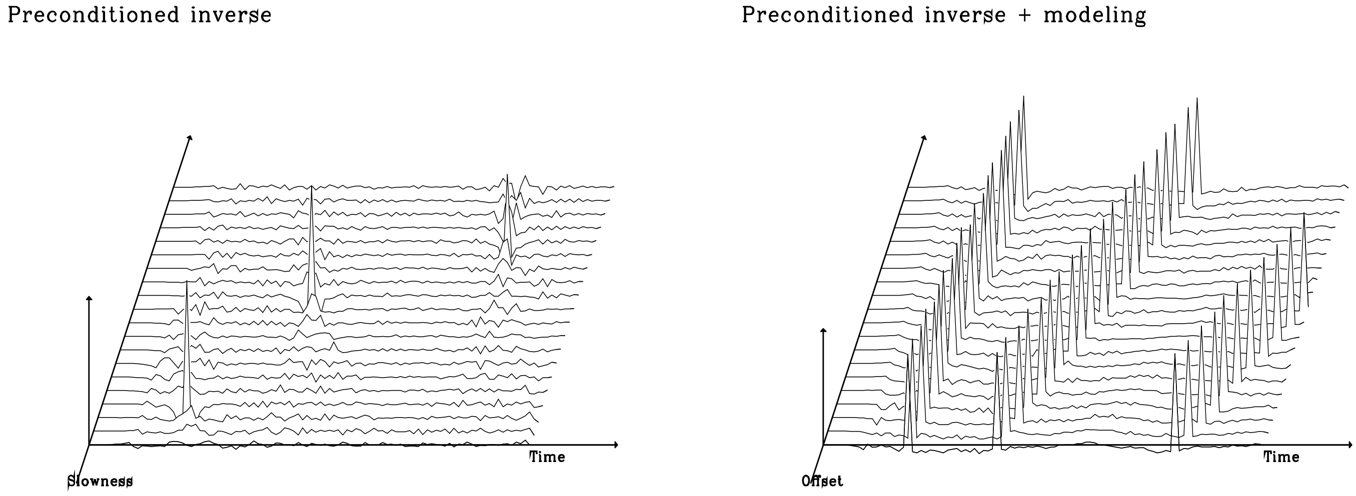

Figure 7.17: Left frame = inverse processing with half-rho preconditioner, right frame = inverse processing with half-rho preconditioner + modeling. stacking-invrho.1 [R]

We can see that the use of the preconditioner did indeed make the operator more unitary. The first iteration has a residual norm that is less than that for the original operator. However, the preconditioner has not accelerated convergence of the algorithm, instead it has *slowed* it. The reason for this appears to be that most of the artifacts from the simple adjoint are due to the truncation and not the filtering effect. The rho filter has impeded the correction of these effects. If we looked at the eigenvalues of the transformation we would probably find that we had moved the largest eigenvalue closer to unity but we had not improved the clustering of the eigenvalues.

What can we conclude from this? It tells us that a preconditioner designed from the continuous problem is not necessarily a good precoditioner for the discrete problem. We need to look at the discrete problem more thoroughly to design a good preconditioner.

Chapter 8

Missing-data restoration

[1]The interpolation of missing data is important to reduce the effort and risk in solving inverse problems. In this chapter, we solve two simple least-squares problems for missing data. In the first case, since the filter is known, the problem is linear. In the second case, because the filter is also unknown we must solve a nonlinear problem. Solving the linear problem by conjugate gradient yields the same results as Claerbout's solution in PVI (Claerbout, 1992). For the nonlinear case I have chosen a different solving scheme from the one in PVI. The results of the two schemes depart slightly and can be explained by the difference in the numerical algorithms.

8.1 INTERPOLATING MISSING DATA WITH A KNOWN FILTER

A method for restoring missing data is to ensure that the restored data, after specified filtering, has minimum energy.

