Hybrid-norm and Fortran 2003: Separating the physics from the solver

Robert G. Clapp

ABSTRACT

Object-oriented approaches allow a separation between solvers and operators. An abstract vector class is created with a limited set of methods. Solvers are written in terms of this abstract vector class and operators act on vectors inherited from the abstract class. Ideally, this separation allows the geophysicist to leverage the work of the mathematician without needing to understand the implementation details of the optimization method. The minimal set of object-oriented features of Fortran95 and its predecessors limited the potential separation between the physics and the solver. New inversion approaches, such as the hybrid norm, further hampered this separation when using conventional vector class descriptions. By using the object-oriented features of Fortran 2003, a more complete separation between solvers and operators can be achieved. By expanding the vector class definition, approaches such as the hybrid norm can be implemented.

INTRODUCTION

A geophysicist understands and/or approximates how a given set of earth properties (model) would create a given set of measurements (data). Geophysics is often an attempt to do the inverse: from a set of recorded data, estimate a model. When the set of measurements and/or the number of model points is large, matrix-based approaches become impractical. Iterative approaches are often the method of choice for large-scale estimating problems. Iterative solvers can become quite complex, quite quickly, and are generally more the domain of the mathematician than the geophysicist. Ideally we would like to leverage the mathematician's expertise without having to understand all of the details of the implementation. Nichols et al. (1993); Gockenbach (1994) all viewed model estimation through an object-oriented framework, allowing this separation using C++. Schwab (1998) described a java-based approach to this problem, and Clapp (2005) described a python-based approach for large, out-of-core solvers. SEP chose instead to use Fortran 90. Unfortunately, Fortran 90 is far from a complete object-oriented language, and as a result complicated inversion problems are cumbersome to describe given its limitations. The recent introduction of the hybrid norm (Claerbout, 2009; Zhang and Claerbout, 2010) is but one example of the limitations of Fortran 90 for solving inverse problems. Recently, Fortran compilers have begun to support Fortran's latest incarnation, Fortran 2003, which is more complete object-oriented language.

In this paper, I show how to implement an abstract solver class in Fortran 2003. I begin by describing one method to implement an abstract operator-based iterative solver. I describe how the definition of a vector needs to be extended to support the hybrid norm. I then show how the abstract operator and vector classes can be implemented in Fortran2003.

OPERATOR-BASED OBJECT-ORIENTED SOLVERS

There are at least two different approaches one can take to solving the typical geophysical inversion problem. Harlan (1996), among others, takes a Bayesian approach to inversion. The Bayesian approach allows a natural inclusion of *a priori* statistical properties of the model. SEP (Claerbout, 1999) has traditionally taken an approach which is described as either classical, traditional, or deterministic. The classical approach attempts to find the model $\mathbf{m}$ that minimizes the data misfit. Given a recorded dataset $\mathbf{d}$, and a linear operator $\mathbf{L}$, we attempt to minimize the residual vector $\mathbf{r}$ which is defined as

$$\mathbf{0} \approx \mathbf{r} = \mathbf{d} - \mathbf{L}\mathbf{m}. \tag{1}$$

In the simplest case where we are using steepest descent to solve the linear least squares inversion, we estimate $\mathbf{m}$ by mapping the initial residual (in this simple case $-\mathbf{d}$) back into the same space as the model to form a gradient vector $\mathbf{g}$ by applying the adjoint of $\mathbf{L}$. We then map the gradient vector back into data-space by applying $\mathbf{L}$ to form $\mathbf{rr}$. Finally, we find the scaling factor of $\mathbf{rr}$ that will make $\mathbf{r} + \mathbf{rr}$ as small as possible. We then repeat this procedure until $\mathbf{r}$ is suitably small. More complex inversion approaches are normally built on this basic concept.

Vector class

The solver writer doesn't need to know anything about $\mathbf{L}$ other than how to apply it and it's adjoint. In fact, the solver writer doesn't need to know much about $\mathbf{m}$ or $\mathbf{d}$. The steepest descent approach described above involves only three mathematical operations. In order to find the best scaling factor $\mathbf{rr}$, we need to calculate a dot product. In order to update the model and the residual, we will need to scale $\mathbf{rr}$ and add it to $\mathbf{r}$. We can define the interface for calling the forward of $\mathbf{L}$ as

```
call lop (logical add, vec m, vec d)
```

If the class `vec` has the ability to perform the add, scale, and dot product functions, we can begin to write a generic solver. Two more initialization functions are needed in the class `vec`. We need to be able to create the gradient vector before we can apply the adjoint. As a result, we need to be able to create a vector with the same number of elements as the model. Put another way, we need to *clone* the model. We also need to be able to zero this vector, or *set* the vector to some value.

