A flexible out-of-core solver for linear/non-linear problems

Ettore Biondi and Guillaume Barnier

ABSTRACT

Based on the SEPlib non-linear solver library described previously, we implement an out-of-core Python-based generic solver for linear/non-linear inverse problems. We closely follow the object-oriented structure of the previous library and add new features in order to handle any programming language describing the application of user-defined operators. Our out-of-core feature addresses many treating memory-intensive inverse problems frequently encountered in geophysical applications. We demonstrate the new solver capability with simple linear problems and a non-linear acoustic isotropic full-waveform inversion (FWI) example.

INTRODUCTION

Geophysicists deal with inverse problems routinely. Many operations such as deconvolution or velocity estimation are inverse problems in disguise (Claerbout, 1992). Clearly it is advantageous to have at our disposal a generic solver capable of handling most of the inverse problems encountered. Métivier and Brossier (2016) present a generic non-linear solver written in *FORTRAN 90* based on reverse communication. Although their library offers many methods to solve a given inverse problem, it lacks the flexibility of being able to work with programming languages different than *FORTRAN*. This constraint is greatly restrictive. In fact, nowadays modeling/processing routines are written in many different programming languages and combined together to fully exploit the available computation architecture (e.g., core libraries written in *C++* or *CUDA* handled by a *Python* or *Shell* interface). For this reason we chose to implement a generic solver written in the freeware scripting language *Python* that can be used for any inverse problem in which its core operators are coded with any programming language. We achieved this goal by using a *system-called operator* concept. In addition, the presented solver is implemented using an out-of-core philosophy, making it capable of handling memory-intensive inverse problems.

We first start by summarizing the object-orient structure of the solver, which is closely related to the one present in Almomin et al. (2015). Secondly, we explain the concept of a system-called operator that allows us to combine the solver with any other programming language. Finally, we describe the usage of the generic linear and non-linear solvers along with a few examples demonstrating their capabilities.

SOLVER STRUCTURE

The great flexibility of the object-oriented structure of the library described in Almomin et al. (2015) is also found in this new solver implementation. In fact, we decided to retain the same class structure:

- problem class: object describing the inverse problem to solver,
- solver class: object that takes problem class and solves the given problem,
- stepper class: object containing the method to use in case a line search is necessary,
- stopper class: object that quits the inversion in case any stopping criterion is met.

This implementation allows us to define any inverse problem and separate it from the method that we want to utilize for finding the problem's solution. The only requirement is that the problem has to be defined as an indirect problem (i.e., the solution has to be found using an objective function (Weglein et al., 2003)). Therefore, the problem class has to contain an objective function method. If a gradient-based method is employed, then linear forward and adjoint operator methods have to be present. We will explain this necessity when describing the generic non-linear solver. The stepper class is used whenever a line search algorithm is necessary to perform an optimization step during the inversion (e.g., whenever we are treating a non-linear problem). The definition of this class is avoidable when a linear problem has to be solved because step lengths are usually already embedded into the solver algorithm (e.g., the conjugate-gradient (CG) method (Aster et al., 2005)). Lastly, the stopper class just defines different stopping criteria that the user can use to stop the inversion. For a more detailed description of the *FORTRAN 2003* version see Almomin et al. (2015).

The typical inverse problem encountered in geophysics is the *L2*-norm inverse problem:

$$\phi(\mathbf{m}) = \frac{1}{2}\|\mathbf{f}(\mathbf{m}) - \mathbf{d}\|_2^2 = \frac{1}{2}\|\mathbf{r}(\mathbf{m})\|_2^2, \tag{1}$$

where $\mathbf{m}$ is the model vector, $\mathbf{f}(\mathbf{m})$ is the modeling operator, $\mathbf{d}$ is the observed data vector, $\mathbf{r}(\mathbf{m})$ is the residual vector, and $\phi(\mathbf{m})$ is the objective function to be minimized. The modeling operator can be either linear or non-linear. Obviously, the two problems are treated with different solvers. The two proposed solvers minimize the objective function shown in equation 1. In addition, in the current solver version, we also included the minimization of a *L2*-norm regularized problem defined as follows:

