Stanford Exploration Project, Report 117, October 23, 2004, pages 183–??

Short Note

A Python solver for out-of-core, fault tolerant inversion

Robert G. Clapp[1]

INTRODUCTION

In the last ten years SEP has seen a progression in the way it does inversion. Earlier version of Claerbout (1999) coded the conjugate gradient loop within a FORTRAN 77 main program. Later, with the adoption of FORTRAN 90, the solver became a subroutine, operators were in modules, and vectors were 1-D arrays of floats (and later complex). This solution proved effective for a large range of problems. It was an ineffective for other problems and resulted in several additional FORTRAN 90 packages. To handle non-linear problems Clapp and Brown (1999) and Guitton (2000) introduced non-linear solvers. To handle problems that could not fit within core memory, Sava (2001) introduced a out of core solver. All of these packages have been used extensively at SEP over the years, but they all suffer some weaknesses that are related to coding in Fortran 90.

During the same period we have seen people in the geophysical community take a more object-oriented approach to the problem than FORTRAN 90 allows. Deng et al. (1996), Gockenbach and Symes (1999), and Harlan (2004b) used C++ as their controlling inversion framework. Schwab and Schroeder (1997); Schwab (1998) and Harlan (2004b) designed inversion systems within Java. For at least Deng et al. (1996) and Gockenbach and Symes (1999), memory limits was one of the reasons for the object-oriented language choice.

The size and complexity of the inversion problems at SEP has dramatically increased. These problems now involve operators that can take days to weeks to run (Sava and Biondi, 2003; Clapp, 2003b) and can involve multiple instances of complex operations (Clapp, 2003a). Running these problems on Beowulf clusters poses a problem. The odds of a multi-week job running without a node failing is low. In addition, multiple instances of the same, or similar operator, is problematic (increasing with the complexity of the problem) in FORTRAN 90.

In this paper I discuss a new solution to the instability and complex problem description. I describe a Python inversion library which uses abstract vector and operator descriptions. From these abstract classes I derive specific classes to handle out-of-core problems. Operators become wrappers around SEPlib programs and vectors wrappers around SEPlib files. Deng et al. (1996) is a similar solution using C++ and SU. The difference is my Python library is built

[1]**email:** bob@sep.stanford.eda

upon the framework described in Clapp (2004), which handles fault tolerance within parallel programs, and provides the building blocks for easy restarting of inversions.

I will begin by discussing the abstract vector and object design. I will then discuss the specific cases of out-of-core vectors and operators. I will conclude with a simple example of using the library.

ABSTRACT CLASSES

A linear inverse system general involves three basic classes. The vector class is a collector class for a series of numbers. The operator class is some type of mapping between vector objects. The solver class does the actual inversion by calling the operator objects given an initial model and data vector.

Vectors and super vectors

Vectors are composed of two parts: an array of numbers and a description of the space the reside in. The space they reside in is arbitrary but all vectors must be able to perform some basic mathematical operations. They must be able to be scaled by a number, able to add, multiply, and take the dot product of themselves and another vector in the same space. In addition, they must be able to perform some operations related to the space they reside in. They need to be able clone themselves and they need to be able to check whether or not another vector resides in the same space.

For coding simplicity I expanded this minimal set. Each derived vector class must define the following functions.

clone Make a copy of both the space the vector resides in and the vector values.

clone_space Make a copy of just the space the vector resides in.

zero Fill the vector values with zeros.

random Fill the vector values with random numbers.

scale Scale the vector by a number.

add Add the vector to another vector.

scale_add Add the vector to another vector multiplied by a scalar.

scale_addscale Scale the vector and add it to another vector scaled by some scalar.

dot Return the dot product of the vector with another vector.

load Given a vector space, read/or create the vector values.

size A rudimentary method to check vector space similarity.

clean Remove all remnants of the vector.

