A flexible library for geophysical inverse problems – structure and usage

Ettore Biondi, Robert G. Clapp, and Guillaume Barnier

ABSTRACT

We implement an object-oriented inversion library based on the concept of operators that is able to be easily applied to both large- and small-scale inverse problems. By using general mathematical vector and function concepts, we design classes, or objects, that can be used to minimize convex and non-convex objective functions. We report different applications of the library to demonstrate its potential on various inverse problems.

INTRODUCTION

Inverse problems arise in most geophysical applications (Claerbout, 1992; Aster et al., 2005). Therefore, it is important to have at our disposal reliable inversion algorithms that could efficiently find an optimal solution to any inverse problem we pose. In addition, we often test our ideas using simple small-scale inverse problems before tackling the field data scenarios, which we consider to be large-scale inversions where terabytes of data have to be processed. Because of this reason, we are in need of an inversion library that is flexible enough to be applied to both small- and large-scale inverse problems without the need of changing our algorithms, with the risk of introducing errors in the designed workflow.

Many different reliable libraries have been proposed over the years (Byrd et al., 1995; Jones et al., 2001; Grant and Boyd, 2014; Métivier and Brossier, 2016). However, most of these libraries are suited to be applied to specific classes of optimization problems and lack a simple interface for user-defined objective functions and inversion algorithms (e.g., Barnier et al. (2018) and Farris and Biondi (2018)). In addition, the usage of some of these libraries for large-scale inversion problems could be a tedious process; especially, when submission to a cluster of machines is necessary. Finally, some inversion packages that are implemented in a compiled programming language are possibly cumbersome to be applied by a user if their codes are written in a different one.

Based on two previously proposed inversion libraries (Almomin et al., 2015; Biondi and Barnier, 2017), we implement an object-oriented inversion package connected to mathematical concepts such as vectors and operators (i.e., linear or non-linear mapping functions). This abstraction results in a greater extent of applicability and a

simplification of the user interface. We implement the package using *Python* given the wide range of available libraries and language-binding interfaces (e.g., Van Der Walt et al. (2011); Jakob et al. (2016)). The latter feature makes Python an ideal platform to implement a flexible interface that smoothly combines libraries written in different programming languages. First, we describe the main abstract classes composing the library to explain the inversion package's structure. Secondly, we provide an actual code example of how instantiated objects are combined together to minimize a quadratic objective function. Finally, we report optimization examples of linear and non-linear inverse problems, such as linearized and full waveform inversions (FWI).

LIBRARY STRUCTURE

The inversion library is part of a repository that can be cloned by running the command "*git clone http://zapad.Stanford.EDU/ettore88/python-solver.git*" from any terminal window.

The library is suited to linear inverse problems such that storage of a matrix is not feasible (i.e., only its application onto a vector is known), or when its inverse cannot be explicitly computed. Additionally, it can be used to solve non-linear inverse problems. All the implemented Python modules are inside the folder *GenericSolver/python/*, while some usage examples are within the folder *GenericSolver/notebooks/unit_tests/*. Beside Python and the *Numpy* package, the software can be employed as is. When combining *C++* with Python, we rely on *Pybind11* and the genericIO library implemented by Clapp (2017). To obtain the documentation and structure of a module or object the user can run the command "*help(name of module/object)*" within a Python session.

Abstract objects

In this section we provide a description of the main abstract, or virtual, classes defined within the library that enables us to provide general building blocks that, when combined, can be used to minimize any objective function. Since all mathematical problems are inherently discrete when solved within a computer, the first building block is represented by a class encompassing all the properties of vectors and vector spaces.

Vector class

This class represents any collection of discrete points living within a certain space; effectively, any vector $\mathbf{x} \in \mathbb{R}^N$ whose size is of N elements. Moreover, any n-dimensional vector can be represented by this class (e.g., images or cubes of data). For instance, a vector $\mathbf{y} \in \mathbb{R}^{N \times M}$ can represent a 2D image of N by M pixels. The module containing the class definition is within the file *pyVector.py*

When dealing with vectors, not many operations have to be defined. For instance, we need to compute norms of vectors or multiply a vector by a scalar as well as operations that combine two different vectors (e.g., sum of two vectors or element-wise multiplication). Note that once those basics operations are defined for a given vector class (e.g., single-precision, double-precision, and complex vectors), a solver object (later described) does not need to know how those operations are performed. Therefore, with this abstraction we can easily handle in-core or out-of-core vector operations using the exact solver structure and making the library able to handle small- and large-scale inverse problems.

Operator class

This class describe the mapping, whether linear or non-linear, from one vector space to a different one. Its module implementation can be found in the file *pyOperator.py*.

Mathematically, an operator is expressed as $\mathbf{y} = \mathbf{f}(\mathbf{x})$, where the vector $\mathbf{x}$ is mapped by the function $\mathbf{f}$ into the vector $\mathbf{y}$. For linear transformations this expression can take two forms since we can map the vector $\mathbf{x}$ to $\mathbf{y}$ using the forward operator or the other way around by mapping the vector $\mathbf{y}$ to $\mathbf{x}$ by employing the adjoint operator (i.e. $\mathbf{y} = \mathbf{A}\mathbf{x}$ or $\mathbf{x} = \mathbf{A}^*\mathbf{y}$, where * represents the adjoint operator). Therefore, when a user is defining a new linear operator, they should implement both forward and adjoint functions in the derived class from the abstract one. Additionally, whether the mapping is linear or non-linear, in the constructor of the derived class, it is important to define the domain and range of the operators (i.e, the vector spaces in which the mapping operates). By doing so many obvious mistake can be avoided when applying any operator. To see an example of a derived class, the reader can refer to the matrix-vector multiplication operator defined in the file *Utest_LCGsolver.py* within the unit_tests folder.