$$\phi_{reg}(\mathbf{m}) = \frac{1}{2}\|\mathbf{f}(\mathbf{m}) - \mathbf{d}\|_2^2 + \frac{\epsilon^2}{2}\|\mathbf{A}(\mathbf{m} - \mathbf{m}_{ref})\|_2^2 = \frac{1}{2}\|\mathbf{r}_d(\mathbf{m})\|_2^2 + \frac{\epsilon^2}{2}\|\mathbf{r}_m(\mathbf{m})\|_2^2, \tag{2}$$

where $\mathbf{A}$ is a linear regularization operator, and $\mathbf{m}_{ref}$ is a reference model.
In both cases, when a gradient-based method is used to optimize these objective function we have to compute their gradients as follows:

$$\nabla\phi(\mathbf{m}) = \left(\frac{\partial \mathbf{f}(\mathbf{m})}{\partial \mathbf{m}}\right)^* \mathbf{r}(\mathbf{m}), \tag{3}$$

$$\nabla\phi_{reg}(\mathbf{m}) = \left(\frac{\partial \mathbf{f}(\mathbf{m})}{\partial \mathbf{m}}\right)^* \mathbf{r}_d(\mathbf{m}) + \epsilon^2 \mathbf{A}^*\mathbf{A}\mathbf{r}_m(\mathbf{m}), \tag{4}$$

where * denotes the adjoint operation. We now turn our attention to the concept with which we are able to perform an out-of-core language-independent inversion.

THE SYSTEM-CALLED OPERATOR CONCEPT

From the previous equations we notice that the most complex operation to be performed is applying different operators (i.e., the modeling operator $\mathbf{f}(\mathbf{m})$, its linearization $\left(\frac{\partial \mathbf{f}(\mathbf{m})}{\partial \mathbf{m}}\right)$, and the linear regularization operator $\mathbf{A}$ forward and adjoint). In fact, in order to solve our inverse problem, we just need to tell the computer how to apply different operators to given vectors. To perform simple vector operations we use python-system calls to the standard SEPlib program *Solver_ops*; on the other hand, to apply the different required operators we define a new object called *operator*.

Whenever we create a new operator object we have to specify a series of terminal/shell commands that describes the application of the given operator to a provided input file and places the result of this operation into an output file. With this template the computer knows how to apply any supplied operator to a given vector and where the result will be placed. Therefore, to perform any inversion we have to just specify the solver the set of commands that we would run from terminal to apply the operators for our inverse problem into parameter files. As an example the following lines describe the application of a generic forward modeling operator that employs SEP-formatted files with the solver-operator definition:

```
#This is a comment
#Applying forward modeling operator
RUN: ./bin/forward.x < input.H par=Par/parfile.p > output.H
```

Note that in front of the command that has to be run it is placed the keyword `RUN:`. This syntax tells the operator object how to parse the parameter file that describes the given operator. In this case, we see that whenever the solver needs to apply the forward modeling operator $\mathbf{f}(\mathbf{m})$, it has to substitute `input.H` and `output.H` with the actual input-file and output-file names, respectively.

The operator object handles multiple lines when the operator is composed as a set of commands. In fact, a more complex operator could be the following:

```
#Applying foward modeling operator as a series of commands
RUN: ./bin/forward1.x < input.H par=Par/parfile.p > tmp1.H
RUN: ./bin/forward2.x < input.H par=Par/parfile.p > tmp2.H
#Combining the results of the previous commands
RUN: ./bin/combine.x file1=tmp1.H file2=tmp2.H output=output.H
#Cleaning temporary files
RUN: rm tmp1.H tmp2.H
```

In this case, the operator is composed of two different programs where their outputs are combined after their application. Note that the commands are run sequentially. Hence, whenever we start a command the subsequent one is going to be run after exit code of this process is returned. As we can see from this description, there is no assumption on what programming language the user employed to code their executables. In fact, with this definition of system-called operator, the python solver can handle operators written in any programming language. In addition, because the solver uses filenames, it is not necessary to store any array inside the solver itself. The actual python class implementation is in file `operator_obj.py` contained in the folder `python_solver/` of the related *zip* file of this report.

