C++11 non-linear solver

Robert G. Clapp, Stuart Farris, Taylor Dahlke, and Eileen Martin

ABSTRACT

Inverse problems such as velocity estimation from reflection/refraction data are inherently non-linear. We developed a library to address non-linear problems using C++11. We demonstrate the library on two simple examples.

INTRODUCTION

The last twenty years has seen an increasing percentage of SEP theses concerned with solving large inverse problems. During that period a large number of inversion libraries have been developed. Nichols et al. (1993) developed a framework using an early version of C++. For years many small inversions problems were handled by the Fortran90 library developed by Fomel and Claerbout (1996). Schwab and Schroeder (1997) developed a library using Java. For out of core and multi-node applications the python library described in Clapp (2005) was used. A second attempt at a C++ optimization library was described in Martin et al. (2014).

All of the above attempts concentrated on solving linear problems. The summer of 2015 saw the first significant attempt at building a framework for non-linear problems with a Fortran 2003 library described in Almomin et al. (2015). The library described in Almomin et al. (2015), while well designed, is limited to in-core problems and only acccesible to Fortran programers. Biondi and Barnier (2017) addresses the first issue by emulating the design of Almomin et al. (2015) but written in python for out-of-core problems.

In this paper we descibe a non-linear version library written in C++11 using the same design as the Fortran library descibed in Almomin et al. (2015). We begin by reviewing the design principals of the library. We then present two simple inversion examples using the library. Finally, we discuss planned future additions to the library.

THE DOCKER ENVIRONMENT

One unique aspect of this project is the integration of the library that we've built with a portable environment system called Docker. Docker is a recent software product that holds an advantage over other environment replicating systems, and holds special promise for integrating with future SEP work in terms of reproducibility. One of the leading challenges for software reproducibility and deployment has been the difficulty in replicating the working environment. Virtual machines are a common way to circumvent this problem, but these machines can take minutes to launch, versus seconds (or less) with most Docker containers.

Creating a docker container for each SEP report would allow the reader to launch the same environment that the author used. Further, since most all SEP research is run on linux-based systems, most containers would share the same parts of their container images. Docker is unique in that when multiple (similar) containers are run, the bulk of resources can be shared between them since they operate from the same base image. This makes the system light, and easier to deploy. For these reasons, we employ Docker in this report to showcase how the product can be used to enhance the reproducibility of SEP research.

One aspect of using Docker containers to replicate run environments is that we can easily build the environment that we want to use by means of a Dockerfile. An example of such a file shows how one can be used to setup the solver library that is used for the examples in this report, and would be used by you, the reader, to replicate the results you see here.

```
From rgc007/geelab:2017
MAINTAINER Bob Clapp <bob@sep.stanford.edu>
RUN yum -y install xorg-x11-server-Xvfb whichcmake boost-develyum clean all
RUN dbus-uuidgen >/etc/machine-id
RUN git clone http://zapad.Stanford.EDU/SEP-external/gieeSolver.git /opt/gieeSolver
RUN git clone http://zapad.Stanford.EDU/bob/genericIO.git /opt/genericIO
RUN mkdir -p /opt/gieeSolver/build
RUN mkdir -p /opt/genericIO/build
RUN cd /opt/genericIO/build
RUN cmake -DCMAKE_INSTALL_PREFIX=/opt/genericIO -DSEPlib_DIR=/opt/SEP/lib ..
RUN make install
RUN cd /opt/gieeSolver/build
RUN cmake -DCMAKE_INSTALL_PREFIX=/opt/gieeSolver -DgenericIO_DIR=/opt/genericIO/lib -DBoost_INCLUDE_DIR=/usr/include ..
RUN make install
```

Listing 1: Sample Dockerfile

In 1, we first call the base image that we wish to build or install on. Afterwards, we can use the RUN operator to execute a series of commands that install programs and utilities on top of the starting image that we designate. In this case, we clone the libraries that are used for the examples in this report, and then install those libraries into the environment that we've specified. Once this container is built, we can save it as an image for later users to run without having to execute the script that holds these build commands. This interface makes creating an distributing environments (such as the one included with this report) a relatively simple task.

