Object-Oriented Design for optimization and inversion software: a proposal

Mark S. Gockenbach *

July 12, 1994

Abstract

The usefulness of mathematics for solving problems of applied science is derived largely from the fact that many seemingly different problems lead to the same mathematical formulation. There is therefore an inherent efficiency in mathematical analysis: the solution of a single mathematical problem can potentially resolve many scientific questions. However, this efficiency is frequently not realized when mathematical software is designed, because abstract mathematical objects such as vectors and linear operators require concrete representations in the computer code. These representations frequently vary from problem to problem, with the effect that the codes must be rewritten when the representations change.

In this paper, we explore the use of Object-Oriented Design principles to overcome the above problem. We propose a scheme for defining abstract classes of vectors and linear operators, and discuss their use in codes for optimization and inversion. The approach presented in this paper is motivated by problems encountered in seismic velocity inversion. The ideas discussed in this paper are currently being implemented in the C++ programming language.

1 Introduction

Mathematical software must represent and manipulate mathematical objects. At the heart of mathematics lies the use of a relatively few abstract mathematical objects to represent many seemingly different kinds of physical and numerical quantities. This can be done because, although the physical quantities may have different form, they behave similarly.

In designing software, it would be desirable to have a mechanism for the uniform treatment of objects that can have more-or-less arbitrary representations. However, common procedural computer programming languages have only a few means of representing the quantities that a program manipulates.

In this paper, I will discuss the use of Object-Oriented Design techniques for writing optimization and inversion software, and in the process, give a (very) brief introduction to the C++

*Department of Computational & Applied Mathematics, P. O. Box 1892, Rice University, Houston, Texas 77251-1892

programming language, which was written to support data abstraction and object-oriented programming. My interest in this topic arises from a desire to develop optimization software to solve seismic inversion problems. I will therefore write the paper from the viewpoint of explaining why traditional procedural programming techniques are not well suited to this endeavor.

The name "object-oriented programming" is derived from the philosophy that a program should be organized around the types of objects which it manipulates, and the operations which can be performed on them. Most of this paper consists of a discussion of a collection of classes of vectors and linear operators, which are the most fundamental objects used in optimization software.

More information about C++ can be found in the book *The C++ Programming Language* (second edition) ([7]) by Bjarne Stroustrup, the designer of the language. This reference also includes a great deal of discussion about object-oriented programming. Examples of the use of both C++ and object-oriented design for optimization can be found in the papers by Meza ([3]) and Nichols et al. ([4]). Another example is given by LAPACK++. LAPACK++ is available on netlib. At the end of this paper, I will discuss how my approach differs from those found in the above references.

2 Vectors

2.1 Traditional programming

Mathematically, a vector space is a collection of "things" together with two operations, addition and scalar multiplication, which can be performed on them. Continuous optimization problems are usually set in a Hilbert space, which adds the notion of an inner product and the associated norm.

The reason I used the word "things" above is that vectors can take many different forms. Ordered pairs of real numbers, continuous functions, the coordinates of a collection of points in 3-space — all of these can be viewed as vectors, given the appropriate definition of the operations mentioned above. Given this fact, writing general software which acts on vectors appears to be an impossible task. However, one soon learns the theorem that all n-dimensional vector spaces are isomorphic to $\mathcal{R}^n$. Since a computer can only deal with a finite amount of data, only finite-dimensional vector spaces can be used in practice anyway, which suggests a way out of this dilemma. All vectors belong to $\mathcal{R}^n$, that is, they are represented by a list of numbers. This is natural in almost every programming language, including FORTRAN, which has (a finite-precision representation of) real numbers and arrays to implement lists.

Unfortunately, although these facts certainly allow the use of FORTRAN and similar languages to solve problems involving vectors, there are difficulties. Consider an optimization problem in which the unknown is a function (of one variable), and suppose the function is represented by samples of its values on a regular grid. The number of samples is the dimension of the underlying vector space. An "off-the-shelf" optimization routine would expect to operate on this dimension along with the array of function values. For example, the calling sequence of the objective function would probably include the dimension and the array. However, the sample rate and the left-endpoint of the grid, along with the number of samples, define the grid, and thus the particular finite-dimensional vector space being used. This information may be needed in the evaluation of the objective function and of the inner product.

To a certain extent, good software will allow the user to include information such as that described above. For instance, a program may allow an array of parameters to be passed to the routine evaluating the objective function. This could contain the grid information in the example above. However, in most cases, it is probably not easy to use an inner product other than the Euclidean inner product, and yet this may be essential in some cases (for instance, if the unknown is a function in a Sobolev space).

In short, FORTRAN provides no mechanism for varying realizations of a vector, nor does it make it easy to change the definition of the operations defined on vectors. This is not to say that it is impossible to address these problems in a procedural language like FORTRAN; indeed, many researchers have done so. However, because procedural languages do not explicitly support data abstraction and object-oriented programming, the code is frequently difficult to write, understand, and maintain.

2.2 Object-oriented design and C++

Now I want to leave behind the limitations imposed by traditional programming languages, and re-consider the question of representing vectors in a computer program. Although vectors may differ in the nature of the data required to specify them, they share common operations: addition, scalar multiplication, inner product. For this reason, vectors fit into the object-oriented philosophy described above: they are objects manipulated by operations with known properties; in other words, without knowing the particular representation, we know how a vector should behave.

An object-oriented programming language will provide a mechanism for *encapsulating* the data and operations that define an object. The data (and its particular representation) are hidden from the programmer, as are the implementations of the operations. However, the programmer can manipulate the object by means of the operations defined on it. Therefore, one can implement an algorithm that operates on vectors using only the operations that act on them. To apply that algorithm to a certain problem, one simply uses the appropriate instance of a vector, whether it be a sampled grid function or an ordinary vector in $\mathcal{R}^n$.

I am going to explain the object-oriented approach more fully by giving fragments of C++ code which implement the ideas. I am going to make no effort to explain the syntax to the point that a reader who does not know C++ could write programs in it; I will, however, try to give a feel for what can be done with the language.