GENERIC LINEAR SOLVER AND EXAMPLES

In this section we provide a brief explanation of how to use the generic linear solver. The name of the python script is `generic_linear_prob.py` and can be found in the folder `python_solver/`. If the script is run without any input it provides a documentation of all the possible parameters that can be used during a linear inversion. Before running any inversion it is necessary to run `make install` from the main folder, which compiles two necessary programs related to the solver, and add the screen-printed path to the user's environment variable *PYTHONPATH*. This operation has to be performed only once. A simple *L2*-norm linear inversion can be performed by calling the script in the following manner:

```
./python_solver/generic_linear_prob.py fwd_cmd_file=./Par/tmp_fwd.txt \
adj_cmd_file=./Par/tmp_adj.txt data=data.H init_model=init_model.H \
inv_model=inv_model.H suffix=_problem iteration_movies=obj,model,residual \
niter=10 log_file=Generic_lin_test.txt wrk_dir=inversion_test
```

Because we want to solve a non-regularized linear problem we just need to specify the linear forward and adjoint operators (i.e., $\mathbf{f}(\mathbf{m}) = \mathbf{Fm}$, and $\mathbf{F}^*$ necessary for the gradient computation). These two operators are provided in the command line with `fwd_cmd_file=./Par/tmp_fwd.txt` and `adj_cmd_file=./Par/tmp_adj.txt` parameters, respectively. All inversion files are going to be placed in the work directory `wrk_dir=inversion_test`. Along with the inverted model `inv_model=inv_model.H`, the solver can return separate different files containing the objective function value,

inverted model, residual vector, and gradient as function of iteration number (see parameter `iteration_movies`). In the logfile `log_file=Generic_lin_test.txt` the user can find information about the inversion run, such as step lengths, elapsed time, and others.

Simple matrix tests

In this report's *Makefile* there are several simple matrix tests that can be run to understand the solver usage. Here, we report three simple matrix tests.

In the first test we want to minimize the objective function of equation 1 when the forward operator is a lower-triangular matrix of size 100 by 100 elements with linear CG method. The makerule to run this example is *Generic_lin_test*. Figure 1a shows the operator matrix used during the inversion. To ensure an invertible linear system, the main diagonal of this matrix is different than zero. The data vector is simply constant and equal to one. After running the solver for 50 iterations the initial objective function value decreases by more 99.4 percent (Figure 1b). Figure 1c displays the residual vector at the last iteration. We can see that the residual elements oscillate around zero and that the first element is furthest from the true data. This behaviour is probably due to the matrix structure chosen in this test.

In our second linear test we minimize the objective function of equation 2 for the same matrix of our first example but adding a model-norm regularization term. As discussed by Claerbout (2014), there are different ways for estimating a balancing weight for the regularization term. We implement the solver so that it can provide an estimated ϵ that balances the first gradient $\mathbf{g}(\mathbf{m}_0)$ in the data space:

$$\epsilon = \|\mathbf{F}\mathbf{g}(\mathbf{m}_0)\| \,/\, \|\mathbf{A}\mathbf{g}(\mathbf{m}_0)\| \,. \tag{5}$$

With this ϵ definition we correctly balance the two objective function components by using a model (the gradient) that has proper model-scale size (i.e., numerical values). In fact, the step-length scaling is removed by the division of the two factors. Moreover, this approach prevents the denominator of this equation to be zero when the user-provided initial model is contained in the null space of the operator $\mathbf{A}$. In the described example we use a regularization weight of 1.0. Because of the regularization term, the solver cannot find a step length after 45 iteration and stops when the total objective function decreases by 98.79848 percent (Figure 2a). The behavior of the residual vector at the last iteration is similar to that of the previous test (Figure 2b). The make rule to run this example is *Generic_lin_reg_test*.

