Krylov space solver in Fortran 2009: Beta version

Robert G. Clapp

ABSTRACT

Solving linear systems using Krylov subspace methods is an ideal candidate for object-orient programming. Iterative solver approaches use only a few different operations on vectors and operators. These operations form the basis of abstract vector and operator classes. Sophisticated solvers can then be written on top of these abstract classes separating the geophysics (operators) from the mathematics (solvers). The minimal set of object-oriented features of Fortran95 and its predecessors limited the potential separation. 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 2008, a separation between solvers and operators can be achieved.

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 estimation problems. Iterative solvers can become quite complex 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 implemented 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 problems encountered in implementing 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 2008, a more complete implementation of the needed object oriented constructs. This paper is a follow up to Clapp (2010), which described how an implementation could be done in Fortran 2003, but was hampered by the immaturity of the Fortran compiler.

In this paper, I show the implementation of an abstract solver class in Fortran 2008. I begin by describing one method to implement an abstract operator-based iterative solver. I describe an abstract vector, abstract operator, and solver class. I finish by showing how a small geophysical inverse problems can be solved using these classes and concrete classes inherited from them.

OPERATOR-BASED OBJECT-ORIENTED SOLVERS

SEP (Claerbout, 1999) has traditionally taken an approach which is described as either classical, traditional, or deterministic to iterative inversion. 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 $\mathbf{\Delta r}$. Finally, we find the scaling factor of $\mathbf{\Delta r}$ that will make $\mathbf{r}+\mathbf{\Delta r}$ 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 its 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 $\mathbf{\Delta r}$, we need to calculate a dot product. In order to update the model and the residual, we will need to scale $\mathbf{\Delta r}$ 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 `vector` class 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 `vector` class. 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. It is also useful to seperate the space a vector exists in and the storage mechanism. The vector space contains thing like whether we are dealing irregular or regular dataset, the

number of samples, and the locations of the samples (for example the origin and sampling of the axes). The table below gives a list of the abstract vector class components.

Function	**Purpose**
`add(vector)`	Add another vector to current vector ($x = x + y$).
`scale(real)`	Scale the vector ($x = x * a$).
`scale_add(scale1,vector,scale)`	Add another vector to current vector ($x = ax + by$).
`scale(vector)`	Calculate the dot product with another vector (return $sum_i a(i) * b(i)$.
`set(real)`	Set the value of a vector ($x = v$).
`mult(real))`	Multiply a vector with another vector ($y = y * x$).
`clone(vec)`	Create another vector of the same type with the same values.
`clone_space(vec)`	Create another vector of the same type with no storage mechanism.
`check_same(vec)`	See if two of vector of the same type and exist in the same space.
`alloc()`	Create a vector from a vector space.
`info(character(len=*),integer)`	Provide user specified debugging information.

From this base class I inherit a `real` vector class and then 1-D to 7-D `real` from this class (Figure1). Further non-uniform classes would inherit form the `real` vector class while out-of-core classes would come directly from the `vector` class.

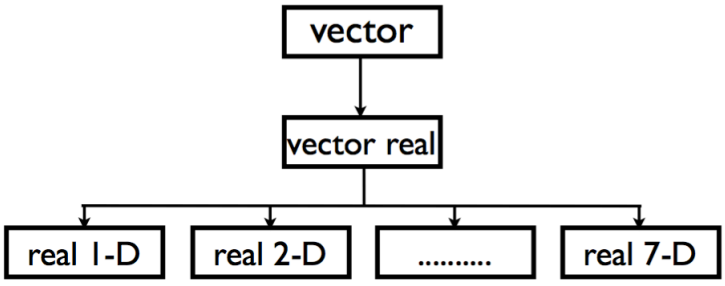

Figure 1: Inheritance class for vectors. The 1-D to 7-D real vectors are inheriting from a real vector class which is inherited from the virtual vector class. [**NR**]

Operators

The base operator class contains the ability to perform a mapping from the vector-space of the operators domain, to the vector space of operator's range (the forward of

the operator), and vice versa (applying the adjoint). 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 or 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 $\mathbf{B}$,

$$\mathbf{L} = \mathbf{AB} \tag{2}$$

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).