I need to say a few words about some aspects of programming in C++ which may not be familiar to a FORTRAN programmer. C++ is an extension of C, so some (but not all) of what I am about to say will be familiar to those who have programmed in C. Since I am mostly going to discuss the definition and use of objects such as vectors and linear operators, it will not be necessary to know most C++ commands, but it will be important to understand dynamic memory allocation and the way arguments are passed to subroutines.

In C++ (as in C), memory can be allocated dynamically, that is, at run-time. This gives great flexibility and usually results in more efficient use of memory on a shared system. The program can allocate the amount of memory it needs to solve a particular problem. On the other hand, FORTRAN uses static memory allocation, so an array must be declared to be large enough to handle the largest problem that a user will ever want to solve. When the program is used to solve a smaller problem, the extra memory is wasted.

This added flexibility comes at a price: the C++ programmer is responsible for releasing memory when it is no longer needed. In the following description of class definitions, the issue of allocating and deallocating memory will be important.

C++ has three different protocols for passing arguments to functions (incidentally, in C++, as in C, all subroutines are functions). These are:

1. **Call by value**. A local copy of the argument is made, and the function operates on that local copy. Thus the function has access only to the *value* of the argument; it cannot change the argument.

2. **Call by reference**. The function receives a reference to the argument, so that the function operates on the argument just as the calling routine does. This has two implications. First, of course, the function can change the argument. Second, there is no copying involved, which is an important efficiency consideration when the argument represents a large object.

3. **Pointers**. The address of the argument is passed through the call by value mechanism. The function can then "dereference" the address to refer to the contents of the argument. This is used in C programming to simulate call by reference, which is new to C++. The use of pointers is more flexible than the use of references, for reasons I will not explain, so it is still useful in C++.

2.2.1 A simple example of a vector class

In C++, an object is described by means of a class declaration, which really creates a *user-defined type*. If I define a class called `Vec`, then the compiler will allow `Vec`s to be used in more-or-less the same ways that built-in types such as `int` and `float` can be used. Of course, a user-defined type is *not* built-in, which means that the designer of the class must tell the compiler everything about the class, such as how to create objects of that type, how to destroy them (that is, release the memory they use), and what operations can be performed on them.

In Table 2.2.1 is found the declaration of a simple vector class. The purpose of this example is to illustrate the idea of encapsulation, so I will use the simplest type of vector for illustration, a vector in $\mathcal{R}^n$. Later I will present a more sophisticated vector class.

As in declarations given later, the line numbers are not part of the C++ code; I include them so that I can refer to individual lines in my explanations.

The first thing to notice about this declaration is that it is divided into two sections, public and private. The purpose of this division is probably already clear to the reader: a program will manipulate a `Vec` by means of its public functions and operators. The representation of the `Vec` is hidden in the private section. Only those functions which are specifically given access to the private section can manipulate the representation directly. In particular, the *member* functions, both public and private, are given this access.

This idea of hiding the representation and forcing the programmer to manipulate the object by means of the public functions immediately solves one of the problems discussed above. Since

```
1.   class Vec
2.   {
3.      private:
4.         int dim;
5.         double *c;
6.      public:
7.         Vec( int n );
8.         Vec( Vec & );
9.         ~Vec();
10.        void print( );
11.        void get( void )
12.        Vec& operator=( Vec & )
13.        Vec operator*( double a ) ;
14.        Vec operator+( Vec & ) ;
15.  };
```

Table 2.1: A simple vector class

programs will not act directly on the representation of the vector, that representation can change without the need of changing the rest of the program.

I will now explain the rest of the class declaration so that I can later discuss more sophisticated aspects of object-oriented programming and C++. The private section is easy to understand for this simple class; it includes the dimension of the vector and a pointer to a double precision variable. In this class, this pointer would be used to point to the first element in an array containing the components of the vector (the actual array will be allocated dynamically when needed).

As much as possible, I will describe what the functions and operators do without displaying the code, because I do not wish to explain C++ syntax.

2.2.2 Constructors and Destructors

As I mentioned earlier, a user-defined type is to be used as if it were built-in, so the class declaration must tell the compiler several things. First of all, the compiler must know how to create objects. This is the purpose of the *constructor(s)*.

A constructor of a class is a public member function having the same name as the class, and with no return type. The above example has two constructors, on lines 7 and 8. This is a good place to explain one special feature of C++: *function overloading*. In FORTRAN and most traditional languages, a subroutine is identified by its name, so that distinct subroutines must have distinct names. However, in C++, a function is identified by its *signature*, which consists of three parts: the function name, the types of its arguments, and the type of class of which the function is a member (if applicable). Two C++ functions are distinct if they have different signatures; hence the constructors on lines 7 and 8 are distinct functions.

The constructor on line 7 does three things:

1. Create an object of type `Vec`, that is, allocate space for an `int` and a `double *` pointer.

2. Assign the `int` variable `dim` the value passed in the parameter `n`.

3. Allocate memory for an array of `doubles` of length `dim`.

The constructor on line 8 is called a *copy constructor*. It is used whenever it is necessary to create a vector and copy another vector to it at the same time. The copy constructor plays a special role in the language. For instance, it is called whenever a `Vec` is passed by value to a function; as discussed earlier, a local copy of the `Vec` is then created, and the copy constructor tells the compiler how to do that. It is also needed for use with the arithmetic operators, as discussed below.

Incidentally, in the declaration of the copy constructor, the syntax for call by reference is used (this is the meaning of the &). We often use call by reference for reasons of efficiency, to avoid many calls to the copy constructor. Call by reference is usually not required for built-in types, since the cost of copying them is usually negligible.

In addition to telling the compiler how to construct objects, it is also necessary to specify how objects are destroyed. This is the function of the destructor, which takes no arguments and whose name is the name of the class preceded by a ~ (see line 9). The destructor is automatically called whenever the object goes out of *scope* (for example, if the object is a local variable in a function, its scope is the function, and its destructor is called when the function returns). The destructor is also called when the programmer calls `delete` to explicitly deallocate a variable.