Moreover, the abstract operator class contains useful functions that could help a user understanding or debugging their operators. For instance, a dot-product test function, to check for operator adjointness (Claerbout, 2008), and power iteration method function, to estimate maximum and minimum eigenvalues, are implemented and can be applied to any derived operator object. Moreover, a non-linear operator class is defined in the module, which enables the combination of a non-linear mapping function with its Jacobian matrix with respect to the input vector parameters (see the class *NonLinearOperator*). This class is extremely important since it can simplify the syntax when dealing with non-linear inverse problems.

Finally, given the level of abstraction of the vector and operator classes, many useful combinations of operators are already defined within the Python module. For example, chain or stack of linear operators (i.e., $\mathbf{C}\mathbf{x} = \mathbf{B}\mathbf{A}\mathbf{x}$ and $\mathbf{D}\mathbf{x} = [\mathbf{B}; \mathbf{A}]\mathbf{x}$), as well as composition of non-linear operators along with the corresponding Jacobian matrix (i.e., $\mathbf{h}(\mathbf{x}) = \mathbf{f}(\mathbf{g}(\mathbf{x}))$ and $\mathbf{H}(\mathbf{x}_0) = \mathbf{F}(\mathbf{g}(\mathbf{x}_0))\mathbf{G}(\mathbf{x}_0)$).

Problem class

The problem class represents the objective function that we want to minimize using a given inversion algorithm. The module containing its definition is within the file *pyProblem.py*.

Since the library is envisioned to solve geophysical inverse problems, we decide to use the following general objective function definition: $\phi(\mathbf{r}(\mathbf{m}))$, where the objective function ϕ is expressed in terms of a residual vector $\mathbf{r}$, which in turn depends on the model vector $\mathbf{m}$. This design choice allows the user to implement objective functions in which different definitions of the residual vector is used (e.g., zero-lag cross-correlation between observed and predicted data). Therefore, to define a derived child from the abstract problem class the user has to implement at least the functions *objf*, *resf*, and *gradf*, which compute objective function value, residual vector, and gradient vector, respectively. An example of this case can be found in the file *Utest_Rosenbrock.py* within the unit_tests folder.

Some of the common inverse problems encountered in the geophysical context are already implemented. The following problem classes have been already present within the pyProblem module:

- Linear least-squares inversion:

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

  where $\mathbf{L}$ is a linear operator and $\mathbf{d}$ is the data vector.

- Linear symmetric-system inversion:

$$\phi(\mathbf{m}) = \frac{1}{2}\mathbf{m}^*\mathbf{A}\mathbf{m} - \mathbf{m}^*\mathbf{b}, \tag{2}$$

  where $\mathbf{A}$ is a positive definite matrix and $\mathbf{b}$ is a known term. This objective function is useful to solve image-space linearized waveform inversion for instance. Its stationary point corresponds to exactly solve $\mathbf{A}\mathbf{m} = \mathbf{b}$. The residual vector in this case is defined by: $\mathbf{r}(\mathbf{m}) = \mathbf{A}\mathbf{m} - \mathbf{b}$.

- Linear regularized least-squares inversion:

$$\phi(\mathbf{r}(\mathbf{m})) = \frac{1}{2}\|\mathbf{L}\mathbf{m} - \mathbf{d}\|_2^2 + \frac{\epsilon^2}{2}\|\mathbf{D}\mathbf{m} - \mathbf{d}'\|_2^2, \tag{3}$$

  where $\mathbf{D}$ is a regularization operator, ϵ is a scalar regularization weight, and $\mathbf{d}'$ represents a prior data/model regularization vector. Neither $\mathbf{D}$ or $\mathbf{d}'$ is required when instantiating this problem object. In fact, by default the regularization operator is assumed to be identity and the prior vector to be absent (i.e., $\phi(\mathbf{m}) = \frac{1}{2}\|\mathbf{L}\mathbf{m}-\mathbf{d}\|_2^2+\frac{\epsilon^2}{2}\|\mathbf{m}\|_2^2$). The residual vector for this problem defined as a stack of vectors such as $\mathbf{r}(\mathbf{m}) = [\mathbf{L}\mathbf{m}-\mathbf{d}; \epsilon(\mathbf{D}\mathbf{m}-\mathbf{d}')]$. Additionally, an *estimate_epsilon* function is implemented and provides an estimate of the regularization weight that balances the magnitude of the two terms present in this objective function.

- Linear L1-regularized least-squares inversion:

$$\phi(\mathbf{r}(\mathbf{m})) = \frac{1}{2}\|\mathbf{L}\mathbf{m} - \mathbf{d}\|_2^2 + \lambda\|\mathbf{m}\|_1, \tag{4}$$

  λ is the regularization weight and the residual vector has a similar definition as in equation 3. This problem is commonly known as the least absolute shrinkage and selection operator (LASSO) problem (Santosa and Symes, 1986; Tibshirani, 1996).

- Non-linear L2 inversion:

$$\phi(\mathbf{r}(\mathbf{m})) = \frac{1}{2}\|\mathbf{r}(\mathbf{m})\|_2^2 = \frac{1}{2}\|\mathbf{f}(\mathbf{m}) - \mathbf{d}\|_2^2, \tag{5}$$

  where $\mathbf{f}$ represents the non-linear operator that maps the model vector into the data space.