SOLVERS

This library was designed to have interfaces similar to the of Biondi and Barnier (2017). This means that we work with solver objects defined by a `problem`, a `stepper`, and a `terminator`. A `problem` object is associated with an objective we wish to minimize, and has methods associated with getting that objecting value for a particular model, getting gradients, data misfits/residuals, and managing the domain and range of a problem. A `stepper` decides for a given problem how far to move from the current model along a certain update path (think of any line search). A `terminator` has a `test()` method that returns a boolean indicating whether to continue iterating when solving an inverse problem.

The solver class, called `nlsolver`, sets up a method to solve a given `problem`, using a particular `stepper`, and a specified `terminator`. All solvers have a method called `run()` that actually solves the problem using the stepper and terminator. But the solver class is abstract, so specific types of solver algorithms must be implemented. Many types of both linear and nonlinear solvers can be implemented in this framework. Two examples of solvers we have implemented are linear conjugate gradient, and nonlinear conjugate gradient. The linear conjugate gradient solver class, called `linSolverCG`, is one of the simpler optimization solver algorithms that can be implemented as a special case of a `nlsolver` because it only requires a `problem` and `terminator`, but does not need a `stepper` because the step length is predetermined in the usual way.

The nonlinear conjugate gradient solver class, called `nonlinSolverCG`, is a type of `nlsolver` that requires pointers to a `problem`, a `stepper`, a `terminator`, as well as a string indicating the method for calculating the CG update, β. As in Biondi and Barnier (2017), the user is only expected to interact with a `nonlinSolverCG` object through two methods: instantiation, and `run()`. Once `run()` is called, everything else happens behind the scenes based on what `problem`, `stepper`, `terminator` and β update were specified at the instantiation.

The steps happening behind the scenes when `run()` is called are:

1. Use `betaAssigner()` method to figure out which β update to use
2. Calculate g_0, the gradient of the initial model guess, x_0
3. Set $d_0 = -g_0$, the first update direction
4. While `terminator`'s `test()` method says to continue iteration:
    (a) Calculate g_{k+1}, the gradient of objective at the current model x_k

(b) Calculate scalar β_k using the specified β update
(c) Calculate the new update direction $d_{k+1} = -g_{k+1} + \beta_k d_k$
(d) Use the `stepper` to select α_k and update the model $x_{k+1} = x_k + \alpha_k d_k$
(e) $k = k + 1$

As in the library of Biondi and Barnier (2017), the nonlinear CG solver has multiple options for how to calculate β, so when a nonlinear CG solver object is instantiated, the user must provide a `std::string` referred to as its `betaMethod`, and a method called `betaAssigner` ensures the proper beta calculation happens. A user can simply use the method `run` with no additional parameters regardless of how β is calculated.Primarily following Hager and Zhang (2006), currently supported methods for β calculation are:

`betaMethod`	$\beta_k =$
FR	$\|g_{k+1}\|^2/\|g_k\|^2$
PRP	$g_{k+1}^T y_k/\|g_k\|^2$
HS	$g_{k+1}^T y_k/(d_k^T y_k)$
CD	$\|g_{k+1}\|^2/(-d_k^T g_k)$
LS	$g_{k+1}^T y_k/(-d_k^T g_k)$
DY	$\|g_{k+1}\|^2/(d_k^T y_k)$
BAN	$g_{k+1}^T y_k/(g_k^T y_k)$
HZ	$\left(y_k - 2d_k \frac{\|y_k\|^2}{d_k^T y_k}\right)^T \frac{g_{k+1}}{d_k^T y_k}$
ST	$\beta = 0$

Note that ST is simply steepest descent.

RE-IMPLEMENTING GIEE

A secondary goal of the C++ nonlinear solver is to re-implement the linear solvers and operators transcribed in Geophysical Image Estimation by Example (GIEE), (Claerbout, 2014). While the ultimate goal of this solver is to address nonlinear problems, the lessons, examples, and experience in GIEE are invaluable for a budding geophysical image processor. Furthermore, many of the nonlinear problems addressed by the SEP are built around the linear ideas described in GIEE. Therefore, we deem it necessary to add GIEE to the C++ nonlinear solver.