In the last linear test using matrices, we want to solve a symmetric negative definite system of equations. In this case the objective function to be solved is the following (Aster et al., 2005):

$$\phi_{sym}(\mathbf{m}) = \frac{1}{2}\mathbf{m}^*\mathbf{H}\mathbf{m} - \mathbf{m}^*\mathbf{d}, \tag{6}$$

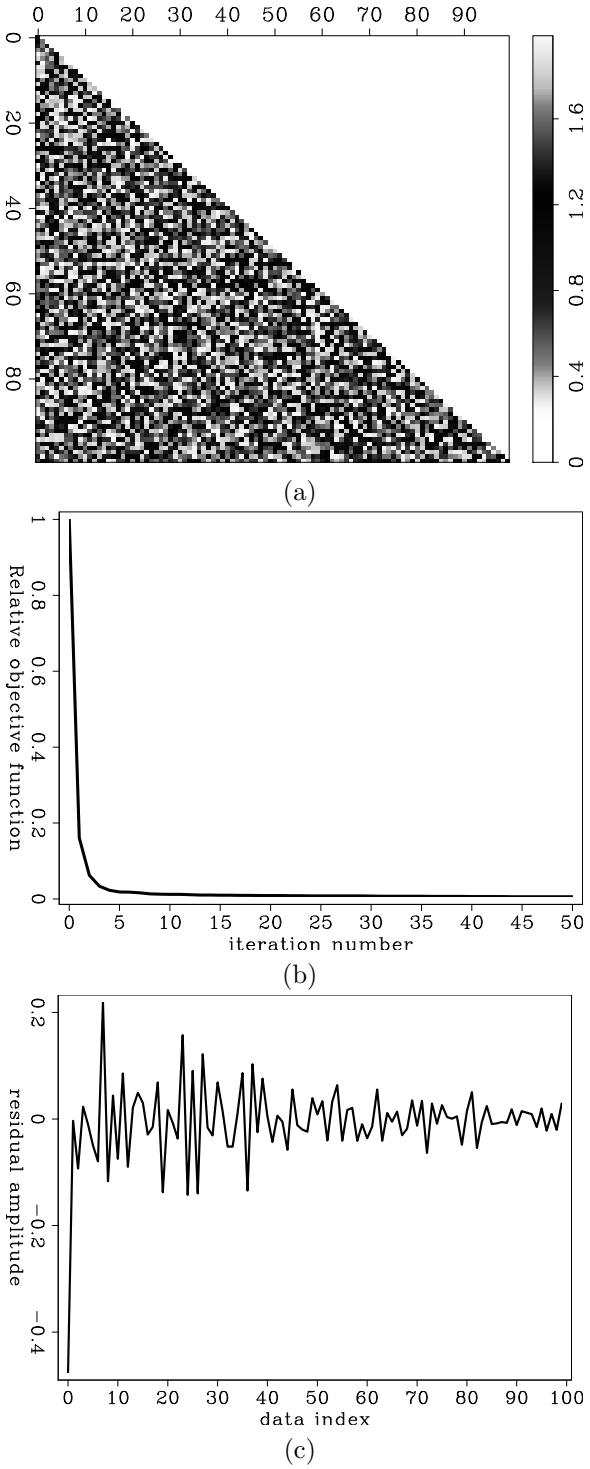
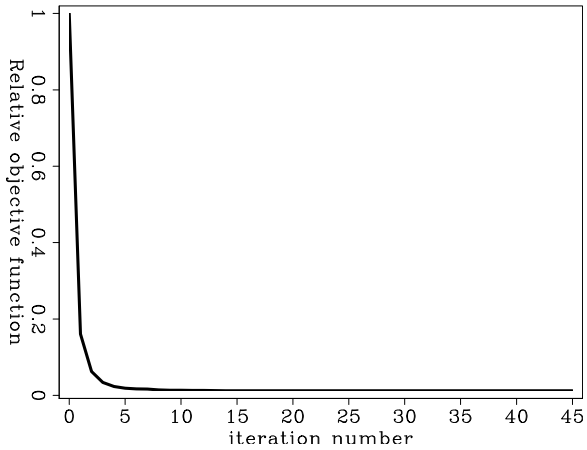
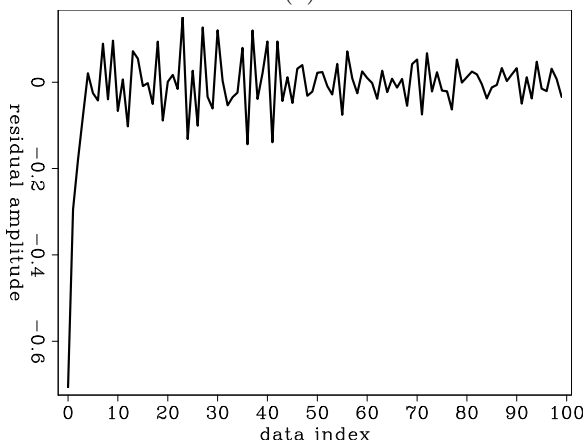