- Non-linear regularized L2 inversion:

$$\phi(\mathbf{r}(\mathbf{m})) = \frac{1}{2}\|\mathbf{f}(\mathbf{m}) - \mathbf{d}\|_2^2 + \frac{\epsilon^2}{2}\|\mathbf{g}(\mathbf{m}) - \mathbf{d}'\|_2^2, \tag{6}$$

  where $\mathbf{g}$ represents a linear or non-linear regularization operator. This problem class has a similar structure as the one of equation 3.

- Non-linear L2 inversion solved through the variable-projection (VP) algorithm (Golub and Pereyra, 1973):

$$\phi(\mathbf{r}(\mathbf{m})) = \frac{1}{2}\|\mathbf{f}(\mathbf{m}) + \mathbf{h}(\mathbf{m})\mathbf{m}_L^* - \mathbf{d}\|_2^2 + \frac{\epsilon^2}{2}\|\mathbf{g}(\mathbf{m}) + \mathbf{h}'(\mathbf{m})\mathbf{m}_L^* - \mathbf{d}'\|_2^2, \tag{7}$$

  where $\mathbf{h}$ and $\mathbf{h}'$ are non-linear functions with respect to $\mathbf{m}$ and linear with respect to $\mathbf{m}_L^*$. For a given $\mathbf{m}$, the variable $\mathbf{m}_L^*$ is found iteratively by minimizing the following objective function:

$$\phi_{VP}(\mathbf{r}(\mathbf{m}_L)) = \frac{1}{2}\|\mathbf{h}(\mathbf{m})\mathbf{m}_L - [\mathbf{d} - \mathbf{f}(\mathbf{m})]\|_2^2 + \frac{\epsilon^2}{2}\|\mathbf{h}'(\mathbf{m})\mathbf{m}_L - [\mathbf{d}' - \mathbf{g}(\mathbf{m})]\|_2^2, \tag{8}$$

  which corresponds to the problem of equation 3.
  Many geophysical inverse problems can be cast in this form and efficiently solved using the VP approach (Rickett, 2013; Huang and Symes, 2015; Barnier et al., 2018). The implementation of this inverse problem can be found in the file *pyVPproblem.py*.

Solver class

The solver class is the driver of the optimization and effectively minimizes (or solves) any proposed problem consistent to its purpose. Meaning a linear conjugate-gradient

algorithm will not be able to solve a non-linear least-square objective function. The solver abstract implementation can be found in the file *pySolver.py*. The function *setDefaults* controls the saving of inversion-related vectors while the solver is minimizing a provided problem object. Otherwise, once the solver returns control to the calling program section, the current optimal model vector can be retrieved either from the solver or the problem objects.

To guide the user in the choice of the solver to employ, here below we list the currently implemented solvers and the supported objective functions that can minimize:

- Linear conjugate-gradient algorithm (LCG) supports the objective functions of equations 1 and 3 (Aster et al., 2005; Claerbout, 2014). Implemented in the module within the file *pyLCGsolver.py*.
- Linear conjugate-gradient algorithm for symmetric system supports the objective function of equation 2 (Gill et al., 1981). Implemented in the module within the file *pySymLCGsolver.py*.
- Iterative shrinkage-thresholding (ISTA) and the Fast ISTA (FISTA) algorithms support the objective function of equation 4 (Beck and Teboulle, 2009). Implemented in the module within the file *pyISTAsolver.py*. This solver requires an estimate of the maximum eigenvalue of the operator $\mathbf{L}$.
- Iterative shrinkage-thresholding with cooling algorithm (ISTc) support the objective function of equation 4 (Hennenfent et al., 2008). Implemented in the module within the file *pyISTCsolver.py*. This solver requires an estimate of the maximum eigenvalue of the operator $\mathbf{L}$ but automatically computes the value of λ.
- Non-linear conjugate-gradient algorithm (NLCG) supports the objective functions of equations 1, 3, 5, 6, and 7 (Nocedal and Wright, 2006; Hager and Zhang, 2006). Implemented in the module within the file *pyNLCGsolver.py*.
- Limited-memory Broyden-Fletcher-Goldfarb-Shanno algorithm (L-BFGS) supports the objective functions of equations 1, 3, 5, 6, and 7 (Liu and Nocedal, 1989). Implemented in the module within the file *pyLBFGSsolver.py*.

All these solver classes can handle convex box bounds of the following form: $\mathbf{m}_{min} \leq \mathbf{m} \leq \mathbf{m}_{max}$, where $\mathbf{m}_{min}$ and $\mathbf{m}_{max}$ are the minimum and maximum vector values, respectively, and the inequalities are assumed to be element wise. In addition, the solvers also accept projection operators such that different constraints can be enforced (e.g., linear constraints).

Furthermore, when a non-linear problem has to be solved (e.g., the objective functions of equations 5, 6, and 7), a line search algorithm has to be applied. The library treats that inversion step using a separate class, providing implementational freedom and separation from the solver employed to minimize an objective function. Currently, the following three stepping algorithms are available:

- Three-point parabolic interpolation. This *Stepper* class uses a three-point interpolation to find a step-length value corresponding to the minimum of the fitted parabola and chooses the minimum among the three tested points. Implemented in the file *pyStepperParabolic.py*. This method is the default stepper for the NLCG method.

- Constant-curvature stepping algorithm. In this approach a single testing point is employed to find the minimum of parabolic approximation in which the curvature term is assumed to be constant (Nocedal and Wright, 2006). Its implementation can be found in the file *pyStepperParabolic.py*.

