Stanford Exploration Project, Report 75, November 17, 1997, pages 487–??

Stanford Exploration Project, Report 75, November 17, 1997, pages 487–??

The C++ language in physical science

Dave Nichols, Geoff Dunbar, and Jon Claerbout[1]

keywords: *linear, modeling, arrays, optimization, algorithm, velocity*

ABSTRACT

We have designed C++ classes that are used to solve optimization problems involving linear operators. The use of C++ enables us to write code to solve these problems at a much higher level than when writing in Fortran. The ability to design classes that abstract the concepts of operators and spaces enables us to separate the task of writing new operators from the task of writing routines to solve the optimization problem.

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. AT&T provides a free program that converts Fortran to C. (There is no known program for the reverse, nor is there ever likely to be.) 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 (especially for input-output), by using the Ratfor preprocessor to improve syntax, and by using saw and sat preprocessors for memory allocation and parameter handling. In the past few years, C has become standardized, and new developments are centered on the C++ language (Stroustrup, 1986). C++ shares the

[1]**email:** not available

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.)

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 (NA) in the conjugate-gradient solver. In Fortran a new solver routine must be written for every new problem. One person (the programmer) must handle all the ingredients of both physics and NA 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.

The C++ language should allow us to better isolate numerical analysis from the physical-science code. If this promise can be realized, advanced techniques of numerical analysis should be more accessible to end users in science and engineering. 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, we took a large body of tutorial Fortran-Ratfor code in the geophysical textbook, "Earth soundings analysis: processing versus inversion" by Jon Claerbout and began converting it to C++.

The C++ language

To ameliorate the problem of many of the readers of this document knowing little of C and C++, we introduce concepts of C and C++ as slowly as we can. A recommended learning pathway is to 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 (Sessions, 1992) which concerns itself mainly with the "class" and "inheritance" concepts, avoiding the full minutia of the C++ language.

Going beyond Fortran, 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), i1=1,n1), i2=1,n2)`. A numerical analyst thinking about solving the multivariate regression $\mathbf{y} \approx \mathbf{A}\mathbf{x}$ 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 and has elaborate provisions for sharing and hiding data, for defaulting and overriding functions. C++ 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.

DEFINITIONS

Linear operators can be regarded as matrices. We do not handle them 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,

$$\mathbf{A}\mathbf{x}.$$

The conjugate subroutine computes

$$\mathbf{A}^{\dagger}\mathbf{y},$$

where $\mathbf{A}^{\dagger}$ is the (complex conjugate) transpose matrix.

When we solve a problem involving the operator both the operator and its conjugate 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 the appendix. Thus the coding of any operator is not finished until a dot-product test has been passed.

All our operators will be defined by the parameters used in creating the particular instance of an operator and the two procedures that apply the forward and conjugate transforms. For instance, a convolution operator is passed an array defining the filter in its constructor. The forward procedure convolves with the filter and the conjugate procedure correlates with the filter. In our C++ code the base operator class " < `type` > `op`" defines a single operator that is designed to be applied to a single 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 space. The word "< `type` >" defines the numerical type of the elements of the space, currently `floatspace` and `complexspace` are supported space types.

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.

An object of class "< `type` > `oparray`" is an array of < `type` > `op` objects that is designed to be applied to an object of class < `type` > `spacearray`. An array of operators is a generalization of the concept of operator, it is analogous to the idea of partitioned matrices. It differs in that the partitions in the operator array are procedurally defined. Our C++ classes define space-arrays and operator-arrays as new C++ classes.

SPACE

An object of class < `type` > `space`, the class to represent a physical space, has two parts. The first part is a multi-dimensional array of numbers representing the sampled data or model. This array can contain floating point values or complex numbers. The second part is a set of axes that describe the physical dimensions of the data or model space.

Every dimension of a < `type` > `space` 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 < `type` > `space` is meant as a collection of real physical data, rather than an abstract collection of numbers. This is ideal for scientific applications, as in geophysics.

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

- One or more constructors.
- A destructor.
- norm(), returns the L_2 norm of the space.

- norm2(), returns the square of the L_2 norm.
- cross(), takes the cross product with another (conforming) space.
- extract(), extracts a sub-space. This function takes a set of parameters that define the sub-space to be extracted.
- insert(), insert a sub-space into a space. This function takes a variety of parameters. It overwrites the existing sub-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 dataformat consists of two parts, an ASCII header file that contains the parameters describing the dataset, and a data file containing the actual data. The following paragraphs taken from PVI describe the basic parameters defined in a header file.