To illustrate this method, let m denote a missing value. The dataset on which the examples are based is $\mathbf{d} = (\cdots, m, m, 1, m, 2, 1, 2, m, m, \cdots)$. The forward modeling operator is the convolution with the given filter $\mathbf{f}$. Now the problem is to minimize $\mathbf{f} * \mathbf{d}$ as follows:

$$0 \quad \approx \quad \mathbf{f} * \mathbf{d} \tag{8.1}$$

Let $\mathbf{M}$ denote a mask operator that hides the known values in the data. This operator is a diagonal square matrix with 1 or 0 as elements. For example, suppose that we have data such as $(m, m, 1, m, 2, 1, m, m)$. Then the mask operator for these data

[1]Hyang-Im Oh

looks like this:

$$\begin{pmatrix} 1 & . & . & . & . & . & . & . \\ . & 1 & . & . & . & . & . & . \\ . & . & 0 & . & . & . & . & . \\ . & . & . & 1 & . & . & . & . \\ . & . & . & . & 0 & . & . & . \\ . & . & . & . & . & 0 & . & . \\ . & . & . & . & . & . & 1 & . \\ . & . & . & . & . & . & . & 1 \end{pmatrix} \tag{8.2}$$

Implementing the mask operator, we can represent the data in this way:

$$\mathbf{d} \quad = \quad \mathbf{Md} + (\mathbf{I} - \mathbf{M})\,\mathbf{d} \tag{8.3}$$

where $\mathbf{Md}$ is the missing part and $(\mathbf{I} - \mathbf{M})\,\mathbf{d}$ the known part. Defining $(\mathbf{I} - \mathbf{M})\,\mathbf{d}$ as $\mathbf{d}_k$ and rewriting the equation (8.1) yields

$$0 \quad \approx \quad \mathbf{f} * (\mathbf{Md}) + \mathbf{f} * \mathbf{d}_k \tag{8.4}$$

We can then separate the known part from the unknown in the equation (8.4) and write it in the form of $\mathbf{y} = \mathbf{Ax}$, as follows:

$$-\,\mathbf{f} * \mathbf{d}_k \quad \approx \quad \mathbf{f} * (\mathbf{Md}) \tag{8.5}$$

The left side, $-\mathbf{f} * \mathbf{d}_k$, is known. The operator is $(\mathbf{f}*)\,\mathbf{M}$. We simply use the C++ least square solver to get $\mathbf{d}$.

```
//   miss1.cc :  1-D missing data with known filter
#include "miss1.h"
floatspace miss1(floatspace & filtsp, floatspace & data, int niter) {

        Axislist ax = data.getaxislist();           // Get axis info of data

        fopMask    mymask(ax.list[0],data);         // Mask operator
        fopContran myconv(filtsp,0);                // Convolution operator
        fopChain   myop( myconv,mymask);            // Conv*Mask
        floatspace resid = -(myconv.Forward(data));  // Make residual
        floatspace outspace = (fophestenes(data,resid,myop,niter))(0,0);//Solver

        return outspace;
}
```

We are now ready to code the problem. Let me explain the program briefly. The two operators applied to the missing data are masking (`mymask`) and convolution (`myconv`). The CLOP library class `fopMask` is constructed by taking either a `floatspace` or an integer array as an argument together with an axis to apply the operator on (see man page 435). According to the position of zero(missing data) in the argument, 1 or 0 is assigned to the diagonal elements in the operator matrix like the one in the equation (8.2). The CLOP class `fopContran` implements a convolution with the transient end effect (see page 341). The convolution operator is constructed from the known filter coefficients in `filtsp`. Since the two linear operators are applied sequentially, we can chain them, which is a nice feature of CLOP. The class `fopChain` concatenates those two operators into a single operator `myop`. The chained operator is treated like any other linear operator. We need not write any new operator but can just concatenate to solve the problem.

Now we have `resid`, `myop`, and `data` prepared in the code, which correspond to $\mathbf{y}$, $\mathbf{A}$, and $\mathbf{x}$, respectively, in $\mathbf{y} = \mathbf{A}\mathbf{x}$. The standard Hestenes solver routine uses CG to solve for the missing data. Here the maximum iteration number `niter` is set to be 10 to generate the Figures 8.1 through 8.4.