Figure 1: (a) Lower-triangular square matrix used in the first linear inversion test. (b) Relative objective function value as function of iteration number. (c) Residual vector at the last iteration. [**ER**]



(a)



(b)

Figure 2: (a) Relative objective function value as function of iteration number. (b) Residual vector at the last iteration. Both plots belong to the second linear test described in the text. [**ER**]

where **H** is a symmetric positive or negative definite matrix. If the matrix is positive definite, this objective function is monotonically decreasing during the inversion. For this test we choose a second-order derivative operator (Figure 3a). Because of the boundary conditions employed the system is invertible. For data vector **d** we again use a constant value of one for all the elements. This choice results in finding a function whose second-order derivative is always constant and positive (i.e., a convex parabola). Indeed, this is the result that we obtain after running for 44 iterations the linear symmetric system CG Figure 3b. To run this example the makerule is *Generic_lin_test_symmetric*.

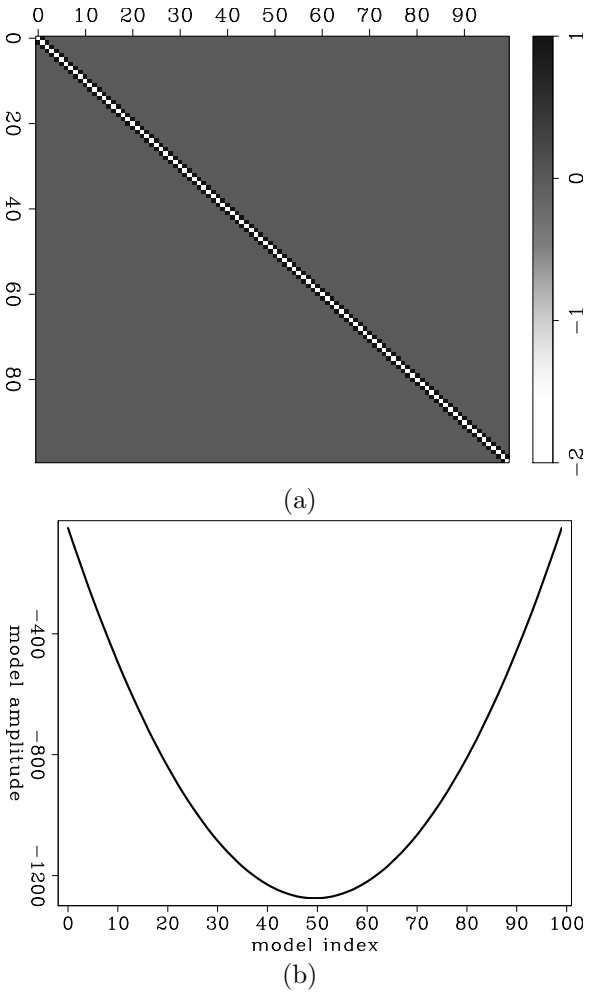


Figure 3: (a) Symmetric negative definite matrix used in the third linear inversion test. (b) Inverted model at the last iteration. [**ER**]

GENERIC NON-LINEAR SOLVER AND EXAMPLES

The same installation procedure applies for the generic non-linear solver. To run a non-linear inversion we ensure that whenever we are evaluating the gradient of

the objective function we have to apply the adjoint linearized operator to the data residuals (see equations 3 and 4). In addition, we need to specify the forward linearized operator that is used to estimate the initial step length for any line search algorithm. Indeed, as input parameters the user has to provide three different operator-template parameter files. A simple call of the non-linear solver takes the following form:

```
./python_solver/generic_non_linear_prob.py  \
fwd_nl_cmd_file=${P}/nl_fwd.txt fwd_cmd_file=${P}/lin_fwd.txt \
adj_cmd_file=${P}/lin_adj.txt data=data.H init_model=init_model.H \
inv_model=inv_model.H suffix=_nlproblem \
iteration_movies=obj,model,residual niter=10 \
log_file=inversion_log.txt wrk_dir=nl_inversion
```