- Uncertainty interval stepper with guaranteed sufficient decrease. This stepping algorithm verifies if the strong Wolfe conditions are met by the tested point. If they do not hold, it updates an interval of uncertainty and test a new point until those conditions are verified (Moré and Thuente, 1994). Implemented in the file *pyStepperCvSrch.py*. This method is the default stepper for the L-BFGS algorithm.

Finally, to test convergence to the optimal solution the solver uses an additional class called *stopper*. This class has to be provided when a solver object is instantiated. All the supported convergence criteria are implemented within the class of the module in the file *pyStoppperBase.py*.

LIBRARY USAGE

In this section we provide a usage example of how the previously described classes are combined so that an inverse problem is solved. The steps necessary for solving an inverse problems can be summarized by the following pseudo code:

Algorithm 1 How to solve an inverse problem

1) Instantiate initial model vector $\mathbf{m}_0$
2) Instantiate problem object $\phi(\mathbf{m})$ using $\mathbf{m}_0$ as starting point
3) Instantiate solver object along with stopping criteria
4) Pass problem object to the solver to obtain $\mathbf{m}^* = \underset{\mathbf{m}}{\operatorname{argmin}}\ \phi(\mathbf{m})$

Here below we report a Python script to solve a linear least-squares inverse problem (equation 1):

```
#Example of minimizing a linear least-squares inverse problem
import numpy as np
#Library related modules
import pyVector as Vec
from Utest_LCGsolver import MatMult_incore
import pyProblem as Prblm
import pyStopperBase as Stopper
```

```
8  import pyLCGsolver as LCG
9
10 #Instantiate model vector
11 n=200
12 model = Vec.vectorIC(np.zeros((n,1))) # m0
13 model.zero() # m0 = 0
14 #Instantiate data vector
15 data = Vec.vectorIC(np.zeros((n,1))) # d
16 data.set(1.) # d = 1
17 #Instantiate operator object from a symmetric matrix
18 #Second-order derivative operator
19 L = np.matrix(np.zeros((n,n)))
20 np.fill_diagonal(L, -2)
21 np.fill_diagonal(L[1:], 1)
22 np.fill_diagonal(L[:,1:], 1)
23 D2 = MatMult_incore(L,model,data) # L (domain = m; range = d)
24 #Create L2-norm linear problem
25 # phi(m) = 1/2*|Lm - d|_2^2
26 L2Prob = Prblm.ProblemL2Linear(model,data,D2)
27 #Instantiate stopper and solver object
28 StopObj  = Stopper.BasicStopper(niter=2000)
29 LCGsolver = LCG.LCGsolver(StopObj)
30 #Solving the problem
31 LCGsolver.run(L2Prob,verbose=True)
32 #Solution vector is in L2Prob.model
```

From this script we can see that the lines 12, 26, 29, and 31 respectively correspond to the four steps described in the pseudo code above. It is important to note that even if we use a different definition of the vector and operator objects, such as objects that employ an out-of-core operations, the lines from 26 to 31 would not change. Effectively, we use the same code to solve the defined objective function. Furthermore, the code structure and logic, in general, for a linear inverse problem would be very close to the reported script.

INVERSION EXAMPLES

In this section we report various examples in which the proposed library is applied to different inverse problems. Given the level of abstraction of the library, we are able to easily combine different programming languages together within the implemented Python interface. Here, we report some of these examples. Moreover, a few simple matrix inversions can be found in the unit_tests folder within the file Utest_LCGsolver.py.

Linearized waveform inversion

In this example we solve the problem of equation 1, where now **L** represents the Born scattering operator (Barnier and Almomin, 2014), **d** is Born data generated using the

true reflectivity model of Figure 1a, and $\mathbf{m}$ is the unknown reflectivity model that we want to find. For solving this inverse problem, we employ a library containing acoustic wave-equation operators implemented using *CUDA* that takes advantage of the computational acceleration given by graphic processing units (GPUs) (Nickolls et al., 2008). The vector class interfacing these operators is based on the genericIO library written in C++, which takes advantage of the Intel Threading Building Blocks system for performing vector operation (Blumofe et al., 1996). Therefore, by using Pybind11 we are able to use the inversion package in connection with these two libraries written in two different programming languages.

The inverted model after 2000 iterations of the LCG method is shown in Figure 1b. We notice a very good agreement with the true reflectivity model almost in every portion of the subsurface. The mismatch is due to the operator null space and the truncation of the inversion process. In fact, by analyzing the logarithm of the scaled objective function (Figure 2a), we can see that the iterative method has almost attained the numerical precision for single-precision floating-point numbers (i.e., $\approx 10^{-6}$), but has not reached it. In addition, when we compute the model percentage matched as function of iterations (Figure 1b), defined as $100(1 - \|\mathbf{m} - \mathbf{m}_{true}\|_2/\|\mathbf{m}_{true}\|_2)$, where $\mathbf{m}_{true}$ represents the true reflectivity model, we see that approximately 80% of the true reflectivity is retrieved at the end of the iterative process. The apparent asymptote might be connected the operator null space, since a finite-bandwidth wavelet (5-20 Hz) and a limited surface acquisition (170 sources and 1700 receivers) are employed in this numerical experiment.