Using the optimization program `miss1()`, values are found to replace the missing m values so that the power in the filtered data is minimized. Figure 8.1 shows interpolation of the dataset with $1 - Z$ as a roughening filter. The interpolated data match the given data where they overlap.

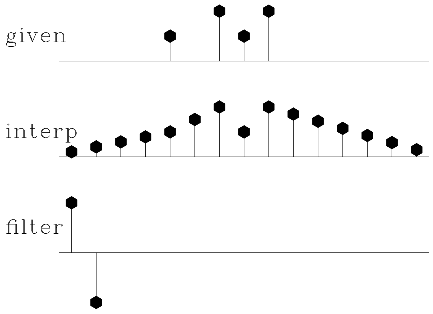

Figure 8.1: The top row shows the given data; the middle, the given data with interpolated values; the bottom,the filter $(1, -1)$ whose output has minimum power. mis-mlines [R]

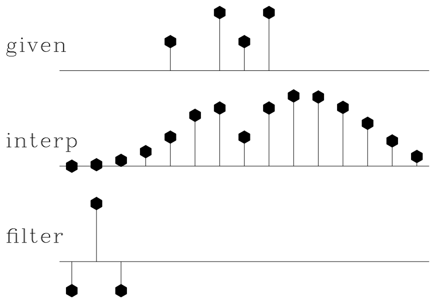

Figure 8.2: The top shows the same input data as in Figure 8.1. The middle is interpolated. The bottom shows the filter $(-1, 2, -1)$. The missing data seem to be interpolated by parabolas. mis-mparab [R]

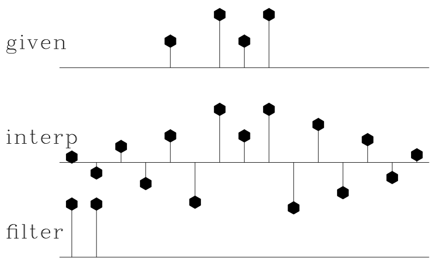

Figure 8.3: The bottom shows the filter $(1, 1)$. The interpolation is rough. Like the given data itself, the interpolation has much energy at the Nyquist frequency. But unlike the given data, it has little zero-frequency energy. mis-moscil [R]

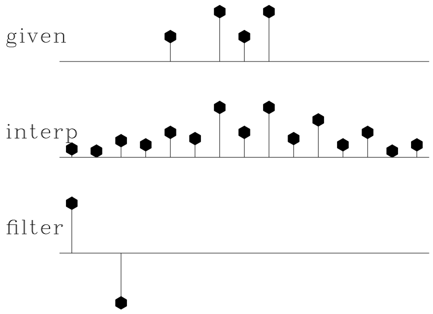

Figure 8.4: The top is the same as in Figures 8.1 to 8.3. The middle is interpolated. The bottom shows the filter $(1, 0, -1)$, which comes from the coefficients of $(1-Z)(1+Z)$. Both the given data and the interpolated data have significant energy at both the zero and the Nyquist frequencies. mis-mbest [R]

Figures 8.1–8.3 illustrate that the rougher the filter, the smoother the interpolated data, and vice versa. Let us turn our attention from the residual spectrum to the residual itself. The residual for Figure 8.1 is the *slope* of the signal (because the filter $1-Z$ is a first derivative), and the slope is constant (uniformly distributed) along the straight lines where the least-squares procedure is choosing signal values. So these examples confirm the idea that the least-squares method abhors large values (because they are squared). Thus, least squares tend to uniformly distribute residuals in both time and frequency to the extent that the constraints allow.

So far, we have concentrated on interpolating the missing data with a known filter. That is a linear problem. However, in real life, it is hard to determine the best filter. We usually have to treat it as an unknown and solve another optimization problem to determine which filter to use. Now the problem becomes a nonlinear one, which the next section discusses.

8.2 INTERPOLATING MISSING DATA WITH AN UNKNOWN FILTER

The material below is an attempt to reproduce the results of the missing data chapter in PVI. In Figures 8.1 through 8.4, the filters are taken as known, and the only unknowns are the missing data. Now, instead of having a predetermined filter, let us solve for the filter as well as for the missing data. The principle we need to use is that the output power is minimized while the filter is constrained to have one nonzero coefficient (otherwise all the coefficients would go to zero).