   A cube has three axes. The number of points on the 1-axis is `n1`. A Fortran declaration of a cube could be `real mydata(n1,n2,n3)`. For a plane, `n3=1`, and for a line, `n2=1`. In addition, many programs take "1" as the default for an undefined value of `n2` or `n3`. The physical location of the single data value `mydata(1,1,1)`, like a mathematical origin (o_1, o_2, o_3), is denoted by the three real variables `o1`, `o2`, and `o3`. The data-cube values are presumed to be uniformly spaced along these axes like the mathematical increments $(\Delta_1, \Delta_2, \Delta_3)$, which may be negative and are denoted by the three real variables `d1`, `d2`, and `d3`.

   Each axis has a label, and naturally these labels are called `label1`, `label2`, and `label3`. Examples of labels are `kilometers`, `sec`, `Hz`, and `"offset, km"`. Most often, `label1="time, sec"`. Altogether that is $2 + 3 \times 4$ parameters, and there is an optional title parameter that is interpreted by most of the plot programs. An example is `title="Yilmaz and Cumro Canada profile 25"`.

   These parameters are used to initialize the axes of a < `type` > `space` and the datafile is read to initialize the matrix. Building a < `type` > `space` from a SEP cube dataset is a two stage process. First the input dataset is initialized by creating a "`SepInput`" object and then the < `type` > `space` is constructed using the `SepInput` object as a parameter. The SepInput class is a C++ class that encapsulates interaction with a SEP cube dataset (Dulac and Nichols, 1989).

```
// Declare a pointer to a SepInput object and assign to it a new object
// that is constructed by reading the seplib header supplied on stdin.
```

```
SepInput * inData = new SepInput( ''stdin'' ) ;

// Declare and create an instance of a floatspace object, called myspace.
// The object is created from the whole of the previously defined
// input dataset.
floatspace myspace(inData);
```

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 spaces. 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 are written in such a way that they have more 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.

At the moment the matrix classes we use are the M++ classes from Dyad Software Co. (1991). The reason that, in later examples, the type of the elements appears in the space definition is that M++ does not support templated classes. In the future we hope to use a new public domain matrix class that supports templates so that all our code will be written in a manner that is independent of the type of the matrix elements.

OPERATOR

A linear operator 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.

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 Conjugate() function that is applied to a data space to produce a model space.

Using the summation convention, here are some familiar geophysical operations. The order of subscripts of an operator is defined by the axis mappings that the forward operator applies. e.g. $A_{x\,v}$ maps the x axis to a v axis.

Convolution along the time axis maps the time axis to a new time axis (let us call it t'). The forward convolution operator can be defined by:

$$\mathbf{y}_{t'} = \mathbf{C}_{t\,t'}\ \mathbf{x}_t$$

The convolution operator $\mathbf{C}_{t\,t'}$ maps the t axis to a t' axis. The conjugate operator, a correlation, is maps the t' axis to a t axis:

$$\tilde{\mathbf{x}}_t = \mathbf{C}^{\dagger}{}_{t\,t'}\ \tilde{\mathbf{y}}_{t'}$$

The NMO modeling operator, $\mathbf{N}$, maps a trace from zero offset to any given offset, x.

$$\mathbf{u}_{xt} = \mathbf{N}_{t_0\,xt}\ \mathbf{v}_{t_0}$$

This operator adds offset, x, dependency to the trace. In contrast the conjugate operator removes the offset dependency of the trace.

$$\tilde{\mathbf{v}}_{t_0} = \mathbf{N}^{\dagger}{}_{t_0\,xt}\ \tilde{\mathbf{u}}_{xt}$$

Note that this operator is applied to a single trace. It is not the same as the NMOStack operator for which the forward operation models to a range of offsets and the conjugate operation stacks over offset.

Our operators are procedural operators. That is, every operator has, at its heart, a procedure which tells it what to do with the space it is being applied to. 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 basic operator class. Any class that is derived from the base operator class inherits this ability without the author having to write any code to implement it.

In many cases the axis definitions are basic to the definition of the operator, so this strictly defined axis compatibility is necessary. e.g. a velocity analysis operator needs to know both the dimensions of the offset-time space that the forward operator should model to, and the dimensions of the velocity-time space that the conjugate operator creates.

In the current implementation you must specify all the attributes of the axes that an operator is to be applied to. This is not always necessary; a convolution operator only needs to match the axis label and sample rate. It can be applied to any length of trace stating at any initial sample. In the future we intend to implement a more liberal scheme for matching the axis definitions of operators and spaces.

