Designing an object-oriented library for large scale iterative inversion

Chris Leader and Robert Clapp

ABSTRACT

A flexible library that allows the user to apply a variety of geophysical imaging/inversion techniques and leverage a selection of solvers can be a very powerful tool. However, constructing this to work in multiple dimensions and with a variety of options is a difficult task. The abstraction provided by object-oriented languages helps us to separate the geophysics from the solver, to use the same function calls for models of different dimensions and to create a single framework that has the potential to apply a range of imaging or inversion methods on heterogeneous computing systems.

INTRODUCTION

Geophysical inversion acts to 'un-do' (or 'invert') certain physics of the Earth in order to estimate Earth properties. A data-set is provided that measures perturbations which relate to subsurface properties or interactions - these could be well-log data, GPS data, tilt data, seismic data, etc. Typically one can then forward model or simulate these physics over an estimated model and attempt to estimate our provided data set. The closer we get, the better our model and physics approximations. An inverse process endeavours to improve this model such that our recorded data are better represented. Given data sets can often be tera-bytes in size, and the appropriate model sampling can result in models that are giga-bytes or tera-bytes in size. Thus matrix-based processes become impractical and we must turn to iterative methods (Jupp and Vozoff, 1975).

Using object-oriented programming, we can separate the physics from the solver. This can be done using C++ (Nichols et al., 1993), Fortran 2003 (Clapp, 2010), java (Schwab, 1998) or Python (Clapp, 2005), amongst others. Furthermore, any contemporary iterative imaging and inversion methods have the same basic mathematical functions at their core - these are two-way wave propagation, correlation (imaging and extended imaging) and the vector operations (essentially a series of dot products) that comprise our solvers. A single library that contains these operations (and other necessary, less computationally intensive functions) can be constructed to permit minimal additional coding or effort needed to perform, say, Reverse Time Migration (RTM, Baysal et al. (1983)), linear inversion (occasionally called LSRTM, Nemeth et al. (1999)) or Full Waveform Inversion (FWI, Tarantola (1984)). This library can also

be designed to accommodate modern distributed networks and heterogeneous computing, including GPUs (Foltinek et al., 2009). This is especially desirable for a group such as SEP, who possess several heterogeneous computing systems and perform many techniques that contain wave propagation and wavefield correlation at their core.

There has been an ongoing effort at SEP to combine our common interests into one framework and the approach discussed herein has used C++ and an appropriate class heirarchy. This paper will discuss aspects of this class-based library, the benefits of this construction, and where it is headed in the future.

OBJECT ORIENTED INVERSION

There are several methods to solve geophysical inverse problems. The majority fall under two categories - statistical methods (Duijndam, 1988) or gradient methods (Claerbout, 1999). Statistical/Bayesian methods can be useful when combining multiple sorts of data or for problems that typically contain very rough objective surfaces. A major advantage is that they allow us to include a priori information about our model. In gradient based methods we aim to find the model $\mathbf{m}$ that minimizes the difference between our estimated data, $\mathbf{d}_{est}$, and our observed data, $\mathbf{d}_{obs}$. This amounts to trying to minimise our residual vector, $\mathbf{r}$, defined as

$$\mathbf{Lm} - \mathbf{d}_{obs} = \mathbf{r} \approx \mathbf{0} \tag{1}$$

where $\mathbf{L}$ is a linear operator. In the simplest case we can use $\mathbf{L}$ and its adjoint $\mathbf{L}'$ in a solver; this will be discussed in more detail in the next section.

Our object oriented library seeks to solve a variety of problems of this nature. $\mathbf{L}$ does not necessarily have to be linear in this inversion scheme. Provided we use a non-linear solver (and typically multiple forward modeling processes), this same framework can be used to solve non-linear geophysical inverse problems.

By defining our vectors ($\mathbf{m}$ and $\mathbf{d}$) and operators ($\mathbf{L}$, which could be a cascade of, say, $\mathbf{A}$ and $\mathbf{B}$) in an object-oriented fashion, we can easily separate the physics, $\mathbf{L}$, from the update scheme. Such a method also keeps the coding simple and flexible. Furthermore, we can use a basic set-up and framework for applying many different geophysical techniques (RTM, LSRTM, FWI, etc). Especially since RTM is simply the first gradient of LSRTM, and LSRTM is essentially the inner-loop in FWI.