8.2.1 Packing both the missing data and the filter into a CG vector

First, it is important to examine the theory and coding behind the Figures 8.5 through 8.8. Initially, we define a roughening filter $\mathbf{f}$ and a data signal $\mathbf{d}$. The regression is $0 \approx \mathbf{f} * \mathbf{d}$ where the filter $\mathbf{f}$ has at least one coefficient constrained to be nonzero and the data contain both known and missing values. Suppose that the perturbations are $\Delta\mathbf{f}$ and $\Delta\mathbf{d}$. We neglect the nonlinear term $\Delta\mathbf{f} * \Delta\mathbf{d}$ as follows:

$$
\begin{aligned}
0 &\approx (\mathbf{f} + \Delta\mathbf{f}) * (\mathbf{d} + \Delta\mathbf{d}) &(8.6)\\
0 &\approx \mathbf{f} * \mathbf{d} + \mathbf{d} * \Delta\mathbf{f} + \mathbf{f} * \Delta\mathbf{d} + \Delta\mathbf{f} * \Delta\mathbf{d} &(8.7)\\
-\mathbf{f} * \mathbf{d} &\approx \mathbf{d} * \Delta\mathbf{f} + \mathbf{f} * \Delta\mathbf{d} &(8.8)
\end{aligned}
$$

Again let's take the mask operator to separate the known values from the missing part of the data, defining $\mathbf{M}_f$ as a mask operator for the unknown filter and $\mathbf{M}_d$ for the missing data. Then,

$$
\begin{aligned}
\mathbf{f} &= \mathbf{M}_f\mathbf{f} + (\mathbf{I} - \mathbf{M}_f)\,\mathbf{f} &(8.9)\\
\mathbf{d} &= \mathbf{M}_d\mathbf{d} + (\mathbf{I} - \mathbf{M}_d)\,\mathbf{d} &(8.10)
\end{aligned}
$$

For simplicity, we can define

$$
\begin{aligned}
\mathbf{f}_k &= (\mathbf{I} - \mathbf{M}_f)\,\mathbf{f} &(8.11)\\
\mathbf{d}_k &= (\mathbf{I} - \mathbf{M}_d)\,\mathbf{d} &(8.12)
\end{aligned}
$$

where $\mathbf{f}_k$ is the known part of the filter and $\mathbf{d}_k$ the known part of the data at some stage of interpolation. Thus, the perturbation terms $\Delta\mathbf{f}$ and $\Delta\mathbf{d}$ in equation (8.8) correspond to $\mathbf{M}_f\mathbf{f}$ and $\mathbf{M}_d\mathbf{d}$, respectively. Now the problem can be rewritten with the mask operators as follows:

$$
-\mathbf{f}_k * \mathbf{d}_k = \mathbf{d}_k * (\mathbf{M}_f\mathbf{f}) + \mathbf{f}_k * (\mathbf{M}_d\mathbf{d}) \tag{8.13}
$$

Rewriting the above equation in matrix form yields

$$
-\mathbf{f}_k * \mathbf{d}_k = \begin{pmatrix} \mathbf{f}_k* & \mathbf{d}_k* \end{pmatrix} \begin{pmatrix} \mathbf{M}_d & 0 \\ 0 & \mathbf{M}_f \end{pmatrix} \begin{pmatrix} \mathbf{d} \\ \mathbf{f} \end{pmatrix} \tag{8.14}
$$

where $\mathbf{f}_k*$ represents convolution with $\mathbf{f}_k$ and $\mathbf{d}_k*$ with $\mathbf{d}_k$.

Now we have a linearized problem exactly in the form of $\mathbf{y} = \mathbf{A}\mathbf{x}$, where

$$
\mathbf{A} = \begin{pmatrix} \mathbf{f}_k* & \mathbf{d}_k* \end{pmatrix} \begin{pmatrix} \mathbf{M}_d & 0 \\ 0 & \mathbf{M}_f \end{pmatrix} \tag{8.15}
$$

and

$$
\mathbf{x} = \begin{pmatrix} \mathbf{d} \\ \mathbf{f} \end{pmatrix} \tag{8.16}
$$

We solve the equation (8.14) for the unknown $\mathbf{x}$ using Hestenes solver, as in the preceding section. Then, $\mathbf{d}_k$ and $\mathbf{f}_k$ are updated and solved again, and so on.