In addition to the `SEP.vector.vector` described above, an additional arbitrary class is necessary. This class, `SEP.vector.super`, is collection of vectors. Take for example a regularized inversion problem,

$$\begin{aligned} \mathbf{d} &\approx \mathbf{Lm} \\ \mathbf{0} &\approx \epsilon\mathbf{Am}, \end{aligned} \tag{1}$$

where **L** relates the data **d** and the **m**, **A** is a regularization operator, **0** is a vector filled with 0s existing in the range space of **A**, and ϵ is some twiddle parameter. A super vector would be the combination of **d** and **0**.

Operators and combination operators

When it comes to writing an object oriented conjugate gradient based inversion library there are two schools of thought. One is a Bayesian approach that correlates vectors with their decorrelators. In the Bayesian approach, Harlan (2004b,a) being one example, an inverse covariance function is associated with each vector. The approach taken in this library, and the approach taken by Gockenbach and Symes (1999) among others, think more in terms of operators. Operators know the space that their domain and range vectors reside in. For this library I took the latter approach. My decision was based more on code complexity issue more than anything else. When building complex inversion operators, the programmer has to be much more careful if the component operators do not know the space of their domain and range vectors.

The abstract class `SEP.operator.operator` is initialized with a string descriptor, domain vector, and a range vectors. The space these vectors exist in are stored in the operator. The operator has several additional functions.

init_op(restart) Initialize the operator.

job_desc(iter) Return an ASCII string describing the operator given the current iteration. This will be used to keep track of the progress of the inversion.

forward_op(model,data,add,restart) Run the forward operation possibly by adding to the data, and with the ability to signify a restart of the operation.

adjoint_op(model,data,add,restart) Run the adjoint operation possibly by adding to the model, and with the ability to signify a restart of the operation.

check_operator(domain,range) A function to check to make sure the domain and range vector conform to the domain and range vectors with which the operator was initialized.

forward(domain,range,status,iter,add,restart) A wrapper for running the forward operation. The additional status and iter parameters are used to record the starting and finished of forward operation to enable restarting.

adjoint(domain,range,status,iter,add,restart) The same idea as the forward function, this time for wrapping the adjoint operation.

To build more complex operations there are two component operator classes `SEP.vector.chain` and `SEP.vector.array`. Both of the complex operators are initialized by a string descriptor and an array of operators. The `SEP.vector.chain` chains two operators, an example is the preconditioning problem,

$$\begin{aligned} \mathbf{d} &\approx \mathbf{LSp} \\ \mathbf{0} &\approx \epsilon \mathbf{p}, \end{aligned} \tag{2}$$

where $\mathbf{S}$ is the preconditioning operator and $\mathbf{p}$ is a vector in the domain of $\mathbf{S}$. In this case `SEP.vector.chain` would be formed from $\mathbf{L}$ and $\mathbf{S}$. A condition of forming this operator is that the range vector of operator `i` must exist in the same space as the domain vector of `i+1`. When this operator is initialized it will automatically create all of the intermediate spaces. The other class, `SEP.vector.array` creates a new operator which is a matrix of existing operators. For examples we can think of the regularization inverse problem in terms of the matrices,

$$\begin{pmatrix} \mathbf{d} \\ \mathbf{0} \end{pmatrix} = \begin{pmatrix} \mathbf{L} \\ \epsilon \mathbf{A} \end{pmatrix} (\mathbf{m}). \tag{3}$$

The `SEP.vector.array` is initialized by the array of operators and the number of collums and rows in the new array operator. In forming the operator the library makes sure that all of the domain and range vectors make sense (for the regularization problem the domain of $\mathbf{L}$ and $\mathbf{A}$ must be the same. From these two building blocks any inverse problem can be described.