Aside from this complication, the solver call is similar to the linear one. The required forward linearized operator `fwd_cmd_file` is necessary when an initial step-length guess has to be made during the line-search algorithm. In fact, a simple way to estimate the value of this guess is to linearize the modeling operator $\mathbf{f}(\mathbf{m})$ around the current model $\mathbf{m}_0$ and find the step length α that minimizes the given objective function along the current search direction $\Delta\mathbf{m}$. For instance, this procedure for the objective function in equation 1 would take the following form:

$$\phi(\mathbf{m}_0 + \alpha\Delta\mathbf{m}) = \frac{1}{2}\left\|\mathbf{f}(\mathbf{m}_0) + \alpha\frac{\partial\mathbf{f}(\mathbf{m}_0)}{\partial\mathbf{m}}\Delta\mathbf{m} - \mathbf{d}\right\|_2^2. \tag{7}$$

The linearization of the non-linear operator enables us to analytically compute the optimal step length by minimizing the derivative of the previous objective function with respect to α:

$$\frac{d\phi(\mathbf{m}_0 + \alpha\Delta\mathbf{m})}{d\alpha} = 0 => \alpha = -\frac{\mathbf{r}(\mathbf{m}_0)^*\frac{\partial\mathbf{f}(\mathbf{m}_0)}{\partial\mathbf{m}}\Delta\mathbf{m}}{\left\|\frac{\partial\mathbf{f}(\mathbf{m}_0)}{\partial\mathbf{m}}\Delta\mathbf{m}\right\|_2^2}, \tag{8}$$

where the search direction is projected into the data space through the forward linearized operator.

In the next section we describe an application of the solver in a full waveform inversion problem. Although we do not show here the usage of any regularization term during the inversion, you can see its use in Barnier et al. (2017). In their examples the solver provides them with a balancing weight for their regularization term. In this case, the solver finds an ϵ that balances the two terms in the objective function of equation 2. If the initial model is in the null space of the regularization operator, the solver will perform a steepest-descent search and recompute the estimated weight.

Full waveform inversion

We test the non-linear solver on a synthetic FWI example where we use an isotropic acoustic wave-equation modeling operator. Here, we minimize a simple $L2$-norm data

difference (equation 1). Figure 4a shows the true velocity model used in this test. We place 200 shots and 2000 receivers spaced 100 m and 10 m at the surface, respectively. To generate the observed data we inject a Ricker wavelet of dominant frequency of 15 Hz. We then apply a bandpass filter to th observed data to a maximum frequency of 5 Hz. We invert this bandwidth only in this example. The starting model is generated by smoothing the true model (Figure 4b).