There are several other functions that can be useful in a generic vector class. The ability to check that two vectors are from the same vector space can avoid many bugs. The ability to fill the vector with random numbers makes it easy to test whether or not an operator passes the dot product. Scaling and adding are often done together by defining a scale-add feature,

$$\mathbf{v} = \mathbf{av} + \mathbf{bw}, \tag{2}$$

that updates a vector $\mathbf{v}$ by scaling it with a scaled version of a second vector $\mathbf{w}$ we can often improve performance. Finally, having the ability to make a copy of only the space a vector sits in rather than making copy of all elements can often improve performance and reduce storage requirements.

Operators

The base operator class contains the ability to perform a mapping from the vector-space of $\mathbf{m}$, its domain, to the vector space of $\mathbf{d}$, the operator's range (the forward), and vice versa. It is beneficial for an operator to store a description of these two spaces (the reason for the clone-space function described above). This performs two functions. First, the operator can perform a sanity check to make sure that the spaces of model and data passed into the forward adjoint function call match the space of initialized domain and range. The second reason is that inversion problems are often more complicated then the generic problem described by equation 1. For example, if $\mathbf{L}$ is actually the cascade of two operator $\mathbf{A}$ and $\mathbf{B}$,

$$\mathbf{L} = \mathbf{AB} \tag{3}$$

we need the ability to check that the domain of is $\mathbf{A}$ is equivalent to the range of $\mathbf{B}$ and we need to create a vector of that size to hold the intermediate result of applying $\mathbf{B}$ in the forward case (and $\mathbf{A}$ in the case of the adjoint).

Combining operators

The number of different ways that operator might need to be combined to solve a given inversion problem is infinite. Fortunately, all possible combinations can be built from four building blocks. The first is the row operation described above. A second applies two different operators to the same vector (a column vector),

$$\begin{bmatrix} \mathbf{d_1} \\ \mathbf{d_2} \end{bmatrix} = \begin{bmatrix} \mathbf{L_1} \\ \mathbf{L_2} \end{bmatrix} \mathbf{m}. \tag{4}$$

Its corollary, a row operator, which forms a single data $\mathbf{d}$ from to models,

$$d = \begin{bmatrix} \mathbf{L_1} & \mathbf{L_2} \end{bmatrix} \begin{bmatrix} \mathbf{m_1} \\ \mathbf{m_2} \end{bmatrix}. \tag{5}$$

Finally, a diagonal operator that applies different operators to different models

$$\begin{bmatrix} \mathbf{d_1} \\ \mathbf{d_2} \end{bmatrix} = \begin{bmatrix} \mathbf{L_1} & \\ & \mathbf{L_2} \end{bmatrix} \begin{bmatrix} \mathbf{m_1} \\ \mathbf{m_2} \end{bmatrix}. \tag{6}$$

The final three all imply the creation of a new vector class that is the combination of two or more vectors. This super vector class is a storage object that calls the appropriate vector class function sequentially (except for the dot product function that must add the result of each call). As described in the next section inversion problems are often combinations of several of these combo-operator/vectors and these functions are often called recursively.

Solvers

An iterative solver operates a problem that can be described as simply as equation 1. Achieving this simple form is often a more complex problem. The problem is broken up into three steps: obtaining an initial residual, finding the vector that best solves the constructed inverse problem, and updating the model according to this vector. Each one of these steps involve several different potential user inputs. For simplicity, I am going to describe all potential inversion problems in terms of a regularized inversion problem with two fitting goals (each goal could be actually multiple fitting goals combined using the functions described above).