In this simple example, the destructor deallocates the memory pointed to by `c`. In more complicated classes, such as are described later, the destructor can perform a variety of functions.

2.2.3 Overloading operators

I will now move to the operators declared on lines 12-14. Most languages support overloading of operators for built-in types; for example, in many languages, $1/2 + 1.0/2.0 = 0.5$, because the division operator is applied differently to integers than to floating point numbers. C++ extends the ability to overload operator to user-defined types.

In my example, I have declared only the most basic operators, namely those for addition and scalar multiplication. In addition, it is necessary to define the assignment operator = (otherwise, how would the compiler know that `dim` is the number of doubles in the array pointed to by `c` which must be copied?).

Before I can explain the syntax of operators, I must give the syntax for calling member functions in general. Note that I have declared a member function called `print` on line 10. If `v` were defined (and initialized) in my program to be a `Vec`, I would have it print itself by issuing the command

```
v.print();
```

Inside of `print`, the private data of `v`, `dim` and `c`, can be referred to by those names.

Now suppose there were a member function declared

```
void example( vector & v );
```

If a program issued the command

```
u.example( v );
```

then, inside the function `example`, two vectors could be manipulated, `u` and `v`. The private data of `u` would be referred to as `dim` and `c`, and the private data of `v` as `v.dim` and `v.c`. The vector `u` is distinguished by the fact that the function was called by `u`.

Member operators are nothing but member functions with a special calling convention. For example, the binary operator + is defined on line 14 of Table 2.2.1. A binary operator takes only one argument, namely the second operand. The first operand is the object that calls the operator. Moreover, the operator can be invoked using either mathematical notation or functional notation. Thus,

```
u + v
```

is equivalent to

```
u.operator+(v).
```

This operator computes the value of `u + v` in the obvious manner and returns the result in a `Vec`. This return requires a call to the copy constructor.

Using member operators, I could define all of the vector space operations, such as inner product, norm, unary minus, and so forth. These operations would give the ability to manipulate vectors without reference to the particular representation. Moreover, since all of the operations, including inner product, can be encapsulated with the data, I would have no problem changing the inner product, if necessary.

In addition to operators, I can define any other public member functions necessary to handle vectors, such as some type of input and output (see lines 10 and 11). I only have to settle on which functions and operations will be defined and with what calling sequences. These calling sequences would then form the interface between the private data and the rest of the program. Let me re-iterate that access to the private data is limited to those functions and operators to which it is specifically granted. This is important.

2.2.4 Refinements for efficiency

One of the main goals of Object-Oriented Design is to allow a program to manipulate the objects which have logical meaning (such as vectors), rather than the representations of those objects (such as arrays). Above and beyond this, for aesthetic reasons, C++ uses operator overloading to allow mathematical expressions to be represented in the code in much the same way that they are written on paper. This naturally has the effect of making the computer code much more readable.

Unfortunately, it is necessary to abandon the aesthetic appeal of the operator notation for reasons of efficiency. Consider the following statement, which could appear in a C++ program which uses the above vector class:

```
w = u*a+v*b;
```

For this example, we will assume that `a` and `b` are `doubles`, that `u`,`v`, and `w` have been declared to be `Vecs` with the same dimension, and that the components `u` and `v` have been initialized.

(Incidentally, the above code exhibits an idiosyncrasy of the C++ operator syntax. Since the operator is called by the first operand, in the case of scalar multiplication, the `Vec` must come first.)

The execution of this statement happens in the following manner. The `operator*` function is called to compute `u*a`, with the result being copied to a temporary `Vec`. The same is done to compute `v*b`. Next, the `operator+` is called to compute the sum of the two temporary `Vec`s, with the result copied to still another temporary `Vec`. Finally, this temporary `Vec` is assigned to `w`. At some later time, possibly as late as the return of the function in which this statement occurs, the three temporary `Vec`s are destroyed by calls to the `Vec` destructor. The point at which the destructor is called is not completely specified by the language, and is therefore up to the designer of the particular compiler being used.

The problems caused by this use of the language are two-fold. First of all, there is the overhead associated with the creation of the temporary objects and the copying of various intermediate results to them. Secondly, since the destruction of the temporary objects may not happen promptly, there is the possible proliferation of temporaries, and the associated waste of memory.

These problems are serious enough for a simple vector class such as that defined above. In one of the cases of interest, namely seismic velocity inversion, the vectors in question are so large that they must be stored on disk during the execution of the program. This situation is an ideal one for object-oriented design, because of the conceptual elegance of hiding the representation, but it is also one in which the above inefficiencies cannot be tolerated.

I want to emphasize that this particular efficiency question arises only because C++ allows operator overloading, which is really an aesthetic feature of the language. All of the same functionality can be achieved by using traditional function call syntax. In this way, the above inefficiencies can be avoided, while the advantages of encapsulation are retained. This will be illustrated below.

2.2.5 Derived classes and virtual functions

The programming paradigm which I have described to this point is described by Stroustrup as "data abstraction". According to him, true object-oriented programming involves the use of *derived classes* and *virtual functions* (see [7], Section 1.2).

The need for something more sophisticated than just data abstraction can be seen through several examples. For instance, consider a mathematical problem which involves vectors from two or more different vector spaces; for instance, suppose both $\mathcal{R}^n$ and L^2 appear. A single definition of a vector class would not allow for the use of both in the same program.

As a more involved example, let me look ahead to the problem of implementing linear operators. There are several different types of operators which may be useful: matrix operators, simple scaling operators ($Lx = \alpha x$), operators defined by a computer subroutine (FFTs, PDE solvers, etc.), and so forth. Any routine that operates on two or more types of operators will require a different class definition for each, but these definitions should not be completely independent (since the operators are all instances of a single abstract "thing").

These situations can be addressed through the use of *inheritance*. One defines a *base class*, say `vector`, which specifies the properties common to all vectors. One then defines *derived classes*, which inherit the properties of the base class and which, in addition, define more specific features.