All these jobs can be nicely done in CLOP by using `spacearray` and `oparray` classes. In the program `missif()`, `x` is defined as `floatspacearray`, which is an array of `floatspace` **d** and `floatspace` **f**. In the same manner, we build the `fopArray` class of the convolution with the data and with the filter, say ($\mathbf{f}_k *\quad \mathbf{d}_k *$). One of the interesting things about using the `fopArray` class is that applying it to `spacearray` class works the same way as the matrix multiplication. Thus, we have to make sure that all the products $\texttt{fopArray}_{(\texttt{i,j})} \cdot \texttt{spacearray}_{(\texttt{j,k})}$ have the same dimension to add them together. For example, $\texttt{op1} \cdot \texttt{space1}$ should be conformable with $\texttt{op2} \cdot \texttt{space2}$ in equation (8.17) to make the operation work as follows:

$$\begin{pmatrix} \texttt{op1} & \texttt{op2} \end{pmatrix} \begin{pmatrix} \texttt{space1} \\ \texttt{space2} \end{pmatrix} \quad = \quad \begin{pmatrix} \texttt{op1} \cdot \texttt{space1} + \texttt{op2} \cdot \texttt{space2} \end{pmatrix} \tag{8.17}$$

The second matrix of the mask operators in equation (8.14) are created by `fopDiagonal` class which takes operators and creates a square matrix with them in the diagonal.

```
//   missif.cc : 1-D missing data interpolation with unknown filter
#include "missif.h"
floatspacearray missif(const floatspace & f, const floatspace & d,
                                             int lag, int niter ) {
        floatspace data = d, filt = f;
        floatArray ff   = filt.getarray();
        Axislist   dax  = data.getaxislist();
        Axislist   fax  = filt.getaxislist();
        floatspacearray x(2,1),resid;
        x(0,0) = data;                         // array of data and filter
        x(1,0) = filt;

        if( ff(lag - 1) == 0.)
                seperr( "Error: filter(lag) = 0. \n");

        int * mask = new int[fax.list[0].length];
        for( int i=0; i < fax.list[0].length ; i++)
                if ( i != lag-1 )
                        mask[i] = 1;           // mask for filter

        fopDiagonal mymask(2);                 // diagonal array of Mask op
        fopMask     dmask(dax.list[0],data);
        fopMask     fmask(fax.list[0],mask);
        mymask.set(0,dmask);
        mymask.set(1,fmask);

        for( int iter=0; iter < niter;  iter++){    // outer nonlinear loop
                                                    // update Operators
                fopContran * convf  = new fopContran(filt,0);
                fopContran * convd  = new fopContran(data,0);
                fopArray   * myconv = new fopArray(1,2);
                myconv->set(0,0,*convf);
                myconv->set(0,1,*convd);
                fopChain   * myop   = new fopChain(*myconv, mymask);

                resid = -(convf->Forward(data));    // make residual
                x = fophestenes(x, resid, *myop, 2);// Hestenes Linear Solver
                data += x(0,0);                     // update unknowns
                filt += x(1,0);

                delete convf; delete convd; delete myconv; delete myop;
        }
        delete mask;
        return x;
}
```

Two convolutions are combined into one `fopArray myconv(1,2)` operator, and two masking operators into `fopDiagonal mymask(2,2)`. Then we're ready to make a single operator $\mathbf{A}$, `myop` in the code by using `fopChain` to plug into the solver routine. Having built the operator , we only need to perform two basic loop calculations. First we linearize the problem and solve it, using the Hestenes solver in the inner iteration loop. Then we do the iteration and update the residual and the operator in the outer one. The residual is recalculated inside the outer nonlinear loop to avoid the accumulation of errors that would result from neglecting the nonlinear product $(\mathbf{M}_f\mathbf{f}) * (\mathbf{M}_d\mathbf{d})$.