Image deblurring using sparse constraint by LASSO

In this experiment the goal is to test the solvers related to the LASSO problem (equation 4). One interesting feature in this example is the fact that we utilize optimized blurring Gaussian filters implemented within the *Scipy* library (Jones et al., 2001). In this case, we compare the inverted model parameters using equations 1 and 4 in which $\mathbf{L}$ is a bi-dimensional Gaussian blurring filter, $\mathbf{d}$ is the blurred known image, and $\mathbf{m}$ represents the unknown deblurred image. Figure 3a displays the known blurred image of two delta functions or spikes. The deblurred image after applying the LCG solver to minimize equation 1, until numerical convergence, is shown in Figure 3b. As expected, the image of the two spikes cannot be fully retrieved due to operator null space and numerical inaccuracy (Lam and Goodman, 2000). On the other hand, when minimizing equation 4, employing the ISTc algorithm, we can retrieve a satisfactory deblurred image of the two spikes (Figure 3c). Similar results can be achieved in this last case when the ISTA or FISTA solvers are employed.

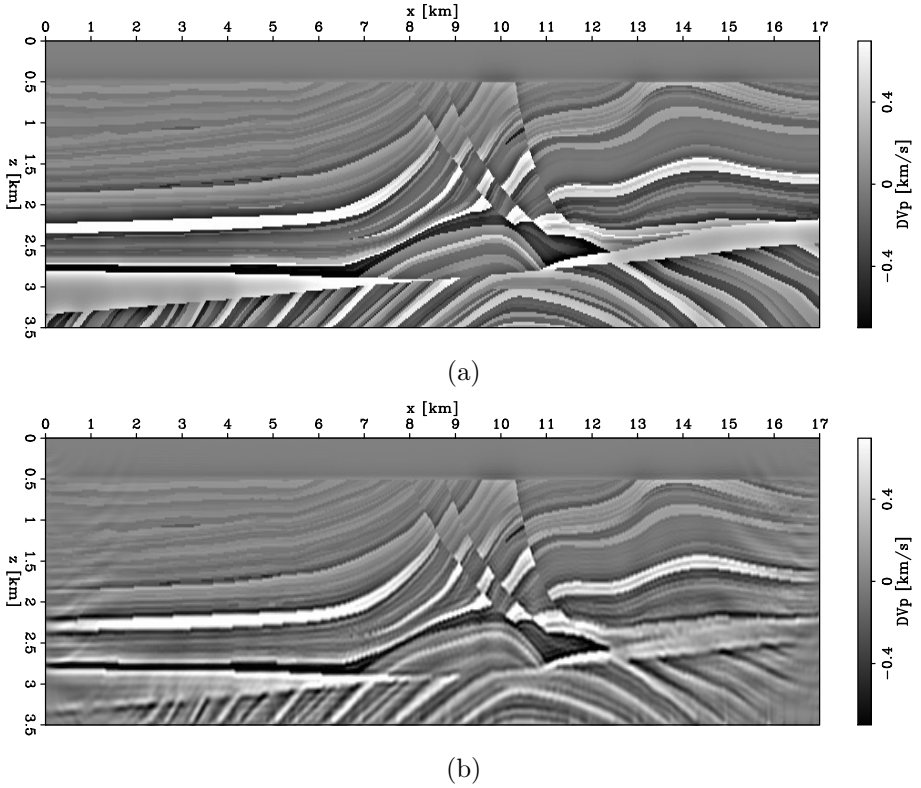


Figure 1: True (a) and inverted (b) reflectivity models. The inversion is performed by applying 2000 iterations of the LCG solver. [**CR**]

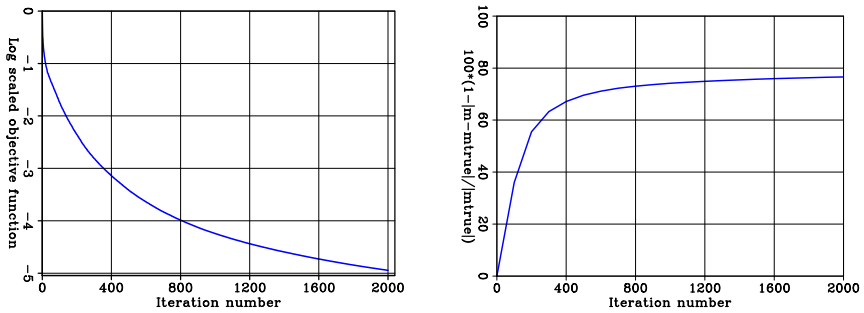
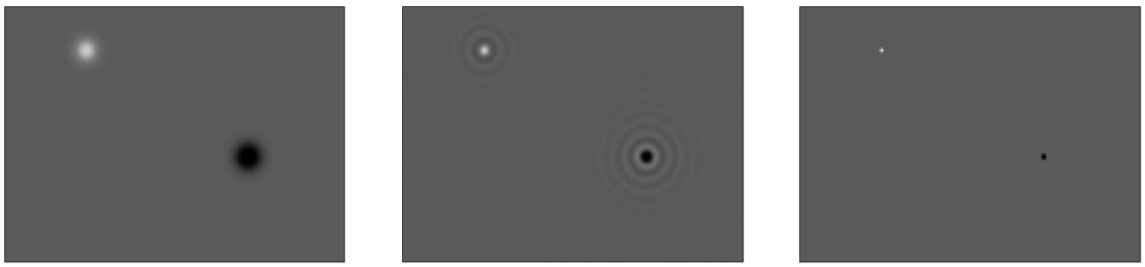


Figure 2: (a) Log of the scaled objective function versus the number of iterations. (b) Percentage of the true reflectivity model retrieved as function of the iteration number. [**CR**]



(a) (b) (c)

Figure 3: (a) Blurred image of two delta functions. Deblurred images by minimizing equation 1 using the LCG algorithm (b) and equation 4 using the ISTc solver (c). [**ER**]

Non-linear optimization: the Rosenbrock function

Before applying any inversion library, it is good practice to test any implemented optimization method on analytic objective functions for which the global or local minima are known. For this purpose, a good benchmark is represented by the Rosenbrock function (Rosenbrock, 1960). This function has the following expression:

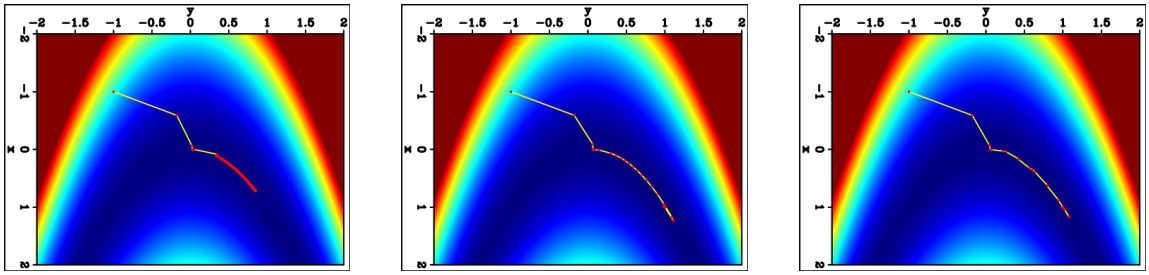
$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2, \tag{9}$$

where x and y are the two unknown parameters and the unique global minimum for $x = y = 1$ such that $f(1, 1) = 0$. To test the implemented algorithms of the library we minimize equation 9 starting from the initial point $x = y = -1$. Figure 4 shows the optimization paths for three different optimization algorithms: non-linear steppest-descent (NLSD), NLCG, and BFGS. For all the three methods the same parabolic stepper is applied. The NLSD method fails to reach the minimum after 500 iterations (maximum number of iterations used). On the contrary, both NLCG and BFGS algorithms reach the function minimum after 206 and 26 iterations, respectively. As expected, the BFGS solver is superior than the others since it builds an approximated Hessian inverse as the inversion progresses (Nocedal and Wright, 2006).

Non-linear optimization: FWI

Now that we know that the algorithms can find the minimum of an analytic function, we employ them on a well-known geophysical inverse problem. In this case, the goal is to minimize the objective function in equation 5, where $\mathbf{f}$ represents the non-linear acoustic wave-equation operator (Barnier and Almomin, 2014), $\mathbf{d}$ the recorded pressure for each shot recorded by receivers at the surface, and $\mathbf{m}$ the unknown velocity model. In this example, we employ the same wave-equation library employed in the linearized waveform inversion case.

In this experiment we employ a portion of the Marmousi model (Martin et al., 2006) (Figure 5a). We generate acoustic pressure data with a wavelet with energy



(a) (b) (c)

Figure 4: Convergence comparison for non-linear optimization algorithm starting from the same initial point. Optimization paths for NLSD (a), NLCG (b), and BFGS (c) algorithms. The red dots represent the sampled points at each iteration. Red and blue colors represent high and low values of the function, respectively. [**ER**]

between 0.5 and 15 Hz using 100 sources and 500 receivers spaced at the surface by 100 m and 20 m, respectively. Figure 5b shows the inaccurate starting velocity model that corresponds to a laterally invariant velocity model.

Because of the inaccurate starting model, we apply a data-space multi-scale approach (Bunks et al., 1995). We sequentially inverted nine frequency bands; namely, 0-3, 0-4, 0-5, 0-6, 0-8, 0-10, 0-12, 0-15 Hz. We apply 200 iterations of a non-linear minimization algorithm per band. For the first band, we compare the performance of NLCG and BFGS methods in terms of objective function value (Figure 6). We clearly observe the superiority of the BFGS algorithm right after approximately 10 iterations in this case. Additionally, by comparing the inverted models by the two approaches (Figure 7), we notice that, for the same number of iterations, the velocity model retrieved by the BFGS method presents more features for the same frequency content. This observation is particularly noticeable in the deeper section of the subsurface. Again, this result is due to the ability of the BFGS algorithm of finding an approximation to the objective-function Hessian inverse, which, in this case, corresponds to a wavelet deconvolution and an illumination compensation.

The final retrieved model after the inversion of all the frequency bands is displayed in Figure 8. We can observe a good agreement with the true model, meaning the inversion is able to invert an accurate velocity model starting from an incorrect laterally invariant initial guess.

CONCLUSIONS

We describe an object-oriented optimization library that provides us with a flexible framework in which different objective functions and algorithms can be easily applied to both small- and large-scale inverse problems. The capability of applying it to small-scale problems allows us quick testing before switching to a realistic large-scale case. We provide a general description of the library structure and of its components