Here we begin to re-implement GIEE by converting all of the in-text examples from Fortran to C++ one chapter at a time. These chapters are found in the solver directory `opt/gieeSolver/giee/`. Each chapter will contain a Makefile that can reproduce all of its figures. For example, Chapter One: Basic Operators and Adjoints illustrates the derivative and convolution operators using a few simple figures. To reproduce these figures, simply enter the Docker environment, as described above, move to the directory associated with chapter one, and run the appropriate make rule. Listing 2 illustrates these steps on the command line. The Figures

1 and 2 will then appear within the figure directory associated with chapter one, `/opt/gieeSolver/giee/ajt/Fig/`.

```
cd /opt/gieeSolver/giee/ajt
make stangrad90.v
make conv.v
```

Listing 2: GIEE Examples

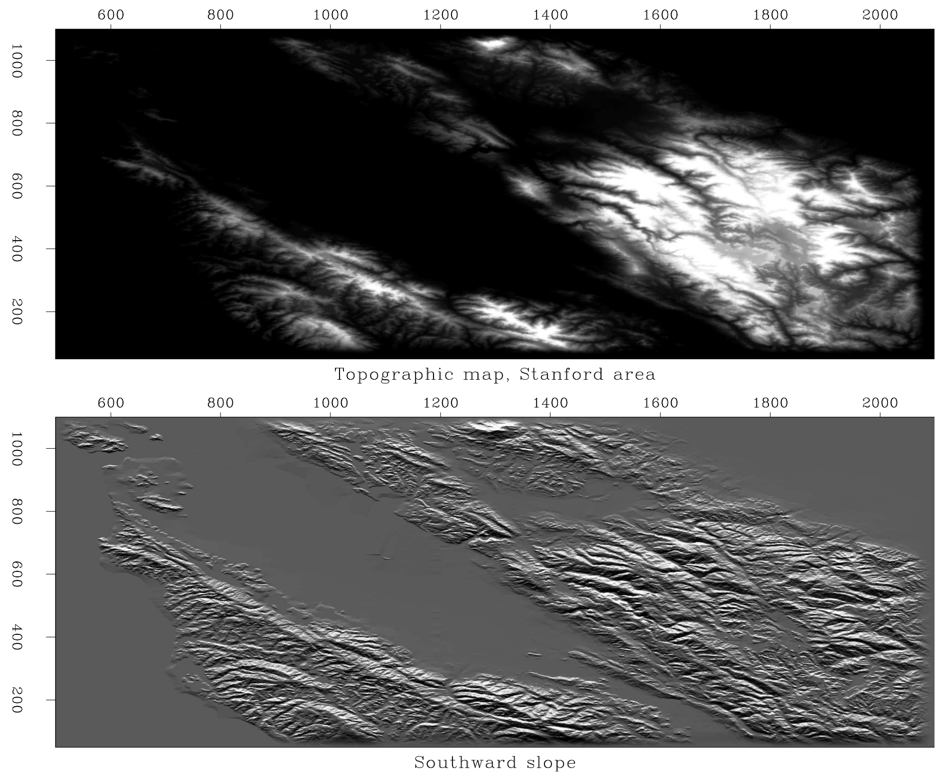

Figure 1: Illustration of derivative operator reimplemented from GIEE. [**ER**]

We plan to re-implement all of the chapters from GIEE using the C++ nonlinear solver framework in ascending order.

INVERSION EXAMPLES

NMO operator

We demonstrate the solver library on a simple normal moveout (NMO) operator, where we invert for the slowness and time position of hyperbolic reflection events. Both examples use conjugate gradient inversion, with one example being regularized.