Here is an example of constructing an operator. The "`floatConvolve`" operator class performs convolution as its Forward function.

```
\\ Declare and initialize an instance of the floatConvolve class
floatConvolve op(axis,filter);
```

Where `axis` is an `Axis` object, that defines the axis to convolve along, and `filter` is a `floatArray` that defines the filter to convolve with. Here is the code to apply this operator to a `floatspace`, `x`, to produce an output `floatspace`, "`result`".

```
floatspace result = op.Forward(x);
```

If the `floatspace` x, does not have an axis that matches the axis that the operator is designed to be applied to an error will be generated at run time.

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

$$\mathbf{z} = (\mathbf{AB})\mathbf{x}$$

and we must regard it as

$$\mathbf{z} = \mathbf{A}(\mathbf{Bx}).$$

Our Fortran programs interpret this as meaning that first memory must be allocated for $\mathbf{y} = \mathbf{Bx}$; $\mathbf{y}$ must be erased; $\mathbf{y} = \mathbf{y} + \mathbf{Bx}$ should be computed; $\mathbf{z}$ should be erased; and finally $\mathbf{z} = \mathbf{z} + \mathbf{Ay}$ 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:

```
<type>opchain C = A * B
```

The class “$<$ `type` $>$ `opchain`” 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.Conjugate(y) ≡ B.Conjugate( A.Conjugate( y ) )
```

When a chained operator is applied in the conjugate sense it will be applied with each component operator conjugated and applied in the reverse order. The code that implements chained operators does not have to be rewritten for new operators, this removes another possible source of errors in writing new code.

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

```
<type>opconj B = conjugate(A)
```

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

ARRAYS

In addition to the basic $<$ `type` $>$ `space` and $<$ `type` $>$ `op` classes, there are also classes that define arrays of each. A “$<$ `type` $>$ `spacearray`” is a two dimensional array of $<$ `type` $>$ `spaces`, which are not necessarily conformable with each other. An operator array is also a two dimensional array which contains operators. Applying a $<$ `type` $>$ `oparray` to a $<$ `type` $>$ `spacearray`:

$$\mathbf{u} = \mathbf{Av}$$

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:

```
<type>spacearray 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 < `type` > `spacearray` and the operator `B` is of class < `type` > `oparray`.

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:

$$\min_x \left| \begin{pmatrix} \mathbf{A} \\ \epsilon\mathbf{I} \end{pmatrix} \mathbf{x} - \begin{pmatrix} \mathbf{y} \\ \mathbf{0} \end{pmatrix} \right|^2 \tag{1}$$

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 conjugate is applied to a space-array it gives a space,

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

All of this is handled automatically by the < `type` > `oparray` class.

The following routine gives the damped least squares solution to *any* linear problem that can be posed in terms of `opa`, an instance of the class < `type` > `op`, and a RHS, `y`, of the class < `type` > `space`.

```
// This routine gives a damped solution to the least squares problem
// opa.Forward( x ) = y;
// by augmenting the operator with a diagonal damping matrix

<type>space damped_solver( <type>space& x0,   // initial solution
                           <type>op opa,      // linear operator
                           <type>space& y,    // rhs
                           float epsilon,     // the damping factor
                           int maxiter )      // maximum no. of iterations
{
  <type>oparray ops(2);            // declare a 2x1 operator array
  <type>Scale   scale( epsilon );  // scale is a diagonal scaling operator
  ops(1) = &opa; ops(2) = &scale; // assign the elements of the operator array

  <type>spacearray rhs(2)          // declare a 2x1 rhs array of spaces.
  <type>space zero = x.empty()     // zero is the same shape as x but all zeros.
  rhs(1) = y; rhs(2)=zero         // assign the elements of the rhs

  // now solve the augmented problem, and return the result
  return <type>space soln = <type>arraysolver( x0, ops, rhs, maxiter );

}
```

The new operator `ops` (of class < `type` > `oparray`) and right hand side, `rhs` (of class < `type` > `spacearray`) are used to solve the damped least squares problem. This is done by the routine < `type` > `arraysolver`, which is one of a family of solver routines that are available.

SOLVERS

The "`damped_solver`" routine presented above calls the routine < `type` > `arraysolver`. This routine could solve the problem by any of a wide number of methods. 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.