A Fortran 2008 `type` must be declared in a module. An example of an operator declaration is seen below. In this case I am creating a causal integration operator.

```
module causal_mod
 use operator_mod  !Description of the generic operator class
 use vec_nd_mod    !The specific vector class used in this module
 implicit none
 type,extends(operator) ::causint_op  !Causint operator declaration
   contains
     procedure,pass :: op=>causint_it !Pointer to the generic forward/adj
     final:: causint_close  !clean function (not necessary in this case)
 end type
```

When leaving a functional unit that has declared a type or when deallocating a type the `final` function is called. In this case we haven't allocated any memory but for completeness, and for debugging, it is often useful to include it.

```
 subroutine causint_close(myop)
   type(causint_op) :: myop
 end subroutine
```

We need a subroutine that sets up a causal integration operator. The only information we need is the size of the vector space in which we are going to be performing causal integration on.

```
 subroutine create_causint_op(myop,v)
   class(vector) :: v   !vector space we are operating on
   class(causint_op) :: myop  !operator we are setting up
   logical :: bm=.false.  !default to having the wrong type of vector
   myop%lab=1
   select type(v)   !check the type of vector
```

```
    class is (real_1d) !make sure it is a 1-D real vector
      bm=.true.  !we have the right type of vector
  end select

  if(.not. bm) call seperr("model and vector must be real_1d")
  call myop%set_domain_range(v,v) !store the domain and range
end subroutine
```

The only thing left is the actual operator. I am going to break it into to parts. The first part is the initialization and the overhead associated with Fortran 2008.

```
subroutine causint_it(myop,adj,add,mod,dat)
  class(causint_op) :: myop  !causal integration object
  logical, intent(in) :: adj,add
  class(vector) :: mod,dat  !vector spaces
  real,dimension(:), pointer :: xx,yy
  real,dimension(:,:),pointer :: ar
  integer :: i,j,nm,nd
  real ::     t
  !Check to make sure the model and data vector
  !spaces match those stored in the operator declaration
  if(.not. mod%check_same(myop%domain))&
    call seperr("domains don't match")
  if(.not. dat%check_same(myop%range)) &
    call seperr("ranges don't match")
  call adj_null(adj,add,mod,dat)

  !Create a pointer to the model values
  select type(mod)
    class is (real_1d)
       xx=>mod%vals
       nm=size(mod%vals)
  end select

  !Create a pointer to the data values
  select type(dat)
    class is (real_1d)
      yy=>dat%vals
      nd=size(dat%vals)
  end select
```

The second part is standard Fortran77/90/2003/2008.

```
   t=0
```

```
   if(adj) then
     do i= nd, 1, -1
         t = t + yy(i)
        xx(i) = xx(i) + t
     end do
   else
      do i= 1, nd
          t   = t + xx(i)
          yy(i) = yy(i) + t
      end do
   end if


  end subroutine
end module
```

Combining operators

The number of different ways that an 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 a chain operator. When the results of applying the first operator $\mathbf{L_1}$ is fed into a second operator $\mathbf{L_2}$,

$$\mathbf{d} = \mathbf{L_2 L_1 m}. \tag{3}$$

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 one a problem that can be described as simply as equation 1. Translating a complicated problem into this simple form is 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 constructing the data space 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 $\mathbf{W_0}, \mathbf{W_1}$.

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

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,

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

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.