SOLVERS

The simplest solver is steepest descent. More complex solvers are almost all built on this basic approach, so we can use this as a case study for our library. Assuming our initial residual, $\mathbf{r}$, is simply $-\mathbf{d}$ (zero starting model), we apply the adjoint of our operator $\mathbf{L}$ to obtain our gradient vector $\mathbf{g}$. Next we map this gradient back into the

data-space by applying $\mathbf{L}$, obtaining $\mathbf{rr}$. We then perform a series of dot-products to estimate a scaling factor, α, that makes $\mathbf{r} + \mathbf{rr}$ as small as possible. This process is repeated until $\mathbf{r}$ is sufficiently small or we run out of time. This can be summarised as

Algorithm 1 General linearised inversion

$\mathbf{r} = \mathbf{F}\mathbf{m}_0 - \mathbf{d}_{obs}$
while iter $<$ niter; iter++ **do**
  $\mathbf{g} = \mathbf{F}'\mathbf{r}$
  $\mathbf{rr} = \mathbf{F}\mathbf{g}\mathbf{g}$
  $(\mathbf{m}, \mathbf{r})$ =stepper$(\mathbf{m}, \mathbf{r}, \mathbf{g}, \mathbf{rr})$
end while
Output $\mathbf{m}$

where 'stepper' is our chosen solver.

Algorithm 2 Steepest descent (a possible 'stepper')

$\alpha = -\text{dotproduct}(\mathbf{g}, \mathbf{rr})/\text{dotproduct}(\mathbf{g}, \mathbf{g})$
$\mathbf{m} = \mathbf{m} + \alpha\mathbf{g}$
$\mathbf{r} = \mathbf{r} + \alpha\mathbf{rr}$

As the pseudo-code suggests, the solver acts to update the model and the residual by using $\mathbf{g}$ and $\mathbf{rr}$. These are each scaled by α and then summed to $\mathbf{m}$ and $\mathbf{r}$, respectively. This solver does not need to know $\mathbf{L}$, or any detailed properties about $\mathbf{m}$ or $\mathbf{d}$. In the case of updating the residual it only needs to calculate a dot-product, to scale $\mathbf{rr}$, and to add this vector to $\mathbf{r}$.

More complicated solvers are generally comprised of these basic operations; they often just remember details of previous scaling factors and gradients. For example, the code for conjugate directions is only a few lines longer and uses the same three functions.

IMPLEMENTATION IN C++

Using C++ we can construct a hierarchy of classes. Each class has a series of functions and variables that can be private or shared, and by making some functions abstract we can name a series of methods that others classes can use and define. This paper will not go into detail about all the classes and the hierarchy of the library, but it will mention the vector and operator classes since these comprise our solver.

We know that for our linear solver we need three operations - dot-product, scale and add. These are all functions that can be placed within our vector class, and by keeping a level of abstraction we can use these same operations and function calls to act on arrays of different dimensions. Other operations we can define in 'vector' are 'clone,' 'zero,' and 'set.' These six operations allow us to do most of the necessary

operations on the vectors within our inversion. It is wise to also include some vector 'tests', such as checking whether two vectors being operated on belong to the same space.

The operator class is constructed in a similar manner, although it is a little simpler. In this case we define a forward and an adjoint, and then allow two functions that set the 'domain' of the operator and the 'range'. These are essentially the dimensions of the model and the data vectors, respectively. Calling an operator has three inputs and a fourth optional input. Inputs are 'add' (whether we want to zero the output array or not), then the vector 'model' and the vector 'data.' The optional input is a iteration number argument, as some techniques require the current iteration number.

Setting up our classes as such results in simple, readable code for the inversion. The exact code for the solver looks as follows:

```
my_vector *r=data->clone_vec();
r->scale(-1.);
for(iter=0; iter < niter; iter++){
  oper->adjoint(false,g,r,iter);
  oper->forward(false,g,rr,iter);
  bool valid=st->steepest_descent(iter,m,r,g,rr,&val);
}
```

```
alpha=-r->dot(rr)/rr->dot(rr);
s->scale(g,alpha);
ss->scale(rr,alpha);
m->add(s);
r->scale(ss);
```

this keeps things simple and readable, and the actual code looks almost the same as the pseudo-code discussed in the previous section. Note that the vectors here could be 2D, 3D or higher (provided they have the same number of dimensions as each other) and the operator could be any forward-adjoint pair. For example, this code snippet could be performing a basic radon inversion, or FWI.

With the operators coded and the various dimensions and data initialized, calling the solver and producing a result is as simple as:

```
step *st=new cgstep();
solver *solv=new solver(st,&data,&operator);
solv->solve(no_iter);
oc_float *result=(oc_float*) solv->return_model();
image.add(result);
```

HETEROGENEOUS COMPUTING

We must also design these codes to work on a selection of computers, in particular a single core node, a multi-core node, a multi-node network, single GPUs, multiple GPUs on a single node, multiple GPUs across nodes, and out-of-core. It is possible to keep a lot of details about the computing out of the majority of the operator and solver code, however the distributed network code involves a lot more work with MPI.