Solver

With the ease in forming complex operators, the solver becomes a rather trivial code. All of the linear solvers are derived from a basic `SEP.solver.solver` class. This class is initialized with a model vector, residual vector, an operator, a step function, the number of iterations to run, and optional verbosity flag, and whether or not the job is being restarted. The initialization process is limited to first cloning model vector and the residual vector to form the gradient and the vector resulting from $\mathbf{Lg}$. It then initializes the step function, calculates and stores (or in the case of restarts, reads) the tasks it needs to perform. Taking a linear step becomes the following trivial code fragment.

```
def step(self,iter):
  self.op.adjoint(self.g,self.rr,self.status,iter)
  self.op.forward(self.g,self.gg,self.status,iter)
  self.status.update_status(str(iter)+".step",["started"])
```

```
if not self.stepper.step(self.x,self.g,self.rr,self.gg): return None
self.status.update_status(str(iter)+".step",["finished"])
return 1
```

Three solvers are derived from this base class: `SEP.solver.smp`, `SEP.solver.reg`, and `SEP.solver.prec`. These classes simply form the objects needed by the solver class.

OUT-OF-CORE EXTENSION

The vector and operator classes described above were extended to perform an out-of-core inversion.

Vector

From the `SEP.vector.vector` class, and a class that knows how to read SEPlib description files (`SEP.file.sep_file`), the `SEP.sep_vector.sfloat` and `SEP.sep_vector.scmplx` were created. These classes use the SEPlib program `Math` to do the required vector operation. The space cloning and check is done by accessing functions derived from `SEP.file.sep_file`.

Operator

Out of core operations are done by the `SEP.operator.oc`. This class inherits from both the `SEP.operator.operator` class and `SEP.operator.run_oc`. The `SEP.operator.run_oc` class handles building the command line. The class has no required options but several optional arguments.

pars A `SEP.par.sep_pars` object containing the list of command line arguments to run with operator call.

add_com The argument to add to the function call when adding (defaults to `add=y`).

restart_com The argument to add to the function call when restarting.

adjoint_com The argument to add to the function call when running the adjoint (defaults to `adj=y`)

data_tag The tag associated with the data (defaults to `data=`)

model_tag The tag associated with the model (defaults to `model=`)

The `SEP.operator.oc` is initialized by the name of the program to run and a domain and range vectors of the type `sep_vector`, and arbitrary additional arguments. The additional arguments are passed to initialize the `SEP.perator.oc_run`. The `adjoint_op` and `forward_op` use the `SEP.operator.oc_run` class to build the commands string and then executes the program.

Parallel operator

To understand this section it useful to refer to Clapp (2004). If an operator is in fact a parallel operator then the `SEP.operator.oc_par` class should be used. This class inherits from the `SEP.operator.oc`. The parallel operator is initialized with a parallel job object. It expects the domain and range vectors to have been specified in the job creation with the keys ‘domain’ and ‘range’. The forward and adjoint operation function are defined in the same manner described above with the exception that domain and range vectors are turned into `parfile` objects and running the job involves executing `parjob.run`.

EXAMPLE

This example is taken from Claerbout (1999) and is completely inappropriate for an out-of-core solver. It is a simple 1-D interpolation problem that takes a second to run on a modern computer and takes 5 minutes to run out-of-core. I am using it because of the simplicity of the concepts. We begin by grabbing all of the command line arguments and checking to make sure `data` and `model` are specified.

```
args=SEP.par.sep_args() #get arguments
dname=args.par("data",error=1)
mname=args.par("model",error=1)
```

We then create the data and model float vectors. In the case of the data is read from a file and the model is defined.

```
data =SEP.sep_vector.sfloat(tag=dname)
model=SEP.sep_vector.sfloat(name=mname)
model.set_axis(200,0.,.4)
```

We then write out the description of the model to disk and zero the file.

```
model.write_file()
model.zero()
```

Our data fitting operator is simple linear interpolation. It takes the coordinates of the data as input and we direct the stdout to `/dev/null`.

```
interp_args=SEP.par.sep_pars(name="interp")
interp_args.add_string("coord=coord.H")
interp_args.add_param(">","/dev/null")
```