For example, below I will propose a base class `vector`, with two derived classes, `Rn` (for vectors in $\mathcal{R}^n$) and `SGFin` (for Sampled Grid Functions – in core). An optimization routine may then be written which operates on `vector`s, that is, on members of the base class. This routine could then be linked to other routines which pass to it objects from any derived class, such as `Rn`s or `SGFin`s. These objects are (special cases of) `vector`s, so there is no syntactical problem.

In order for this scheme to succeed, the member functions of the base and derived classes must be *virtual*. The essence of a virtual function is *dynamic binding*: the decision whether to apply a member function of the base class or of a derived class is made at run-time. In my example, either an `Rn` or an `SGFin` can be passed to a routine expecting a `vector`, and the correct member functions will still be called.

As I will discuss below, the ability to use derived classes and virtual functions will be equally useful for defining linear operators.

2.3 The proposed vector classes

I will now present my proposal for defining vector classes for use in optimization and inversion software. The foregoing discussion will hopefully have prepared the reader to understand the meaning of and rationale behind the definitions I make below.

2.3.1 The base class `vector`

The declaration of the class `vector` is found in Table 2.3.1. Before I discuss this declaration, I need to cover a few more points of syntax.

The keyword `const`, which also appeared in the earlier example, is used to aid the C++ philosophy of trapping as many logical errors as possible at compilation. In particular, when an argument is passed by reference to a function, the author of the function can declare that argument to be constant, so that attempts to change its value will be flagged by the compiler. This allows the efficiency of call by reference (no copying) together with the safety of call by value.

For the same reason, member functions can be declared to be `const`, indicating that they do not change the object which calls them. For example, the member function `dot` is defined on line 21; it operates on the calling `vector`, as well as a second `vector` which is passed as an argument. Inside of the parentheses, this second vector is declared to be `const`, while after the parentheses, the calling `vector` is also declared to be `const`. This is as it should be; computing the inner product of two vectors should not involve changing either vector.

The second point of syntax that arises with this declaration is the use of "= 0" after a member function declaration. This indicates that no definition of that function will be provided; it is a *pure* virtual function. Its only purpose is to define the calling sequence for the same function in derived classes. Any class declaration which includes a pure virtual function becomes a pure base class, which implies that no objects of that class can ever be defined.

Pure base classes serve several purposes. First of all, they allow the development of code which will in actuality operate on some (possibly as yet unknown) derived classes, and they declare the basic functions for those derived classes. Moreover, they allow for certain generic functions to be defined which will be the same for all derived classes. I will discuss this later in connection with

```
class vector
{
   public:
      vector(){}
      virtual ~vector(){};
      virtual vector * GetVectorLike() const = 0;
      virtual vector * GetVectorLike(char *) const = 0;
      virtual vector * GetVectorCopy() const = 0;
      virtual int copy( const vector & ) = 0;
      virtual int add( const vector &,const vector & ) = 0;
      virtual int add( const vector & ) = 0;
      virtual int sub( const vector &,const vector & ) = 0;
      virtual int sub( const vector & ) = 0;
      virtual int axpy( real,const vector &,const vector & ) = 0;
      virtual int axpy( real,const vector & ) = 0;
      virtual int xpay( real,const vector & ) = 0;
      virtual int smul( real,const vector & ) = 0;
      virtual int smul( real ) = 0;
      virtual int dot( const vector &,real & ) const = 0;
      virtual int norm( real & ) const = 0;
      int scnorm( const linop &,real & ) const;
      int scdot( const linop &,const vector &,real & ) const;
      virtual int print( ostream & ) const = 0;
      SpaceDef * SpD;
      void * GetVectorClass() const;
      int CompVectorSpace( const vector & ) const;
};
```

Table 2.2: The base class `vector`

`x.copy( y );`	$x \leftarrow y$
`x.add( y,z );`	$x \leftarrow y + z$
`x.add( y );`	$x \leftarrow x + y$
`x.sub( y,z );`	$x \leftarrow y - z$
`x.sub( y );`	$x \leftarrow x - y$
`x.axpy( a,y,z );`	$x \leftarrow a*y + z$
`x.axpy( a,y );`	$x \leftarrow a*x + y$
`x.xpay( a,y );`	$x \leftarrow x + a*y$
`x.smul( a,y );`	$x \leftarrow a*y$
`x.smul( a );`	$x \leftarrow a*x$

Table 2.3: Meaning of the basic arithmetic member functions

the member functions defined on lines 23 and 24.

One more point must be mentioned. C++ has a concept called "templated classes", which should allow the declaration of vectors and operators without the specification of the underlying scalar type (say, `float` or `double`). For the initial implementation, I am not taking advantage of this feature of the language; instead, I am using the identifier `real` to represent either `float` or `double`. The particular choice must be made in a `#define` preprocessor command.

For the reasons mentioned above, there is no use of operator overloading or of the copy constructor. The ordinary arithmetic operations, including assignment (see line 9) are declared on lines 9-18. These functions are designed so that the result of the operation is assigned to the vector which calls the function. Notice that most of these functions have been overloaded; this is to allow the `vector` which calls the function to also be an operand or not, as desired. This is made clear in Table 2.3.1, which gives the action of each function.

By not using operators, I have lost the ability to compute arbitrary linear combinations such as

```
x*a + y*b + z*c
```

in a single line of code. As explained above, doing this would be terribly inefficient. Instead, I have explicitly defined all of the combinations that I expect to need. If others are needed, then I will either have to compute them in two or more steps, or define a new member function. The latter option may be desirable for reasons of efficiency.

I envision using the class `vector` and its derived classes in the following way. An optimization algorithm will be implemented in a function. This function will operate on `vector`s. However, since `vector` is a pure base class, a `vector` can never actually be created. The objects that the function manipulates must therefore be objects from classes derived from `vector`. But where are these objects created?