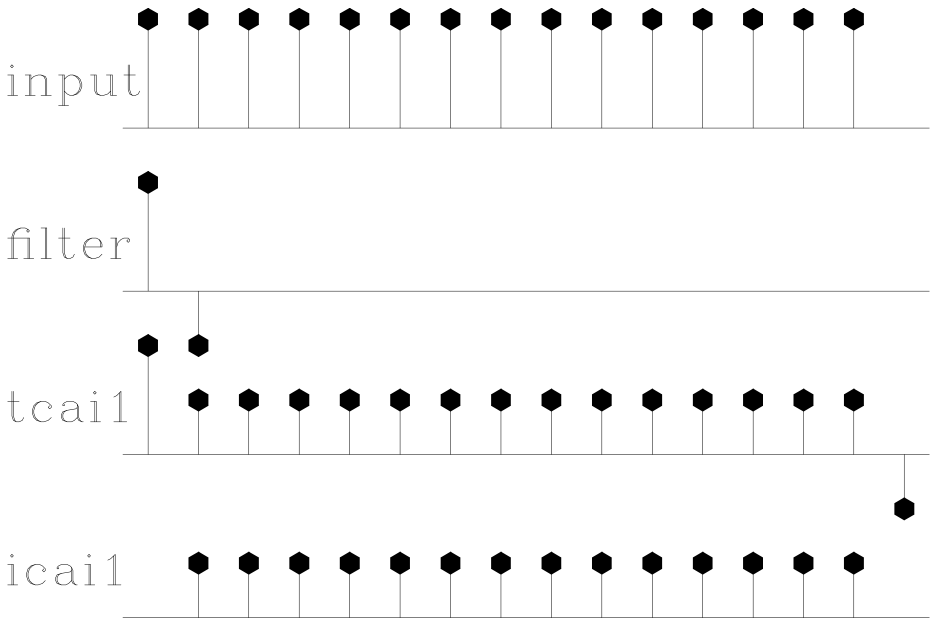

Figure 2: Illustration of transient and internal convolutions reimplemented from GIEE. [**ER**]

Conjugate gradient inversion

We begin with a true model as shown in Figure 3. From this we apply the forward NMO operator to create hyperbolas. We apply a smoothing to these hyperbolas to avoid inversion crime (Figure 4). We then apply 25 iterations of conjugate gradient inversion, begining with an empty (all zeros) initial model. We get results that roughly match the true model in spatial extent as well as amplitude (Figure 5).

Regularized conjugate gradient inversion

For this case, we regularized the model space with a first order LaPlacian smoothing operator. We set the parameter ϵ to 10.0, which controls the strength of the regularization term. Using a higher ϵ value means we will gain a smoother inverted result. Figure 6 shows the result of this regularized inversion. Note that this result is less spiky than the unregularized inversion result in Figure 5.

FUTURE WORK

There are several ways that we want to grow this project. From a library perspective we need to add additional non-linear solvers and line search methods. We also need an expanded library of operators. This project started as way to begin the transition of both the book and class associated with Claerbout (2014) from using Fortran90