In the program `missif()`, we need to determine two integers for the number of iterations in the inner loop and in the outer one. I have chosen 2 for the inner linear Hestenes solver to avoid breaking the linearity assumption by making a big change from the previous unknown. Otherwise, the output becomes unstable which results in diverging interpolated data. For the outer nonlinear loop, 22 is used.

Figures 8.5 through 8.8 show the interpolated data and filter made from `missif()`. In Figure 8.5 the filter is constrained to be of the form $(1, a_1, a_2)$. The result is the same as in PVI. The filter comes out slightly different from the $(1, 0, -1)$ in Figure 8.4, which is based on a subjective analysis. Constraining the filter to be of the form $(a_{-2}, a_{-1}, 1)$ in Figure 8.6 yields the same interpolated missing data as in Figure 8.5, as is the case in PVI. The next two figures show the interpolation for synthetic data consisting of a fragment of a damped exponential, and off to one side of it an impulse function. Most of the energy is in the damped exponential. Figure 8.7 shows that the spectrum and the extended data are about what we would expect. From the extrapolated data, it is impossible to see where the given data end. In Figure 8.8 the filter is constrained in the middle, which produces output with a much shorter

Figure 8.5: The top row shows known data. The middle includes the interpolated values. The bottom is the filter with the leftmost point constrained to be unity and other points chosen to minimize output power. mis-missif [R]

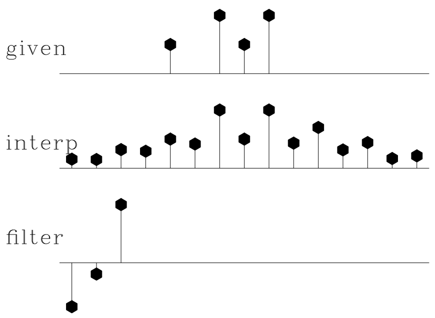

Figure 8.6: The filter here had its rightmost point constrained to be unity—i.e., this filtering amounts to backward prediction. The interpolated data seem to be identical, as with forward prediction. mis-backwards [R]

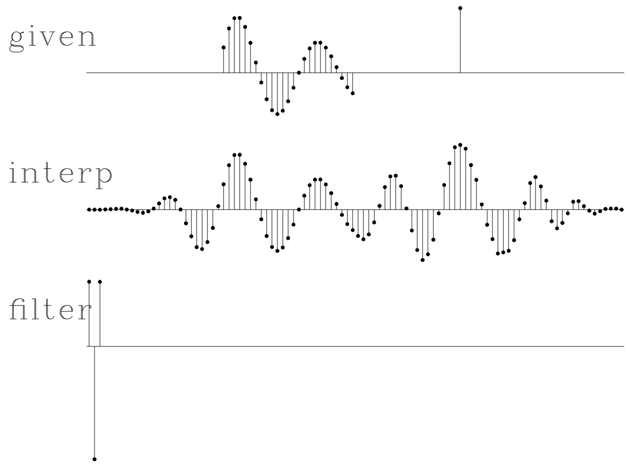

Figure 8.7: The top row shows synthetic data with the missing data represented by zeros. The middle includes the interpolated values. The bottom is a prediction-error filter. The filter may look symmetric but is not quite. mis-exp [R]

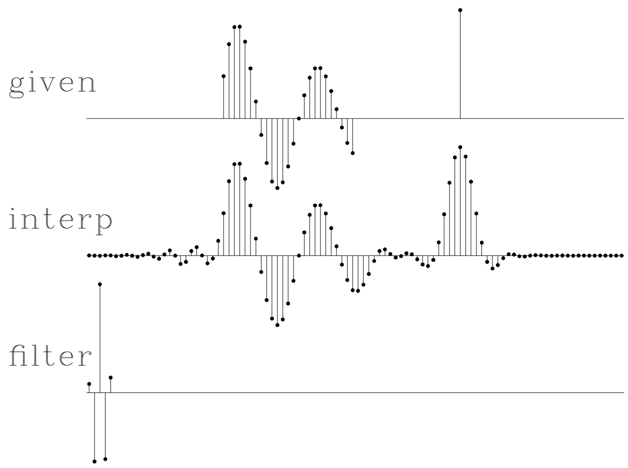

Figure 8.8: The top row shows synthetic data with the missing data represented by zeros. The middle includes the interpolated values. The bottom is the filter, an interpolation-error filter. mis-center [R]

wavelength than in Figure 8.7. The boundary between real data and extended data is not nearly as well hidden as in Figure 8.7.