The answer is two-fold. First of all, the calling routine must provide the function with at least one object derived from `vector`; this is natural, since typically at least a starting point must be provided. The optimization function then creates other objects of the correct type using the

`xptr = y.GetVectorLike();`	an uninitialized vector of the same type as **y** is created and **xptr** becomes a reference to that **vector**
`xptr = y.GetVectorLike( ''filename'' );`	a vector of the same type as **y** is created, it is initialized with data read from the file filename, and **xptr** becomes a reference to it
`xptr = y.GetVectorLike();`	a copy of **y** is created and **xptr** becomes a reference to it

Table 2.4: Meaning of the `GetVectorLike` and `GetVectorCopy`

`x.dot( y,a );`	`a` $\leftarrow$ `<x,y>`
`x.norm( a );`	`a` $\leftarrow \sqrt{\texttt{<x,x>}}$
`x.scdot( A,y,a );`	`a` $\leftarrow$ `<x,Ay>`
`x.scnorm( A,a );`	`a` $\leftarrow \sqrt{\texttt{<x,Ax>}}$

Table 2.5: Meaning of the inner product and norm functions

`GetVectorLike` and `GetVectorCopy` functions declared on lines 6-8. These functions are explained in the Table 2.3.1.

The member functions `dot` and `norm` behave as expected, although the question may arise as to why these functions return `int`s instead of the results of the computations, which would be `real`s. I have made a tentative decision that all member functions will return an `int` error code, with nonzero indicating an error. Therefore, functions which compute a `real` value return it through an argument. The only exceptions are the functions which create or destroy objects. If an error is encountered in one of these functions, the program exits.

The functions `scnorm` and `scdot` are interesting because they are not virtual. The purpose of these functions is to implement a scaled inner product. They take as an argument a linear operator which defines the scaled inner product in terms of the the function `dot`, and they need not be virtual since they act the same for any derived vector: the operator is applied, and then `dot` is called. The actions of `dot`, `norm`, `scdot`, and `scnorm` are given in Table 2.3.1.

The virtual function `print` is provided mainly for debugging purposes. For each derived class, it should be defined to print out some useful information about the vector, either the data or where the data is stored, or some part of the data. The output is sent to the *stream* passed in the argument (C++ implements I/O using streams of bytes; these streams can go to or come from either files or the screen).

Neither the constructor nor the destructor do anything, since only objects derived from this class can be created and destroyed.

Finally, for purposes of error checking, it is necessary to have the ability to decide whether two `vector`s come from the same derived class, and if they have the same defining information (for example, are they both `Rn`s, and is the dimension the same?). For this reason, I have defined an object called a `SpaceDef`, which makes it possible to check both of these conditions. This is the purpose of the functions `GetVectorClass` and `CompVectorSpace` (see lines 24-26).

For purposes of brevity, I will not discuss how `SpaceDef` and the related functions are implemented. However, the ability to specify precisely the type of vector is important when defining linear operators (which require knowledge of the domain and range).

2.3.2 The derived class `Rn`

The declaration of the derived class `Rn` is given in Table 2.3.2. This class has a private section containing a pointer which will point to an array. The dimension of the vector is stored in the `SpaceDef`, which is a public data member of the base class `vector`. The constructors allow for three ways of creating an `Rn`:

- create an "empty" vector with the dimension unassigned and no storage allocated (line 7);
- create a vector in $\mathcal{R}^d$, with storage allocated but not initialized (line 8);
- create the vector defined in the file `fname` (line 9).

The destructor, of course, simply deallocates the array which held the components of the vectors.

All of the operations are implemented in the obvious fashion. Note that the functions `scdot` and `scnorm`, which were declared in class `vector` need not be re-declared nor re-defined. They were not virtual, and they can be applied to any object derived from class `vector`. In a minor way, this illustrates the efficiency of the inheritance concept: only those functions which change need to be implemented for the derived class.

The only new public function is an operator: `operator()`. `operator(i)` returns a reference to the `i`th element of the vector. The fact that this operator returns a reference allows users to change the values of the components.

2.3.3 The derived class `SGFin`

The need to manipulate sampled grid functions in a natural way is one of the main motivations for this work. In seismic inversion, the physical parameters of a section of the earth, such as the seismic wave velocity, are represented as functions of one or more spatial dimensions. In realistic problems, the amount of data needed to represent such a function can be so great that it may not be possible to keep all of the necessary functions in memory during an inversion or other computation.

It is therefore necessary to manipulate objects that are stored on disk, which gives a prime opportunity to take advantage of the data hiding capabilities of C++. In Table 2.3.3 is the declaration of the class `SGFin`, which is derived from the pure base class `vector`. As discussed below, this is a prototype for a class `SGF`, which will be implemented in the future. Before explaining the new features of this class, I will describe the file format that I have tentatively adopted.

To define a sampled grid function on a regular rectangular grid, one must know the dimension of the grid, and the number of samples, origin, and sample rate in each dimension. In the current implementation, the grid can be at most 4-dimensional (this restriction is not at all necessary, but

```
class Rn: public vector
{
   private:
      real * c;
   public:
      Rn(){}
      Rn( int d );
      Rn( char * fname );
      ~Rn();
      virtual vector & GetVectorLike() const;
      virtual vector & GetVectorLike(char *) const;
      virtual vector & GetVectorCopy() const;
      virtual int copy( const vector & );
      virtual int add( const vector &,const vector & );
      virtual int add( const vector & );
      virtual int sub( const vector &,const vector & );
      virtual int sub( const vector & );
      virtual int axpy( real,const vector &,const vector & );
      virtual int axpy( real,const vector & );
      virtual int xpay( real,const vector & );
      virtual int smul( real,const vector & );
      virtual int smul( real );
      virtual int dot( const vector &,real & ) const;
      virtual int norm( real & ) const;
      virtual int print( ostream & ) const;
      real & operator()( int i );
};
```

Table 2.6: The derived class `Rn`

it does make the calling sequence of the routines `getSGFin` and `putSGFin` simpler). All of the data related to an `SGFin` is stored in binary, with this information defining the grid first in the file. The order is:

- dimension `dim` of grid (`int`)
- number of samples in dimension 1,...,`dim` (`ints`)
- origin in dimension 1,...,`dim` (`reals`)
- sample rate in dimension 1,...,`dim` (`reals`)