EXAMPLE

Converting RMS velocities to interval velocities is one of the most basic problems in reflection seismology. The Dix equation is one of the most common approaches but often leads to unrealistic velocities when dealing with the noise in the RMS velocity measurement. Clapp et al. (1998) points out that there is a linear relationship between $\mathbf{v^2_{rms}}$ and $\mathbf{v^2_{int}}$ using either a modified version of causal integration or using causal integration $\mathbf{C}$ directly and first scaling $\mathbf{v^2_{rms}}$ by sample number. With this

linear relation we can now add a model styling goal, such as smooth $\mathbf{v}^{\mathbf{2}}_{\text{int}}$, given us the fitting goals

$$\begin{aligned} \mathbf{d} &\approx \mathbf{Cm} \\ \mathbf{0} &\approx \epsilon\mathbf{Dm}, \end{aligned} \tag{10}$$

where $\mathbf{D}$ is the derivative operator, $\mathbf{d}$ is the scaled $\mathbf{v}^{\mathbf{2}}_{\text{rms}}$, and the model is $\mathbf{v}^{\mathbf{2}}_{\text{int}}$. We need a weighting term which gives higher importance to good RMS picks, and equal weights all of the RMS velocities (undoing the effect of the sample number scaling), resulting in

$$\begin{aligned} \mathbf{0} &\approx \mathbf{W}(\mathbf{d} - \mathbf{Cm}) \\ \mathbf{0} &\approx \epsilon\mathbf{Dm}. \end{aligned} \tag{11}$$

We can precondition the model by noting that causal integration is the inverse of the derivative except at the first sample. We know that first interval velocity value is the same as the first RMS velocity, resulting in the final setting of fitting equations,

$$\begin{aligned} \mathbf{0} &\approx \mathbf{W}(\mathbf{d} - \mathbf{CMp}) \\ \mathbf{0} &\approx \epsilon\mathbf{p}, \end{aligned} \tag{12}$$

where $\mathbf{p}$ is the preconditioned variable and $\mathbf{M}$ is a masking operator that doesn't allow the first value to change. The advantage of selecting this problem is that its solution is a rather thorough test of all the necessary features of the solver. It involves a starting model, a weighting operator, and a masking operator. It requires both chaining operators and making column operator objects. All of the hard stuff is done away from the user in the solver, the user only needs to construct the required operators and initialize and run the solver.

For this example the conversion is all handled in a module. The module begins by `using` all of the modules that declare the operators and solvers, and the declaration of variables.

```
module vrms_2_int_mod !Transform from RMS to interval velocity
  use causint_mod  !Causal integration
  use weight_mod   !Weighting operator
  use mask_mod     !Masking operator
  use cg_step_mod  !Conjugate gradient
  use obj_solver_mod !Solver module
contains
  subroutine vrms2int( niter, eps, weight, vrms, vint)
    integer,            intent( in)     :: niter     ! iterations
    real,               intent( in)     :: eps       ! scaling parameter
    type(real_1d)                       :: vrms      ! rms velocity
    type(real_1d),target                        :: vint! interval velocity
    real, dimension(:), pointer         :: weight    ! data weighting
```

```
    integer                                 :: st,it,nt
    logical, dimension( size( vint%vals))    :: mask
    logical, dimension(:), pointer      :: msk
    real,    dimension( size( vrms%vals))    :: wt
    real, dimension(:), pointer         :: wght
    type(prec_solver)  :: p_s   !Preconditioned solver

    type(causint_op),target :: ca_op,ca2_op   !Causal integration
    type(mask_op),target    :: m_op           !Masking operator

    type(weight_op),target :: wt_op   !Weighting operator
    type(cgstep),target :: cg   !Conjugate gradient operator
    type(real_1d),target :: dat  !Data
```

Next we need to scale the data, create the weighting vector, and masking vector.

```
vrms2int.f90        vrms2int.unrat      vrms_2_int_mod.mod
    nt = size( vrms%vals)

    call create_vec1(dat,vrms%vals)
         do it= 1, nt
           dat%vals( it) = vrms%vals( it) * vrms%vals( it) * it
           wt( it) = weight( it)*(1./it)           ! decrease weight with time
         end do

    mask = .false.;   mask( 1) = .true.            ! constrain first point
    vint%vals = 0.     ;   vint%vals(1)= dat%vals( 1)

    allocate(msk(size(mask)))
    msk=.not.mask

    allocate(wght(size(wt)))
    wght=wt
```