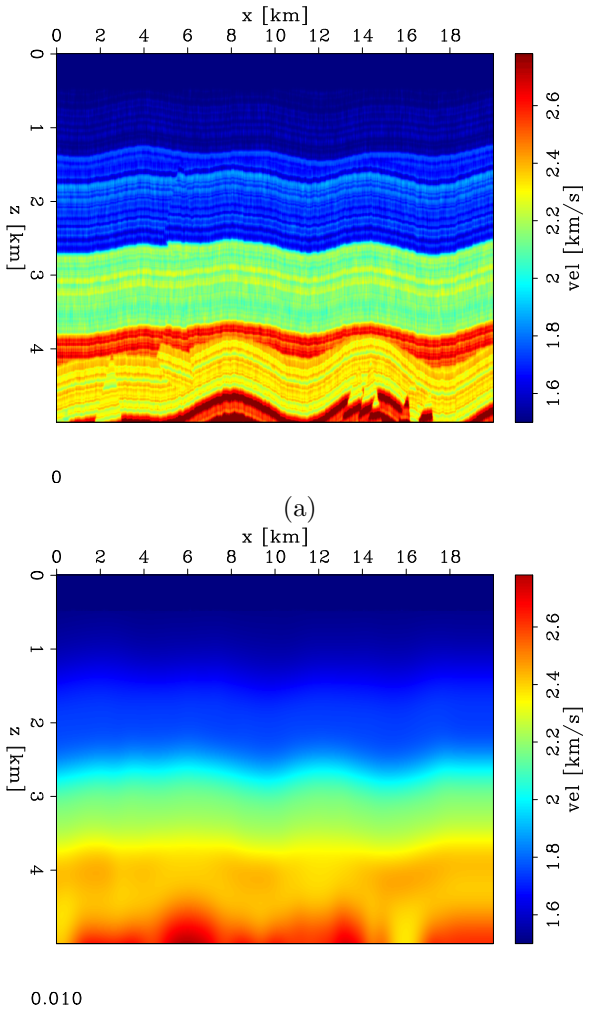


Figure 4: True (a) and initial (b) velocity models used in the described FWI example. [**ER**]

We apply the same preconditioning technique proposed by Biondi et al. (2017) in which an approximated Gauss-Newton Hessian inverse is estimated from model-space Hessian matrix applications. We run 14 iterations of a non-linear conjugate-gradient method (Fletcher and Reeves, 1964). From Figure 5 we clearly distinguish a difference when we precondition the gradient. In fact, this simple approach improves the convergence rate of the algorithm. In terms of inverted models, we also see a noticeable difference (Figure 6). We observe that the preconditioned inverted model presents

more details, especially, in the deeper portion. In this example, the approximated Hessian inverse matrix provides an amplitude-balancing effect when applied to the computed gradient during the inversion.

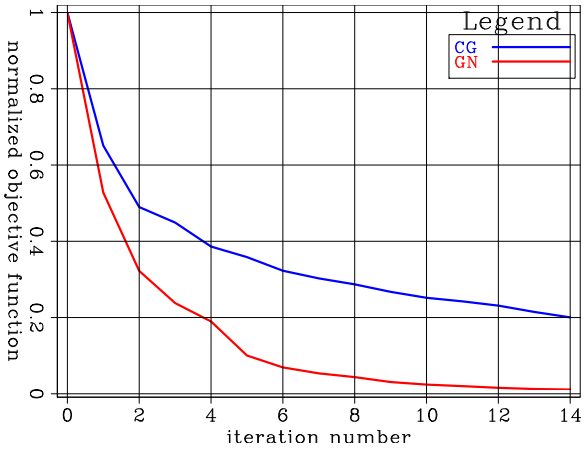


Figure 5: Relative objective function comparison between conjugate-gradient method using an approximated Gauss-Newton Hessian inverse as preconditioner (red curve) and without it (blue curve). [**CR**]

CONCLUSIONS AND FUTURE DIRECTIONS

We present a flexible out-of-core Python-based linear/non-linear solver library that uses of system-called operators to compute applications of these to given vectors. This feature provides the user a solver in which inverse-problem operators can be written with any programming language. Because the solver runs storing only filenames it has the potential of handling memory-demanding problems. We demonstrate its capabilities on different linear and non-linear inverse problems. In all these tests we show the convergence of the solver to a reasonable optimal model.

Currently, the solver employs only linear and non-linear CG methods, and different line-search algorithms are implemented within the non-linear optimization (i.e., sampler, parabolic, and linear steppers). In the future, memory-limited BFGS method is going to be added into the non-linear solver frame. Also, truncated Gauss-Newton and Newton steppers are going to be fully deployed into the solver. Finally, to overcome the limiting factor of applicability to SEP-formatted file we are going to use the *Generic IO* library proposed by Clapp (2017).

REFERENCES

Almomin, A., E. Biondi, Y. Ma, K. Ruan, J. Jennings, R. Clapp, M. Maharramov, and A. Cabrales-Vargas, 2015, SEPLib nonlinear solver library – Manual: SEP-Report, **160**, 39–71.