Following the grid information are the samples of the function values, with the samples stored according to the innermost dimension. In other words, if the function being represented is $f(x, y, z)$ and the grid defines the values of x,y, and z to be $x_0, x_1, \ldots, x_{l_0-1}$,$y_0, y_1, \ldots, y_{l_1-1}$, and $z_0, z_1, \ldots, z_{l_2-1}$, then the function values are stored in the order

$$\begin{array}{c} f(x_0, y_0, z_0) \\ f(x_1, y_0, z_0) \\ \vdots \\ f(x_{l_0-1}, y_0, z_0) \\ f(x_0, y_1, z_0) \\ f(x_1, y_1, z_0) \\ \vdots \\ f(x_{l_0-1}, y_1, z_0) \\ \vdots \\ f(x_0, y_{l_1-1}, z_0) \\ f(x_1, y_{l_1-1}, z_0 \\ \vdots \\ f(x_{l_0-1}, y_{l_1-1}, z_0) \\ f(x_0, y_0, z_1) \\ f(x_1, y_0, z_1) \\ \vdots \\ f(x_{l_0-1}, y_0, z_1) \\ \vdots \end{array}$$

These are `reals`.

The public section of the `SGFin` declaration differs from what we have seen in the constructors, and in the methods provided for accessing the data. There are two constructors, and since some of the arguments are assigned default values, this gives several different ways to create an `SGFin`. In every case, when an `SGFin` is created, the grid geometry is stored as private data (`dim`,`lens`,`origs`,`steps` are the dimension of the grid, numbers of samples, origins, and sample rates, respectively), and the data is stored in an array. The destructor stores the `SGFin` in a permanent file, if desired. The constructors are summarized in the Table 2.3.3.

```
class SGFin:  public vector
{
   private:
      char *fname;                 // Name of file holding data
      int save;                    // 1 = save file upon destruction; 0 = don't
      real * dat;                  // Pointer to array holding data
      int datlen;                  // Length of data array
   public:
      int getSGFin( real * ptr,int size,int i1=-1,int i2=-1,int i3=-1 );
                                   // Returns pointer to all or part of
                                   // the SGFin of
                                   // dimension size, either all of
                                   // f (if size = dim) or, for example,
                                   // f..i1..idim if size = 2,etc.
      int putSGFin( real * ptr,int size,int i1=-1,int i2=-1,int i3=-1 );
      SGFin( char * srcname,int s=0, char * filename = NULL );
      SGFin( const SGFin &,int s=0,char * filename = NULL );
      virtual ~SGFin();
      virtual vector & GetVectorLike() const;
      virtual vector & GetVectorLike( char * srcfile ) const;
      virtual vector & GetVectorCopy() const;
      virtual int copy( const vector & );
      virtual int add( const vector &v1,const vector &v2 );
      virtual int add( const vector &v2 );
      virtual int sub( const vector &v1,const vector &v2 );
      virtual int sub( const vector &v2 );
      virtual int axpy( real a, const vector &v1,const vector &v2 );
      virtual int axpy( real a,const vector &v2 );
      virtual int xpay( real a,const vector &v2 );
      virtual int smul( real a,const vector &v );
      virtual int smul( real a );
      virtual int dot( const vector &,real & ) const;
      virtual int norm( real &) const;
      virtual int print( ostream & ) const;
};
```

Table 2.7: The derived class SGFin

`SGFin( filename )`	create an `SGFin` which is a copy of the one in the file `filename`; do not save it upon destruction.
`SGFin( filename,1 )`	create an `SGFin` which is a copy of the one in the file `filename`; overwrite `filename` with the new value upon destruction.
`SGFin( filename,1,srcname )`	create an `SGFin` which is a copy of the one in the file `srcname` and save its value to the file `filename` upon destruction.
`SGFin( f )`	create an `SGFin` like `f` (an `SGFin` which already exists; initialize the data to zero and do not save upon destruction.
`SGFin( f,1,filename )`	create an `SGFin` like `f`; initialize the data to zero and save its value to the file `filename` upon destruction.

Table 2.8: `SGFin` constructors

call	result
`getSGFin(d,3);`	f
`getSGFin(d,2,j);`	$f(\cdot,\cdot,z_j)$
`getSGFin(d,1,i,j);`	$f(\cdot,y_i,z_j)$

Table 2.9: Sample calling sequences for getSGFin and their meanings

The public functions `getSGFin` and `putSGFin` are used to read and write all or part of the data. In the current implementation, these functions will access the data only in such a way that certain contiguous storage is involved. For example, suppose that the `SGFin F` represents a function $f(x,y,z)$. I will denote by $f(\cdot,\bar{y},\bar{z})$ the function obtained by restricting f to the set $\{(x,\bar{y},\bar{z})|x \in \mathcal{R}\}$, and similarly for $f(\cdot,\cdot,\bar{z})$.

With this notation, Table 2.3.3 describes the calling sequences for `getSGFin`. Note that the calling routine must provide the array, pointed to by the first argument, to hold the data. The second argument is the dimension of the set to which the function is to be restricted, and the following arguments, if needed, define that set.

`putSGFin` works the same way. It is not possible to use these routines to directly manipulate, say, $f(x_i,\cdot,z_j)$. These routines are not used to implement the other member functions; they are provided solely for the use of simulators which may be written to operate on `SGFin`s. If the current strategy proves to be too restrictive, it will be replaced by a more flexible one.

The dimensional information (number of samples, sample rate, etc.) are contained in the `SpaceDef` declared in the base class `vector`. Public member functions will eventually be provided to allow simple access to this information.

The private section of `SGFin` contains information about whether and where the `SGFin` is to be stored upon destruction, as well as the amount of data (`datlen`) and a pointer to the data itself (`dat`).