The 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. Below is a Hestenes conjugate gradient solver to solve the equation $\mathbf{y} \approx \mathbf{A}\,\mathbf{x}$ for $\mathbf{x}$, where $\mathbf{y}$ is a known $<$ `type` $>$ `space` and $\mathbf{A}$ is of class $<$ `type` $>$ `op`. Note that the only $<$ `type` $>$ `space` member functions used here are the functions that scale by a float, add another $<$ `type` $>$ `space` and take the norm-squared.

```
// The hestenes routine is passed an initial output space "x"
// the rhs space "y", the operator "A" and the maximum number of
// iterations to use.
// It returns an estimate of the solution.
// N.B. this routine does not use the initial solution other than
// using it to define the model space.

<type>space <type>hestenes(<type>space x,
                           <type>space & y,
                           <type>op & A,
                           int maxiter ) {

  <type>space step,cstep,grad,resid;
  float gamma,gamma1,alpha,beta,ynorm;

  x = 0;
  ynorm = y.norm2();
  resid = y;
  grad = A.Conjugate(resid);
  step = grad;
  gamma1 = step.norm2();

  int i=0;

  while ((i++<maxiter)  && (resid.norm() / ynorm > .00001)  ) {
     cstep = A.Forward(step);
     alpha = gamma1 / cstep.norm2();
     x += step * alpha;
     resid = resid - cstep * alpha;
     grad = A.Conjugate(resid);
     gamma = grad.norm2();
     beta = gamma / gamma1;
     gamma1 = gamma;
```

```
      step = step * beta + grad;

   }
   return x;
}
```

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 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.

WRITING NEW OPERATORS

To write an operator, one must write the constructors, which initialize the operator, the destructor (which only needs to be written if something is dynamically allocated) and a ":: `dodata`()" member function to do the actual calculation.

Here is the header file for the convolution operator class, which define the data members and member functions of the operator:

```
class floatConvolve : public floatopone {
public:
    // Construct a floatConvolve Operator given a floatArray
    // filter, and an Axis to operate on
  floatConvolve(floatArray &,Axis &);
private:
     void dodata(floatArray &,floatArray &,int);

     floatArray filter;
};
```

Note that the only new public function is the constructor, which takes an array of filter coefficients and an axis specification as parameters.

Here is a constructor for the convolution operator:

```
floatConvolve::floatConvolve(floatArray & f,Axis & a) {
    inaxis = (Axislist) a;
    outaxis = inaxis;
```

```
    outaxis.list[0].length += (::rows(f) - 1);
    filter = f;
}
```

Note that the output axis length is set to be shorter than the input axis length. This convolution operator truncates its output.

No destructor is necessary, because nothing is dynamically allocated, so everything is deallocated automatically.

Finally, the dodata routine, to do the calculation for the operator:

```
void floatConvolve::dodata(floatArray & x,floatArray & y,int conj) {
   for (int ib=0; ib<(::rows(filter)); ib++) {
      for (int ix=0; ix<inaxis.list[0].length; ix++) {
         if (conj) {
            x(ix) += y(ib+ix) * filter(ib);
         } else {
            y(ib+ix) += filter(ib) * x(ix);
         }
      }
   }
}
```

The base class for operators is written so that the dodata function of a new operator is passed two arrays. One matches the axes specified by the `Axislist` object `inaxis` and the other matches the the axes specified by `outaxis`. The routine is also passed a flag that specifies whether the operator is to be applied in the forward or conjugate sense.

If there are more axes present in the input space the `dodata` routine will be called repeatedly, once for each subspace that matches the operator's axes. This means that a 1-D operator only requires code dealing with 1-D arrays, all the other dimensions are handled by the base operator class.

BUILDING BLOCKS

Many new operators can be built by chaining and compositing existing operators. The user does not need to write a new operator class. The following basic operators have already been written.

Operators currently available	
Forward operation	**Conjugate operation**
Convolution	Correlation
Padding	Truncation
Selection	Expansion
Masking	Masking
Scaling	Scale by conjugate
Nmo modeling	Nmo
Multi offset nmo modeling	Nmo stack
Shift (delay)	Shift (advance)
1-D linear interpolation	1-D linear spreading
Velocity analysis (vspray)	Multi velocity modeling
Matrix multiply	Multiply by conjugate matrix

FUTURE WORK

Much work remains to be done and we look forward to collaborating with many other authors by working within the same C++ class framework. Some of the remaining tasks are:

- More operators need to be written, especially ones dealing with complex data.
- Some of the current syntax is clumsy because we do not have a templated matrix class. This will be corrected when a public domain templated matrix class becomes available.
- We need a simple operator example that encapsulates a pair of fortran routines that apply an operator and its conjugate. This will help fortran programmers to migrate their existing code to the new framework.
- We need numerical analysts to write better solver routines and to help us design a preconditioned solver routine so that we can experiment with various types of preconditioning.