The first step involves finding the initial residual vectors, $\mathbf{r_d}$ and $\mathbf{r_m}$. The user might begin by specifying some initial values for these two vectors. These values then need to be updated according to the data $\mathbf{d}$ associated with the problem, a potential initial model $\mathbf{m_0}$, the operators being used $\mathbf{L_1}, \mathbf{L_2}$, and weights applied to the residual $\mathbf{W_0}, \mathbf{W_1}$.

$$\begin{bmatrix} \mathbf{r_d} \\ \mathbf{r_m} \end{bmatrix} = \begin{bmatrix} \mathbf{r_d} \\ \mathbf{r_m} \end{bmatrix} + \begin{bmatrix} \mathbf{W_1} & \mathbf{0} \\ \mathbf{0} & \mathbf{W_2} \end{bmatrix} \left(\begin{bmatrix} \mathbf{d} \\ \mathbf{0} \end{bmatrix} - \begin{bmatrix} \mathbf{L_1} \\ \mathbf{L_2} \end{bmatrix} \mathbf{m_0} \right). \tag{7}$$

Once the initial residual is calculated, we iterate to find $\mathbf{x}$ through,

$$\begin{bmatrix} \mathbf{r_d} \\ \mathbf{r_m} \end{bmatrix} = \begin{bmatrix} \mathbf{W_1 L_1} \\ \mathbf{W_2 L_2} \end{bmatrix} \mathbf{Sx}, \tag{8}$$

where $\mathbf{S}$ is a preconditioning operator. Finally we need update our model,

$$\mathbf{m} = \mathbf{m_0 Sx}. \tag{9}$$

This procedure allows a single solver to be written for a myriad of different inverse problems. It also demonstrates one of the biggest weaknesses of Fortran 90. Fortran 90 does not support function pointers. As a result, SEP has traditionally written different solvers for regularized and preconditioned problems. Combination operators could only be created by writing a function that specifically named the two operators that were to be combined. As a result, creating complex inversion problems quickly became cumbersome and prone to errors.

SUPPORTING THE HYBRID NORM

The L2-norm is often a non-optimal choice both because of the non-Gaussian nature of noise in data and it's tendency to create smooth models with traditional regularization schemes. To improve model estimation, authors have suggested using either direct L1-norm approaches, or more commonly, Iteratively Reweighed Least-Squares (IRLS) to approximate different norms. IRLS attempts to find the best model at a non-L2 norm by a series of linearization s. Each non-linear iteration updates the weighting $\mathbf{W}$ of the residual. For example we can achieve L1 like behavior by updating the weighting operator through,

$$w(i) = \frac{1}{|r(i)|}. \tag{10}$$

IRLS methods tend to be difficult to use because the user must choose carefully the number of relinearizations and the numbers of steps between relinearizations carefully. (Claerbout, 2009) suggested an alternate approach that dynamically changes the weighting function every iteration and uses a Taylor expansion of the standard conjugate direction algorithm to update the solution. Further he suggests a different norm, the hybrid norm, that creates a smooth transition between the standard L2 problem and a L1 problem. Given an error function E and a residual vector $r(i)$ the hybrid norm is defined

$$E = \sum_i R^2(\sqrt{1 + r^2(i)/R^2} - 1), \tag{11}$$

where $\mathbf{R}$ is a user supplied bad-data percentile.

Creating an inversion framework that supports a Taylor expansion approach to conjugate directions requires adding two additional features to our vector class description. First, we need to associate a norm to each vector. Second we need to be able to multiply a vector by another vector, element by element. Adding support for the hybrid norm requires more changes. A vector must now have a bad-data percentage associated with it, it must be able to find its *ith* percentile value, and create a vector with this value.

IMPLEMENTATION IN FORTRAN 2003

The Fortran 2003 standard makes Fortran a nearly complete object-oriented language. Fortunately (because it improves the compilers ability to optimize) and unfortunately, the object-oriented language components are not described in a very compact manner. The basic object-oriented construct ia a `type`. In Fortran 90, types could not contain function pointers and there was no inheritance concept. In Fortran 2003, a type can contain a reference to a function pointer. A type now is broken into two parts separated by `contains` statement. Above the `contains` statement variables are defined, below procedure pointers. For example, we can create a type `vec` which contains a function that can add one vector to another. The type is declared by:

```
type vec
  real, allocatable :: vals(:)
  contains
   procedure, pass :: add=>add_me
 end type
```

The `=>` keyword indicates that to access the function we should use the name on the left but the name of the procedure is found on the right. The `pass` keyword will be described later. Within the module that contains the `vec` definition, we need to define the `add_me` function:

```
 subroutine add_me(vec1,vec2)
   class(vec) :: vec1
   type(vec)  :: vec2
   vec2%vals=vec2%vals+vec1%vals
 end subroutine
```

Note that `vec1` is declared using the `class` keyword rather than the `type` keyword. The `class` keyword indicates that anything of type `vec` or anything that inherits from `vec` can call this function. We can access the `add` function through the standard Fortran `call` keyword:

```
 type(vec) :: vec1,vec2
 call vec1%add(vec2)
```

Note how the function definition has two arguments while the call description has a single argument. The `pass` keyword is the reason for the discrepancy. The `pass` keyword indicates that the type itself should be passed as the first argument.