2.3.4 Other derived classes

There are several other classes of vectors which would be useful to have in a class library. One of the first to be implemented is a class representing a vector from a Cartesian product of vector spaces. This will be easy to do; all that is needed is the number of factors in the product and an array of `vector` pointers. The public functions are then written to loop through the factors, applying the same function (which, by means of dynamic binding, will be from the class of which the factor is an object) to each one.

The class `SGFin` is intended to be mainly a test class; for realistic seismic work, the grid functions must be stored on disk even during the execution of the program. I will soon be defining the class `SGF`, which will store the data in a temporary file rather than an array during the execution of the program.

There are many other possibilities for vector class, such as functions defined by spline coefficients instead of samples. More derived classes will be implemented in the future, after the basic approach has been validated.

3 Linear Operators

There are several types of linear operators which will be useful for optimization. When the underlying vector space is `Rn`, then all of the linear operators can be represented as matrix operators, so I will define various derived classes of matrix operators. For optimization, the classes needed are diagonal matrices, symmetric matrices, and general (nonsymmetric) matrices; the last two classes must be further divided into sparse and dense storage classes.

For approximating the Hessian matrix in a minimization algorithm, the limited memory BFGS method of Nocedal ([5]) involves a certain linear operator defined by outer products. This operator can be implemented for any vector class.

In seismic inversion, many different types of linear operators can be involved: PDE solvers, convolutions, FFTs, and so forth. These types of operators are best defined by computer subroutines, and general operators such as these frequently take parameters.

I propose to implement the above types of operators as derived classes of a common base class. I believe that the elegance obtained by encapsulating the vector information in the vector classes and the linear operator definitions in the linear operator classes will simplify the designing of complex codes considerably.

3.1 The base class `linop`

The declaration of `linop` can be found in Table 3.1. Note that it is necessary to define both the domain and the range of the operator; the `SpaceDef` pointers on lines 5-6 allow for this. Beyond

```
class linop
{
   public:
      linop(){}
      virtual ~linop(){}
      SpaceDef * DomSpD;
      SpaceDef * RangeSpD;
      virtual int action( const vector &,vector & ) const = 0;
      virtual int action( vector & ) const = 0;
      virtual int print( ostream & ) const = 0;
};
```

Table 3.1: The base class `linop`

this, the only public functions are `action`, which applies the operator to a vector in the domain to obtain a vector in the range, and `print`, which is provided for debugging purposes.

Note that `action` is overloaded, allowing for the possibility of replacing the input by the output when this is desirable for efficiency of storage. Of course, this operation will only be allowed if the domain and range are the same.

As the declaration shows, `linop` is a pure base class. At this point, three derived classes have been implemented, `muI`, `lmbfgs`, and `genop`.

3.2 The derived class `muI`

`muI` is the simplest of the linear operators. It implements the operator defined by scalar multiplication. It also allows for an efficient implementation of the identity operator, which is sometimes needed.

The declaration of `muI` is found in Table 3.2. The flag `id` (line 4) indicates whether the operator is the identity. If it is, then `action` either copies (line 8) or does nothing (line 9). If `id` is zero, then `action` calls `smul` to perform the scalar multiplication. Note the power of virtual functions here; an operator of class `muI` does not know the derived class to which the input vector belongs, but through dynamic binding, the member function from the correct class will be called.

3.3 The derived class `lmbfgs`

I will not explain the declaration of `lmbfgs`, which is found in Table 3.3, in detail. I only wish to note that such an operator is defined by certain outer products of vectors; the use of derived classes and virtual functions allow this class of operator to be implemented without reference to the particular type of vector involved.

I will also note that this class has a new public member function, `update`. In the framework of a secant method for minimization, the Hessian approximation is updated after each new iterate is determined.

```
class muI: public linop
{
   public:
      int id; // 1 if identity
      real mu;
      muI( int flag,real mu,SpaceDef * spd );
      virtual ~muI();
      virtual int action( const vector &,vector & ) const;
      virtual int action( vector & ) const;
      virtual int print( ostream & ) const;
};
```

Table 3.2: The derived class `muI`

```
class lmbfgs:  public linop
{
   private:
      int maxu;     // Maximum number of BFGS updates stored
      int curu;     // Current number of BFGS updates stored
      muI * b0;     // Initial Hessian approximation
      vector ** slist;                  // Points to last curu steps
      vector ** ylist;                  // Points to last curu gradient differences
      real * rho;   // Last curu multipliers
      int * indx;   // Pointers into slist,ylist,rho
   public:
      lmbfgs( int,SpaceDef *,muI * );    // Constructor
      virtual ~lmbfgs();                // Destructor
      virtual int action( const vector &,vector & ) const;
                    // Compute the action of the operator on a vector
      virtual int action( vector & ) const;
                    // Compute the action of the operator on a vector
      virtual int update( const vector & x,const vector & xplus,
            const vector & g,const vector & gplus );
      virtual int print(ostream &) const;
};
```

Table 3.3: The derived class `lmbfgs`

```
1.  typedef int (*genopCall)( const genop *, const vector &,vector & );
2.
3.  class genop:  public linop
4.  {
5.     public:
6.        genop()
7.        genop( int n,SpaceDef * DomSpD,SpaceDef * RangeSpD,genopCall call );
8.        genop( int n,SpaceDef * DomSpD,genopCall call );
9.        ~genop();
10.       genopCall opCall;
11.       int nparam;
12.       void ** param;
13.       virtual int action( const vector &,vector & ) const;
14.       virtual int action( vector & ) const;
15.       virtual int print( ostream & ) const;
16. };
```

Table 3.4: The derived class `genop`

3.4 The derived class `genop`

The class `genop` is intended to implement a general linear operator which is defined by a computer subroutine. The declaration of this class is given in Table 3.4.

I have tried to define `genop` so as to give maximal flexibility for linking linear operators defined by simulators to the rest of the C++ code. There are two points which require special attention: simulators will frequently be written in other languages, notably FORTRAN, and they may require various parameters in addition to the `vector` which serves as the input.

In order to construct an object of class `genop`, it is necessary to define a C++ function with the calling sequence given in the `typedef` on line 1. This C++ function can be a "wrapper" for a subroutine written in any convenient language, such as FORTRAN. In this way, existing simulators can be introduced into the code as `genop`s; there is no need to rewrite them in C++.