PRACTICAL USE

Our framework has been used to rewrite many routines from the book, PVI. During the rewriting process the rigorous error checking built into the C++ code revealed an error in the book in a Fortran solver for one operator. Although the simple routines were of a similar complexity to the original Fortran routines, the more complicated problems that involve combining existing operators in chains and array were much simpler to write in C++. We intend to continue this work and we intend to solve more complicated problems that discourage coding in Fortran.

CONCLUSIONS

We have designed a C++ class framework for handling optimization problems involving linear operators. The use of C++ has enabled us to separate the task of writing procedural operators from the task of writing routines to solve the optimization problems. This separation was not possible in Fortran and we were limited in the type of problems we could solve in Fortran because of the growing complexity of the code.

The encapsulation of the concepts of linear operators and spaces in concrete C++ classes enable authors to deal with problems at a much higher level than in Fortran. The error checking built into the base operator and space classes is a valuable aid to writing new code and prevents many of the trivial book-keeping errors that occur when writing in Fortran.

AVAILABILITY AND STATUS

If you are reading this from a CD-ROM the code that we have written will be on the CD. This is not a finished product, the code is still the subject of ongoing development. The current space classes are based on the M++ array classes. M++ is a commercial product and therefore not present on the CD. M++ is available from Dyad Software Co., 16950 151st Ave. SE, Renton WA 98058. (206)-228-3170. When a suitable public domain array class becomes available we intend to use it in place of M++ so that we can redistribute complete, working code.

ACKNOWLEDGMENTS

Much of the design of the C++ classes developed from regular discussions with Mike Grant and Lester Dye (the author of M++) both of whom have contributed greatly to our understanding of C++ in general and the design of array classes in particular.

REFERENCES

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

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

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

Sessions, R., 1992, Class construction in c and c++: object-oriented programming fundamentals: Prentice Hall.

Stroustrup, B., 1986, The C++ programming language: Addison Wesley Publishing Co.

APPENDIX A

The dot product test

To test whether a program pair behaves as a linear operator should, we do a dot-product test. The current working version of PVI says:

> The dot-product test is a simple test for verifying that the two procedures are conjugate to each other.
>
> The associative property of linear algebra says that we do not need parentheses in a vector-matrix-vector product like $\mathbf{y}^\dagger\mathbf{A}\mathbf{x}$ because we get the same result no matter where we put the parentheses. They serve only to determine the sequence of computation. Thus,
>
> $$\mathbf{y}^\dagger(\mathbf{A}\mathbf{x}) = (\mathbf{y}^\dagger\mathbf{A})\mathbf{x} \tag{A-1}$$
> $$\mathbf{y}^\dagger(\mathbf{A}\mathbf{x}) = (\mathbf{A}^\dagger\mathbf{y})^\dagger\mathbf{x} \tag{A-2}$$
>
> (In general, the matrix is not square.) For the dot-product test, load the vectors $\mathbf{x}$ and $\mathbf{y}$ with random numbers. Compute the vector $\tilde{\mathbf{y}} = \mathbf{A}\mathbf{x}$ using your program for $\mathbf{A}$, and compute $\tilde{\mathbf{x}} = \mathbf{A}^\dagger\mathbf{y}$ using your program for $\mathbf{A}^\dagger$. Inserting these into equation (A-2) should give you two equal scalars
>
> $$\mathbf{y}^\dagger(\mathbf{A}\mathbf{x}) = \mathbf{y}^\dagger\tilde{\mathbf{y}} = \tilde{\mathbf{x}}^\dagger\mathbf{x} = (\mathbf{A}^\dagger\mathbf{y})^\dagger\mathbf{x} \tag{A-3}$$
>
> The left and right sides of this equation will be computationally equal only if the program doing $\mathbf{A}^\dagger$ is indeed conjugate to the program doing $\mathbf{A}$.

Using the C++ classes we can write a single dot product test routine that all new operators can be tested with. We need only supply the operator and sample data and model spaces as arguments. The test routine will generate spaces of random numbers and check that the dot product test condition is satisfied. Here is the code for the version of the routine for `floatop` operators. Equivalent routines are available for complex operators and array operators.