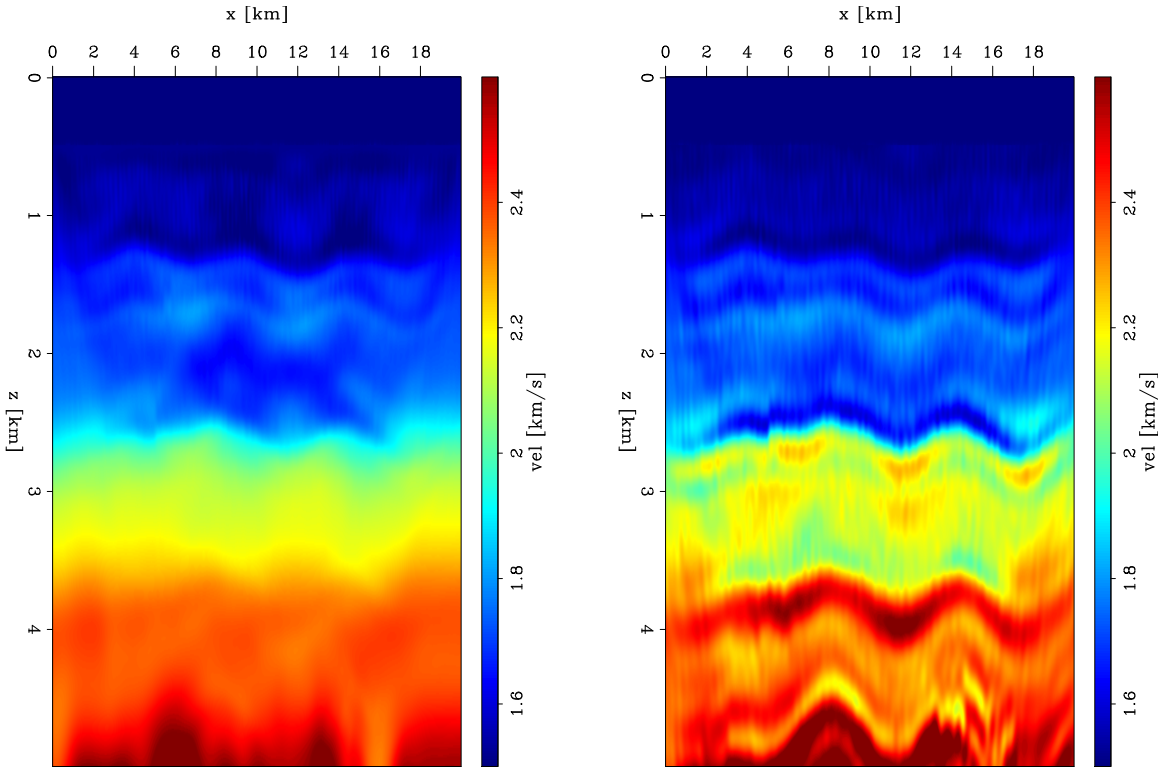


Figure 6: Inversion result comparison. Left panel: inverted model without preconditioner. Right panel: inverted model using an approximated Hessian inverse as preconditioner. [**CR**]

Aster, R., B. Borchers, and C. Thurber, 2005, Parameter Estimation and Inverse Problems: Elsevier.

Barnier, G., E. Biondi, and B. Biondi, 2017, Applying TFWI for imaging under complex overburden: SEP-Report, **168**.

Biondi, E., G. Barnier, and B. Biondi, 2017, Preconditioned elastic FWI: SEP-Report, **168**.

Claerbout, J., 2014, Geophysical image estimation by example: Lulu. com.

Claerbout, J. F., 1992, Earth soundings analysis: Processing versus inversion, volume **6**: Blackwell Scientific Publications, Cambridge, Massachusetts, USA.

Clapp, R., 2017, Generic IO: SEP-Report, **168**.

Fletcher, R. and C. M. Reeves, 1964, Function minimization by conjugate gradients: The computer journal, **7**, 149–154.

Métivier, L. and R. Brossier, 2016, The SEISCOPE optimization toolbox: A large-scale nonlinear optimization library based on reverse communication: Geophysics, **81**, F1–F15.

Weglein, A. B., F. V. Araújo, P. M. Carvalho, R. H. Stolt, K. H. Matson, R. T. Coates, D. Corrigan, D. J. Foster, S. A. Shaw, and H. Zhang, 2003, Inverse scattering series and seismic exploration: Inverse problems, **19**, R27.