The declaration of `genop` includes `nparam`, an `int` describing the number of parameters required for the definition of the operator, and `param`, a pointer to an array of `void` * pointers. Any object, including a built-in type, can be pointed to by a `void` * pointer, so this scheme gives a maximal amount of flexibility to pass parameters to the operator.

At the same time, however, at least in seismic inversion work, the parameters needed to define an operator fall into two classes and should probably not all be treated in the same way.

In many applications, the forward map depends on two (or more) vectors, with at least one of them having a linear influence on the data. For example, the forward map may take the form

$$A : X \to \mathcal{L}(Y, D),$$

where X, Y, and D are Hilbert spaces and the model parameters are represented by $x \in X$ and $y \in Y$. This notation indicates that A depends linearly on y and (possibly) nonlinearly on x. For a specific x, then, $A(x)$ is a linear operator. The intended use of the member `param` is to allow x to passed to A, thus defining a linear operator.

At first glance, this may seem to be a rather circuitous method for achieving the desired end. Why not just pass both x and y in the calling sequence of the function, as would typically be done in procedural programming?

The answer lies in the mathematics: $A(x)$ is a distinct object, a linear operator, which may be operated upon. For instance, $A(x)$ could represent a normal operator, and one may wish to pass it to a conjugate gradient routine to solve the normal equations. By defining a parameterized operator, we can treat any particular instance of that operator (that is, $A(x)$ for some particular x) as an object in its own right. I believe that the gain in conceptual elegance is significant.

The question remains of how to treat other types of parameters required for a simulation, such as grid spacing and the like. Since these are quantities that do not change throughout the simulation, one solution is to define a structure which will contain them and which the simulator can access. Another possible solution is for the simulator to read them from a file. Experience should show the best way of handling this problem.

4 Discussion of related work

The approach that I have taken in this proposal is rather different from other efforts in this area. I will discuss some of the differences, and the reasons for the choices that I have made.

In designing these classes, I have been guided by a desire to allow many different types of vectors to be represented and manipulated. Consider the following three applications:

- Seismic inversion. The vectors are finite-dimensional representations of functions. However, very different methods of representing functions exist. In some cases, the flexibility of sampled grid functions is desirable. In other cases, the control given by a spline representation is needed.
- Computational biology. The unknowns may be the coordinates of the atoms in a molecule. In this case, the vector could be represented as a vector in $\mathcal{R}^{3m}$, say, but this ignores the fact that the components of the vector naturally occur in triples, with each triple having an independent meaning.
- Parameter identification. The unknowns may be a finite sequence of scalars, in which case representation as a vector in $\mathcal{R}^n$ is natural.

I would like to be able to write an optimization code which can be applied, without modification, to a problem in any of the above fields.

Of course, object-oriented design does not eliminate the need to directly manipulate the actual representation of the vectors; rather, it allows that manipulation to be hidden at a level where it need not affect the writer of the optimization code. In this way, it is hoped, the physics of the particular problem can be segregated from the numerical analysis.

I have tried to devise a set of classes that is very general and will allow most reasonable extensions. Other authors have designed classes with more specific problems in mind.

For example, LAPACK++ includes the definition of a very broad family of vector and matrix classes to facilitate the use of high performance linear algebra algorithms. Their classes are both more general than my proposed matrix classes, in that they have a wide variety of derived matrix classes (tridiagonal, banded, orthogonal, etc.), and more limited: they only deal with vectors from $\mathcal{R}^n$ (or $\mathcal{C}^n$).

Others have proposed similar vector-matrix classes; see, for instance, Davies ([2]) and the software produced by Dyad Software Corporation ([1]) and Rogue Wave Software ([6]).

Nichols et al. ([4]) have designed vector classes which represent sampled grid functions, together with classes of linear operators which implement operations occurring in geophysical research (convolutions, FFTs, shifts, etc.). Their work is very similar in motivation to my SGF class. However, they apparently do not define other vectors sharing a common base class.

Meza ([3]) has approached the question from a different level, defining classes to represent optimization problems and optimization algorithms. His work builds on the classes of Davies ([2]), and provides at least one approach to the question of how to handle algorithmic parameters and simulation parameters. Since an algorithm is an object, it can have space to store the parameters, which could be passed to the constructor, for example. Similarly, the simulator is part of the definition of the optimization problem, which, as an object, can provide storage for the parameters defining the simulation.

5 Acknowledgements

This work was partially supported by the National Science Foundation, the Office of Naval Research, the Texas Geophysical Parallel Computation Project, the Schlumberger Foundation, and The Rice Inversion Project. TRIP Sponsors for 1994 are: Advance Geophysical, Amoco Production Co., Conoco Inc., Cray Research Inc., Exxon Production Research Co., Interactive Network Technologies, Mobil Research and Development Corp., and Texaco Inc.

I would also like to thank Phil Keenan for several helpful discussions concerning class construction in C++.

References

[1] Dyad Software Corporation. M++ class library. Bellevue, Washington, 1991.

[2] R.B. Davies. Newmat07, an experimental matrix package in C++. robertd@kauri.vuw.ac.nz, 1993.

[3] J.C. Meza. Opt++: An object–oriented class library for nonlinear optimization. Technical Report 94-8225, Sandia National Laboratories, Sandia National Laboratories, Livermore, CA, 1994.

[4] D. NICHOLS, G. DUNBAR, and J. CLAERBOUT. The C++ language in physical science. *OON-SKI '93*, pages 339–353, April 1993. Proceedings of the First Annual Object–Oriented Numerics Conference.

[5] J. Nocedal. Updating quasi–newton matrices with limited storage. *Mathematics of Computation*, 35:339–353, 1980.

[6] Rogue Wave Software. Math.h++. Corvallis, Oregon, 1992.

[7] B. STROUSTRUP. *The C++ Programming Language*. Addison–Wesley, Reading, MA, second edition, 1991.