An abstract type can also be constructed. An abstract type contains one or more function pointers that haven't been assigned. The abstract type can never be declared. Only types that inherit from it can be declared in a functional unit and only if all function pointers have been assigned. An abstract type is declared with the `abstract` keyword. The keyword `deferred` is used for all functions that will be assigned by inherited objects. In addition, you can define the interface of each function using the `abstract interface` construct. Below is an example of using these features:

```
type,abstract :: vector
  contains
  procedure(add_dec), pass, deferred :: add
end type
```

```
abstract interface
subroutine add_dec(v1,v2)
  class(vector) :: v1,v2
end subroutine
end abstract interface
```

We can declare a type that inherits from this `vector` class using the `extends` construct:

```
type, extends(vector) :: vec_real
 real :: vals(:)
 contains
 procedure, pass :: add=>add_real
end type
```

The function `vec_real` must have the same interface as the type it extends from with the exception of the first argument which now must be of type `vec_real`. Note how the abstract interface for the `add` function defines `v2` as a `class` object. To add two vectors we need them to be the same `type`. We can check the `type` of a class object using the `select type` construct:

```
subroutine add_real(v1,v2)
 class(vector) :: v2
 class(vector_real) ::v1
 select type(v2)
   type is(vector_real)
      v1%vals=v1%vals+v2%vals
   end select
end subroutine
```

Within the `type is` code block, `v2` is assumed to be of type `vector_real` and all components of `vector_real` can be accessed.

Fortran 2003 also provides a cleanup feature. The `finalize` keyword is called when an object is no longer needed (for example, when leaving a subroutine where it has been declared). In the example below, the `deleteit` function is used remove any memory associated with an object.

```
type, extends(vector) :: vec_real
 real :: vals(:)
 contains
 procedure, pass :: add=>add_real
 final ::  deleteit
end type
```

Finally, we need the ability to clone an abstract type. The `allocate` function now takes a keyword argument `source`. For example, we can create an object `v2` of the same type as `v1` even though `v1` is of an abstract rather than concrete type.

```
subroutine cloneit(v1,v2)
  class(vector),pointer :: v1,v2
  allocate (v2,source=v1)
end subroutine
```

With these extensions to the Fortran language, it is possible to completely separate operator writing from solver writing.

Current compiler limitations

Unfortunately, most Fortran compilers are still not Fortran 2003 compliant. Specifically, they lack the ability to copy from source (needed for cloning) and have yet to implement the `final` construct. By the end of 2010, both Intel and PGI promise to fully support these features.

CONCLUSIONS

Iterative-based inversion maps cleanly into an object-oriented framework. Vector, operator, and solver abstract classes can be built upon to solve nearly any inversion problem. The hybrid-norm approach requires some modification from the standard vector class description but can easily be accommodated. The Fortran 2003 standard contains all of the object-oriented features needed to write an inversion library.

REFERENCES

Claerbout, J., 1999, Geophysical estimation by example: Environmental soundings image enhancement: Stanford Exploration Project.

———, 2009, Blocky models via the l1/l2 hybrid norm: SEP-Report, **139**, 1–10.

Clapp, R. G., 2005, Inversion and fault tolerant parallelization using Python: SEP-Report, **120**, 41–62.

Gockenbach, M. S., 1994, Object-oriented design for optimization and inversion software: A proposal: TRIP-Report, **1994**, 1–24.

Harlan, W. S., 1996, Promising research topics: http://sepwww.stanford.edu/oldsep/harlan /papers/seg96review.ps.gz.

Nichols, D., H. Urdaneta, H. I. Oh, J. Claerbout, L. Laane, M. Karrenbach, and M. Schwab, 1993, Programming geophysics in C++: SEP-Report, **79**, 313–471.

Schwab, M., 1998, Enhancement of discontinuities in seismic 3-D images using a Java estimation library: **99**.

Zhang, Y. and J. Claerbout, 2010, Least-squares imaging and deconvolution using the hb norm conjugate-direction solver: SEP-Report, **140**, 129–142.