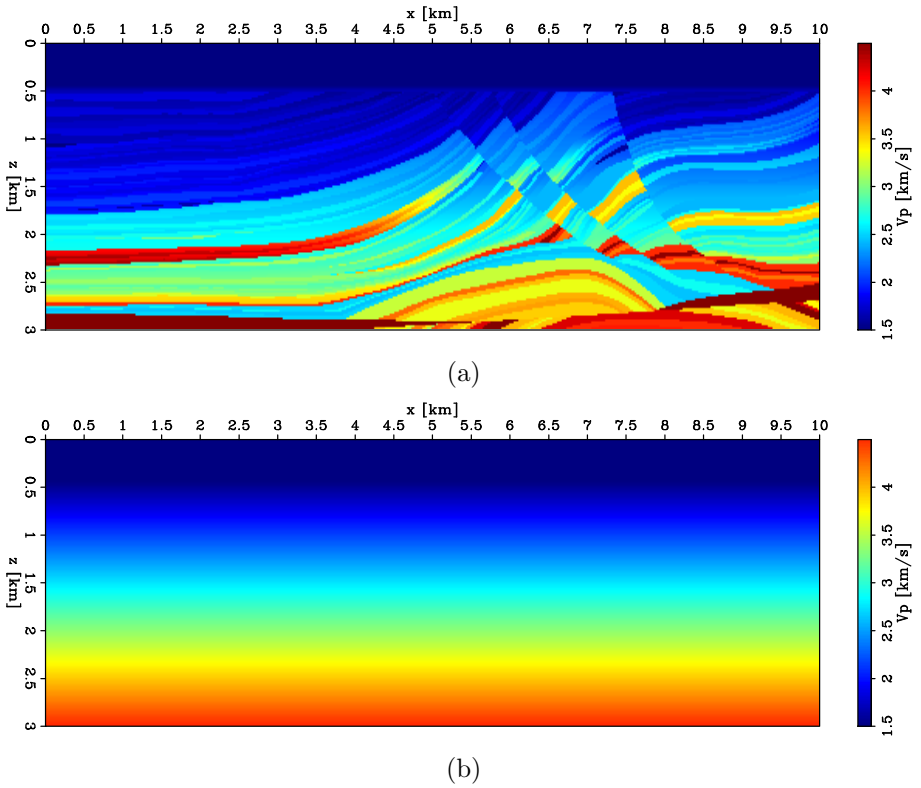


Figure 5: (a) True Marmousi model. (b) Initial v(z) model used in the FWI experiment. [**ER**]

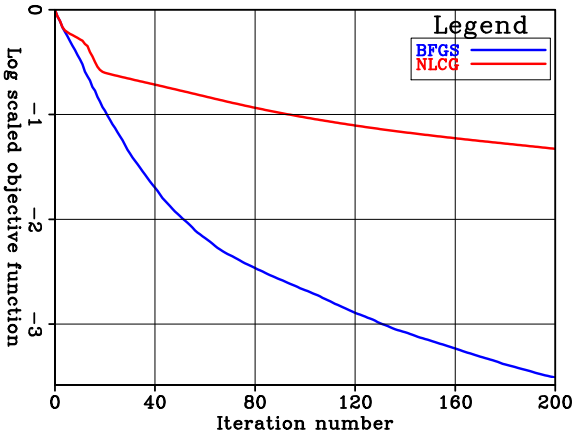


Figure 6: Comparison between convergence curves of NLCG (red curve) and BFGS (blue curve) methods. [**ER**]

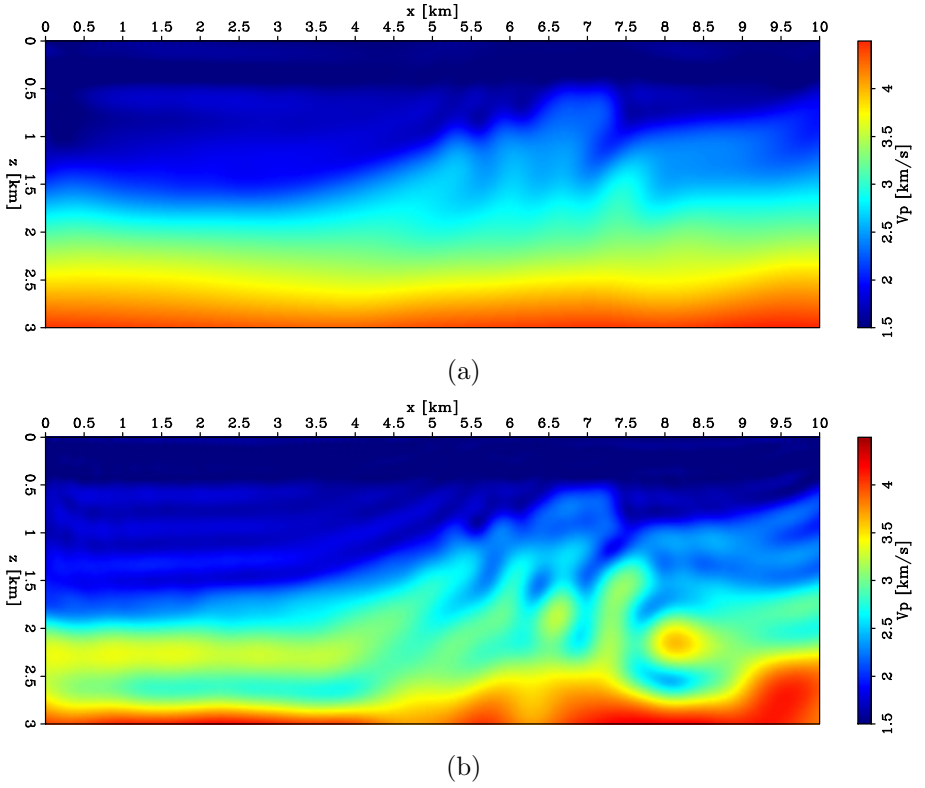


Figure 7: First-band (0-3Hz) FWI inverted models by NLCG (a) and BFGS (b) algorithms. [**ER**]