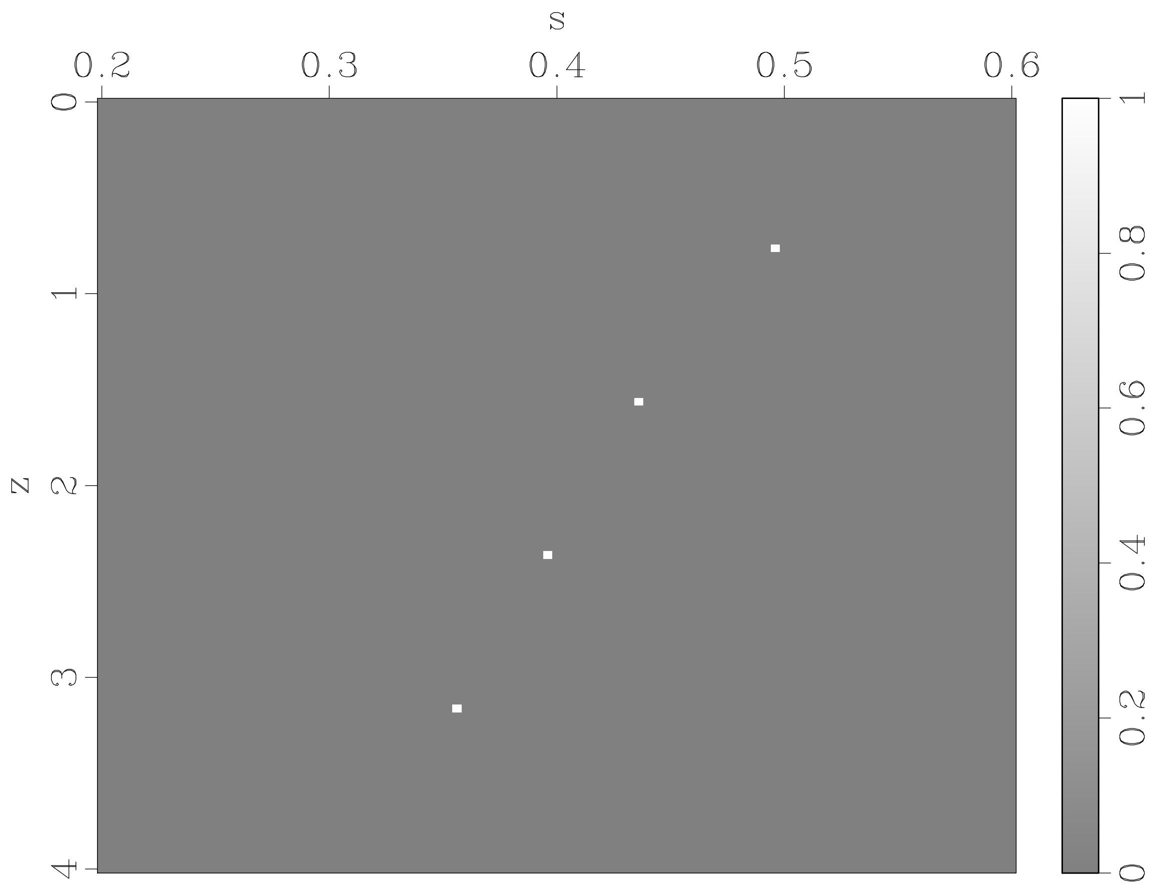

Figure 3: The true model showing the τ and s space representation of four hyperbolas. [**ER**]