We create the argument list for the preconditioner. In this case we just need to direct stdout to `/dev/null` and specify the type of filter.

```
precpar=SEP.par.sep_pars(name="devnull")
precpar.add_param("filter",2)0
precpar.add_param(">","/dev/null")0
```

Next we create our operator objects. In both cases the domain vector is the model. In the case of the interpolation operator the range is the data. For the preconditioning the range is also described by the model space.

```
interp=SEP.operator.oc("Interp1.x",model,data,pars=interp_args)
prec=SEP.operator.oc("prec.x",model,model,pars=precpar)
```

We create our step function.

```
step=SEP.cgstep.cgstep("matcg")
```

Finally we create our solver object and iterate for 20 iterations. Figure 1 shows the result of the inversion. The '*' are the known data and the line is the estimated curve.

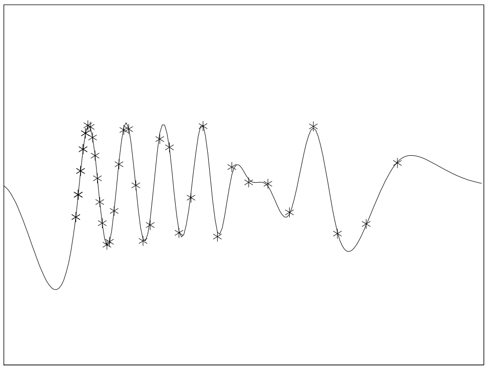

Figure 1: A simple linear interpolation result unsing the out-of-core library.The '*' are the known data and the line is the estimated curve. bob2-inverse [CR]

```
solver=SEP.solver.prec("interp1d",model,data,interp,prec,step,20,.1,verb=0)
if not solver.solve():
  SEP.util.err( "Trouble iterating")
```

CONCLUSION

Fortran 90 is ill-suited for handling complex inversion problems and dealing with hardware instability for multi-week jobs. Using Python and treating programs as operators offers an attractive alternative to completely rewriting code.

REFERENCES

Claerbout, J., 1999, Geophysical estimation by example: Environmental soundings image enhancement: Stanford Exploration Project, `http://sepwww.stanford.edu/sep/prof/`.

Clapp, R. G., and Brown, M., 1999, Applying SEP's latest tricks to the multiple suppression problem: SEP–**102**, 91–100.

Clapp, M. L., 2003a, Directions in 3-D imaging - Strike, dip, both?: SEP–**113**, 363–368.

Clapp, M. L., 2003b, Velocity sensitivity of subsalt imaging through regularized inversion: SEP–**114**, 57–66.

Clapp, R. G., 2004, Fault tolerant parallel SEPlib: SEP–**117**.

Deng, H. L., Gouveia, W., and Scales, J. A., 1996, The cwp object-oriented optimization library: http://www.cwp.mines.edu/html_reports/coool/main.html.

Gockenbach, M. S., and Symes, W. W., 1999, The hilbert class library: http://www.trip.caam.rice.edu/txt/hcldoc/html/index.html.

Guitton, A., 2000, Implementation of a nonlinear solver for minimizing the Huber norm: SEP–**103**, 281–289.

Harlan, W. S., 2004a, C++ implementation of gauss-newton and conjugate-gradients optimization: http://billharlan.com/pub/code/conjugate_gradients/.

Harlan, W. S., 2004b, Gauss-newton and conjugate-gradient optimization: http://billharlan.com/pub/code/inv/.

Sava, P., and Biondi, B., 2003, Wave-equation mva: Born rytov and beyond: SEP–**114**, 83–94.

Sava, P., 2001, oclib: An out-of-core optimization library: SEP–**108**, 199–224.

Schwab, M., and Schroeder, J., 1997, A seismic inversion library in Java: SEP–**94**, 363–381.

Schwab, M., 1998, Enhancement of discontinuities in seismic 3-D images using a Java estimation library: Ph.D. thesis, Stanford University.