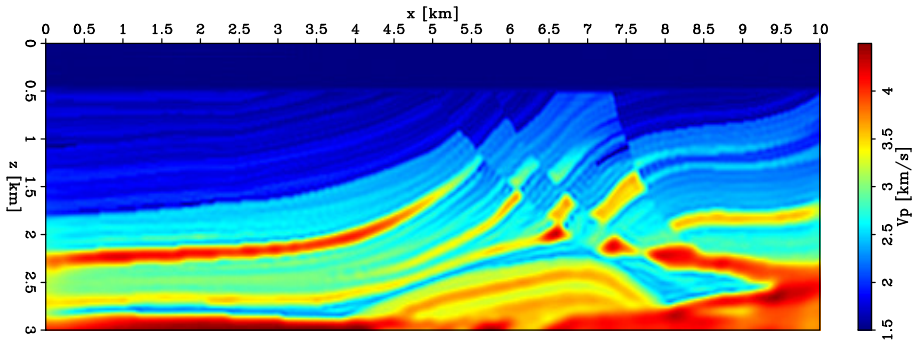


Figure 8: Final inverted velocity model after all the frequency bands (0-3, 0-4, 0-5, 0-6, 0-8, 0-10, 0-12, 0-15 Hz) are inverted using 200 iterations of the BFGS solver per band. [**CR**]

as well as a simple Python script example to solve a linear system of equations.

We report various linear and non-linear minimization examples that demonstrate the ability of the library of working with different packages, implemented in different languages, and being able to deal with small- and large-scale inversion problems in a painless fashion.

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.

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

Barnier, G., and A. Almomin, 2014, Tutorial on two-way wave equation operators for acoustic, isotropic, constant-density media.

Barnier, G., E. Biondi, and B. Biondi, 2018, Full waveform inversion by model extension, *in* SEG Technical Program Expanded Abstracts 2018: Society of Exploration Geophysicists, 1183–1187.

Beck, A., and M. Teboulle, 2009, A fast iterative shrinkage-thresholding algorithm for linear inverse problems: SIAM journal on imaging sciences, **2**, 183–202.

Biondi, E., and G. Barnier, 2017, A flexible out-of-core solver for linear/non-linear problems.

Blumofe, R. D., C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, 1996, Cilk: An efficient multithreaded runtime system: Journal of parallel and distributed computing, **37**, 55–69.

Bunks, C., F. M. Saleck, S. Zaleski, and G. Chavent, 1995, Multiscale seismic waveform inversion: Geophysics, **60**, 1457–1473.

Byrd, R. H., P. Lu, J. Nocedal, and C. Zhu, 1995, A limited memory algorithm for bound constrained optimization: SIAM Journal on Scientific Computing, **16**, 1190–1208.

Claerbout, J., 2014, Geophysical image estimation by example: Citeseer.

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

———, 2008, Basic earth imaging.

Clapp, R. G., 2017, Facilitating code distribution: Docker and Generic IO.

Farris, S., and B. Biondi, 2018, Velocity estimation with alternating gradiometry and wavefield reconstruction inversion.

Gill, P. E., W. Murray, and M. H. Wright, 1981, Practical optimization: Academic press.

Golub, G. H., and V. Pereyra, 1973, The differentiation of pseudo-inverses and nonlinear least squares problems whose variables separate: SIAM Journal on numerical analysis, **10**, 413–432.

Grant, M., and S. Boyd, 2014, CVX: Matlab software for disciplined convex programming, version 2.1: `http://cvxr.com/cvx`.

Hager, W. W., and H. Zhang, 2006, A survey of nonlinear conjugate gradient methods: Pacific journal of Optimization, **2**, 35–58.

Hennenfent, G., E. v. d. Berg, M. P. Friedlander, and F. J. Herrmann, 2008, New insights into one-norm solvers from the pareto curve: Geophysics, **73**, A23–A26.

Huang, Y., and W. W. Symes, 2015, Born waveform inversion via variable projection and shot record model extension, *in* SEG Technical Program Expanded Abstracts 2015: Society of Exploration Geophysicists, 1326–1331.

Jakob, W., J. Rhinelander, and D. Moldovan, 2016, pybind11 seamless operability between c++11 and python. (https://github.com/pybind/pybind11).

Jones, E., T. Oliphant, P. Peterson, et al., 2001, SciPy: Open source scientific tools for Python.

Lam, E. Y., and J. W. Goodman, 2000, Iterative statistical approach to blind image deconvolution: JOSA A, **17**, 1177–1184.

Liu, D. C., and J. Nocedal, 1989, On the limited memory bfgs method for large scale optimization: Mathematical programming, **45**, 503–528.

Martin, G. S., R. Wiley, and K. J. Marfurt, 2006, Marmousi2: An elastic upgrade for marmousi: The Leading Edge, **25**, 156–166.

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

Moré, J. J., and D. J. Thuente, 1994, Line search algorithms with guaranteed sufficient decrease: ACM Transactions on Mathematical Software (TOMS), **20**, 286–307.

Nickolls, J., I. Buck, and M. Garland, 2008, Scalable parallel programming: 2008 IEEE Hot Chips 20 Symposium (HCS), IEEE, 40–53.

Nocedal, J., and S. Wright, 2006, Numerical optimization: Springer Science & Business Media.

Rickett, J., 2013, The variable projection method for waveform inversion with an unknown source function: Geophysical Prospecting, **61**, 874–881.

Rosenbrock, H., 1960, An automatic method for finding the greatest or least value of a function: The Computer Journal, **3**, 175–184.

Santosa, F., and W. W. Symes, 1986, Linear inversion of band-limited reflection seismograms: SIAM Journal on Scientific and Statistical Computing, **7**, 1307–1330.

Tibshirani, R., 1996, Regression shrinkage and selection via the lasso: Journal of the Royal Statistical Society: Series B (Methodological), **58**, 267–288.

Van Der Walt, S., S. C. Colbert, and G. Varoquaux, 2011, The numpy array: a structure for efficient numerical computation: Computing in Science & Engineering, **13**, 22.