For single node solutions (single or multi-core, single or multiple GPU) it is only the propagation and the correlation that must be distributed across units. The solver code does not have to change. Furthermore, the details of the decomposition can be kept separate within these aforementioned routines. Provided the code is told which method is desired, the majority of the operator does not have to change either.

Currently the library works with single and multiple GPUs on a given node. This provides an incredible amount of computation power with few of the classic GPU memory restrictions. Whether the user wants to use a single or multiple devices is simply a command line argument. For multiple GPUs domain decomposition is performed (Micikevicius, 2012); however all details of this are kept out of the operator and solver codes. Only one code, gpu_funcs_3d.cu has any knowledge of how this is done, making the majority of the library very clean and readable.

Efficient CPU based multi-core and multi-node implementations are currently being improved upon as part of a SEP initiative. Introducing them to the system is very simple; they can either be included in `gpu_funcs_3d.cu` or a separate file, then whichever method is used can be determined by an input argument. Keeping these numerically intensive aspects somewhat separate from the operators further helps to make the code flexible.

PHASE ENCODING

Extending the code to use phase encoded inversion (Gao et al., 2010; Morton and Ober, 1998) is less straightforward. In phase encoded inversion we combine groups of shots into supershots, and then aim to recover a common model between them. In the most extreme case we can combine all shots into one supershot, so that we only perform two forward model operations and one adjoint operation per iteration. By changing the encoding matrix that we use between iterations, very fast model convergence (as a function of cost) can be seen (Krebs et al., 2009; Romero et al., 2000).

The inversion set-up needs knowledge of two data-sets: the original/separated data-set and our encoded dataset. However our current operator class only gives us the option of passing one data vector. This means we have to introduce two data vectors within our phase encoded operator routine in the forward routine. This is not a difficult task, but it limits the flexibility of using the same code as LSRTM for phase encoded LSRTM without introducing a series of if statements.

CONCLUSIONS

An object oriented framework creates a flexible, clean and readable inversion library. The class system means our solver can neatly deal with vectors of multiple dimensions while the internal workings of our operator (the geophysics) is kept separate. The details of how the propagation and correlation are performed on the network are kept separate again from the main operator construction. This creates further flexibility for general operator coding and means only a few command line arguments can be used to greatly vary how the computationally intensive aspects of the inversion are dealt with.

FUTURE WORK

There are many ways of extending this library in the future. Current effort is being put into multi-core and multi-node solutions for propagation and correlation, into ways of accelerating extended imaging, and into non-linear solvers.

REFERENCES

Baysal, E., D. Kosloff, and J. Sherwood, 1983, Reverse time migration: Geophysics, **45**, 1514–1524.

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

Clapp, R., 2010, Hybrid-norm and fortran 2003: Separating the physics from the solver: SEP, **142**.

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

Duijndam, A., 1988, Bayesian estimation in seismic inversion. part i: Principles1: Geophysical Prospecting, **36**, 878–898.

Foltinek, D., D. Eaton, J. Mahovsky, P. Moghaddam, and R. McGarry, 2009, Industry scale reverse time migration on GPU hardware: SEG Technical Program Expanded Abstracts, **28**, 2789–2793.

Gao, F., A. Atle, and P. Williamson, 2010, Full waveform inversion using deterministic source encoding: SEG Technical Program Expanded Abstracts, **29**, 1013–1017.

Jupp, D. L. B. and K. Vozoff, 1975, Stable iterative methods for the inversion of geophysical data: Geophysical Journal of the Royal Astronomical Society, **42**, 957–976.

Krebs, J. R., J. E. Anderson, D. Hinkley, R. Neelamani, S. Lee, A. Baumstein, and M.-D. Lacasse, 2009, Fast full-wavefield seismic inversion using encoded sources: Geophysics, **74**, WCC177–WCC188.

Micikevicius, P., 2012, Multi-GPU programming: Presentation at GPU Technology Conference 2012.

Morton, S. A. and C. C. Ober, 1998, Fastershot-record depth migrations using phase encoding: SEG Technical Program Expanded Abstracts, **17**, 1131–1134.

Nemeth, T., C. Wu, and G. Schuster, 1999, Least squares migration of incomplete reflection data: Geophysics, **64**, 208–221.

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.

Romero, L. A., D. C. Ghiglia, C. C. Ober, and S. A. Morton, 2000, Phase encoding of shot records in prestack migration: Geophysics, **65**, 426–436.

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

Tarantola, A., 1984, Inversion of seismic reflection data in the acoustic approximation: Geophysics, **49**, 1259–1266.