Finally we need to initialize the operators, setup the solver, and solve for the interval velocity squared.

```
    call create_weight_op(wt_op,vrms,wght) !Create weighting op
    call create_causint_op(ca_op,vint,"a1") !Causal op
    call create_causint_op(ca2_op,vint,"a2")!Preconditioning
    call create_mask_op(m_op,vint,msk)  !Masking operator

    p_s%lop=>ca_op;   !Mapping operator
    p_s%st=>cg;       !Step function
```

```
    p_s%pop=>ca2_op   !Preconditioning operator
    p_s%dat=>dat;     !Data
    p_s%mod=>vint     !model
    p_s%jop=>m_op;    !Masking operator
    p_s%wop=>wt_op    !Weighting operator
    p_s%eps=eps;      !Scaling factor
    p_s%p0=>vint      !Initial preconditioned variable
    call p_s%solve(niter)  !Solve for interval v^2

    call ca_op%op(.false.,.false.,vint,dat)  !Estimated RMS^2

    do it= 1, nt
       vrms%vals( it) = sqrt( dat%vals( it)/it)  !RMS velocity
    end do
    vint%vals = sqrt( vint%vals) !Interval velocity
    deallocate(wght); deallocate(msk)
 end subroutine
 end module
```

Figure 2 shows the result of running the inversion. The left panel shows the original RMS velocity and the mapped RMS velocity. The right panel shows the estimated interval velocity.

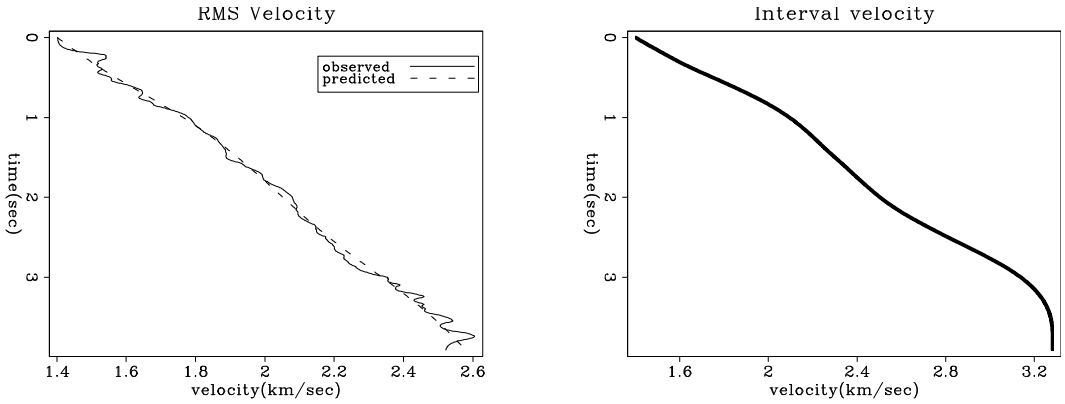

Figure 2: The left panel shows the original RMS velocity and the mapped RMS velocity. The right panel shows the estimated interval velocity. [**ER**]

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 Fortran 2008 standard contains all of the object-oriented features needed to write an inversion library. The resulting inverse code is more verbose than the Fortran 90 approach but the added flexibility makes this an acceptable penalty.

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.

———, 2010, Hybrid-norm and fortran 2003: Separating the physics from the solver: SEP-Report, **142**, 85–92.

Clapp, R. G., P. Sava, and J. F. Claerbout, 1998, Interval velocity estimation with a null-space: SEP-Report, **97**, 147–156.

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

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.