8.3 SUMMARY

Missing data can be restored by using the theorem that the output has minimum energy after specified filtering. The roughening filter can either be guessed or optimized. The former is a linear problem and the latter a nonlinear problem.

For the nonlinear problem, I have used a different scheme from Claerbout's. My code first linearizes the problem, then solves it explicitly and iterates. In contrast, the code in PVI hides the linear solving step inside the nonlinear iteration loop. The two schemes yield different results. The interpolated output from my program is very dependent on the number of iterations in the inner linear loop. The results can be improved by using a smaller number of iterations. For a larger number, such as five or more, the solution diverges. As mentioned earlier, it seems that a large number of iterations violates the linearity assumption. For now Claerbout's method seems to converge faster and gives a little better result when used for interpolating missing data. However, an interesting question remains as how to find in general the best scheme for a given nonlinear optimization problem.

Finally, coding the missing data problems in C++ is a pleasantly efficient process. We can simply use existing objects in the CLOP library. The programs in the preceding two sections look almost the same as the mathematics itself, in contrast to the verbose Fortran expressions for handling variables. Furthermore, we can separate the problem into two steps, making necessary operator objects and passing them to the solver routine, which can be designed by those who do professional numerical analysis.

Chapter 9

Appendix: Man Pages

9.0.2 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,  ori-
          gin, 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
```

9.0.3 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 dimen-
          sion 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.

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

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

9.0.4 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

ByteConverter(UI)    Cube Reference Manual    ByteConverter(UI)

          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

ByteConverter(UI)    Cube Reference Manual    ByteConverter(UI)

     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
```

9.0.5 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 allo-
          cate the disk space. If a DataStore is given  the  data
          are copied from the input data store.

SEE ALSO
     InterViews, intro
```

9.0.6 Man page: op

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

NAME
     <type>op - A virtual base class for linear operators operat-
     ing 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  func-
     tion,   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 func-
     tions. 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
```

9.0.7 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
```

9.0.8 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
```

9.0.9 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
```

9.0.10 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
```

9.0.11 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
```

9.0.12 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
```

9.0.13 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; oth-
     erwise, the generator is unseeded.

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

9.0.14 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
```

9.0.15 Man page: operator

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

NAME
     <type>operator - A virtual base class for all linear  opera-
     tors 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 func-
     tion  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

9.0.16 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
```

9.0.17 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
```

9.0.18 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
```

9.0.19 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
```

9.0.20 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

9.0.21 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
```

9.0.22 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
```

9.0.23 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
```

9.0.24 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
```

9.0.25 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
```

9.0.26 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
```

9.0.27 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
```

9.0.28 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
```

9.0.29 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,

9.0.30 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

9.0.31 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
```

9.0.32 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
```

9.0.33 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 SepInfo data structure. A SepData object allows operation such as Window, Merge, Pad, Copy. Each of these operations modifies the description of the data structure. These operations 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 structure.

```
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 contained into this, the data in the input object have priority over this data.
All these structure dimension are made on the coordinates of the input data SepInfo object. To add traces to plane the origin of the trace must reflect its position 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 overritten 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 begin-
          ning 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  overwrit-
          ten  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.
```

9.0.34 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  compati-
          ble with the current object. i.e. if they have compati-
          ble 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
```

9.0.35 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  posi-
          tion 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 tem-
          porary 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
```

9.0.36 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  com-
     mand is created with the unix "popen" routine.

SepOutput(UI)        Cube Reference Manual        SepOutput(UI)

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 con-
          tains 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  destruc-
          tor  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.
```

9.0.37 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
```

9.0.38 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

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

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
```

9.0.39 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

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

          <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
```

Index