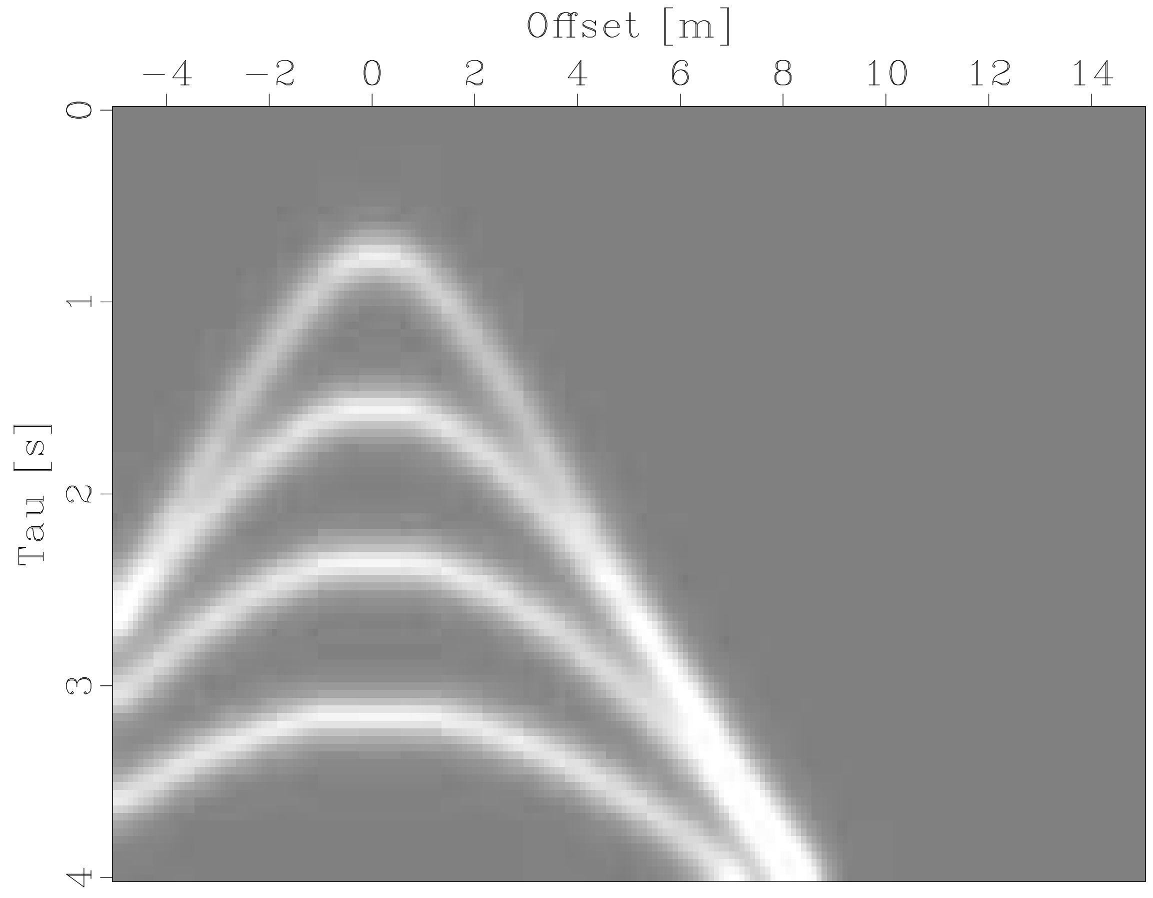

Figure 4: The initial data used for the CG inversion. In this case, we performed the forward operator on the true model and smoothed the result. [**ER**]

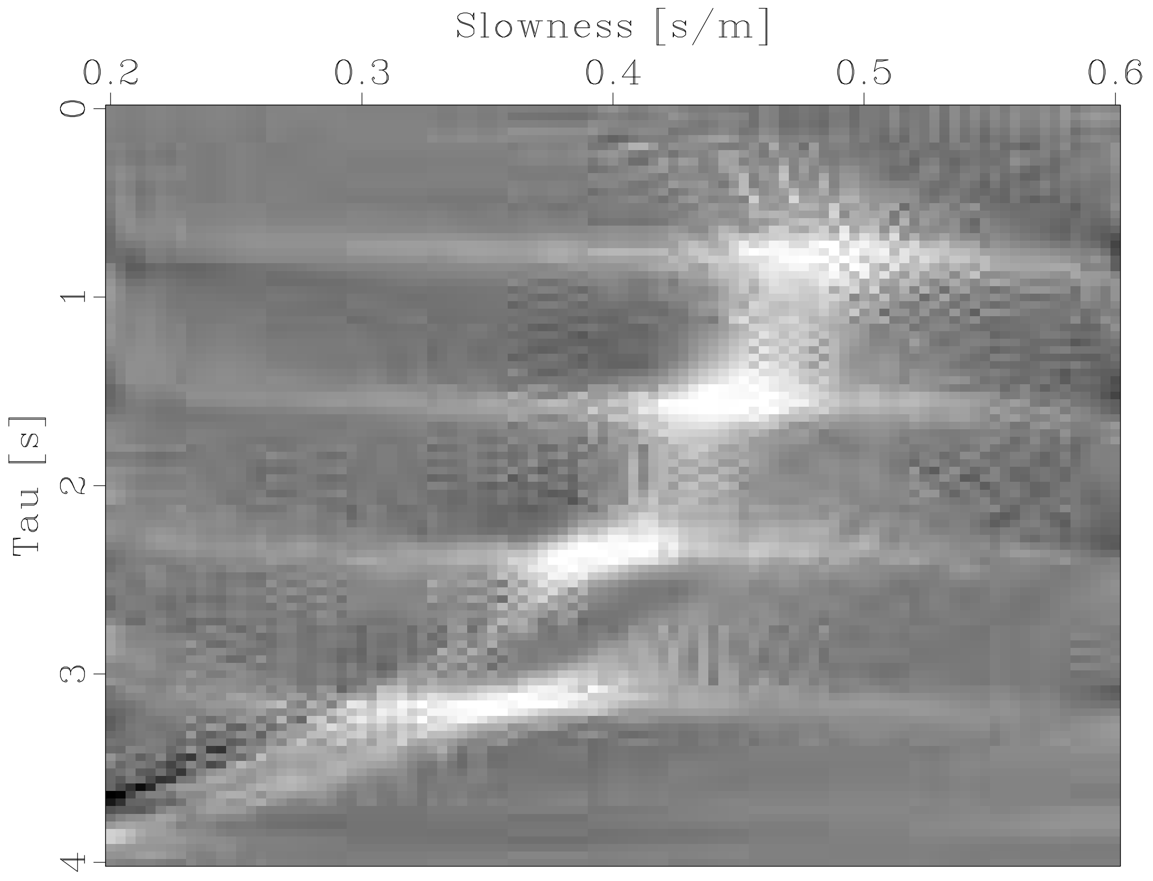

Figure 5: The result of performing 25 iterations of conjugate gradient inversion using the NMO operator. [**ER**]

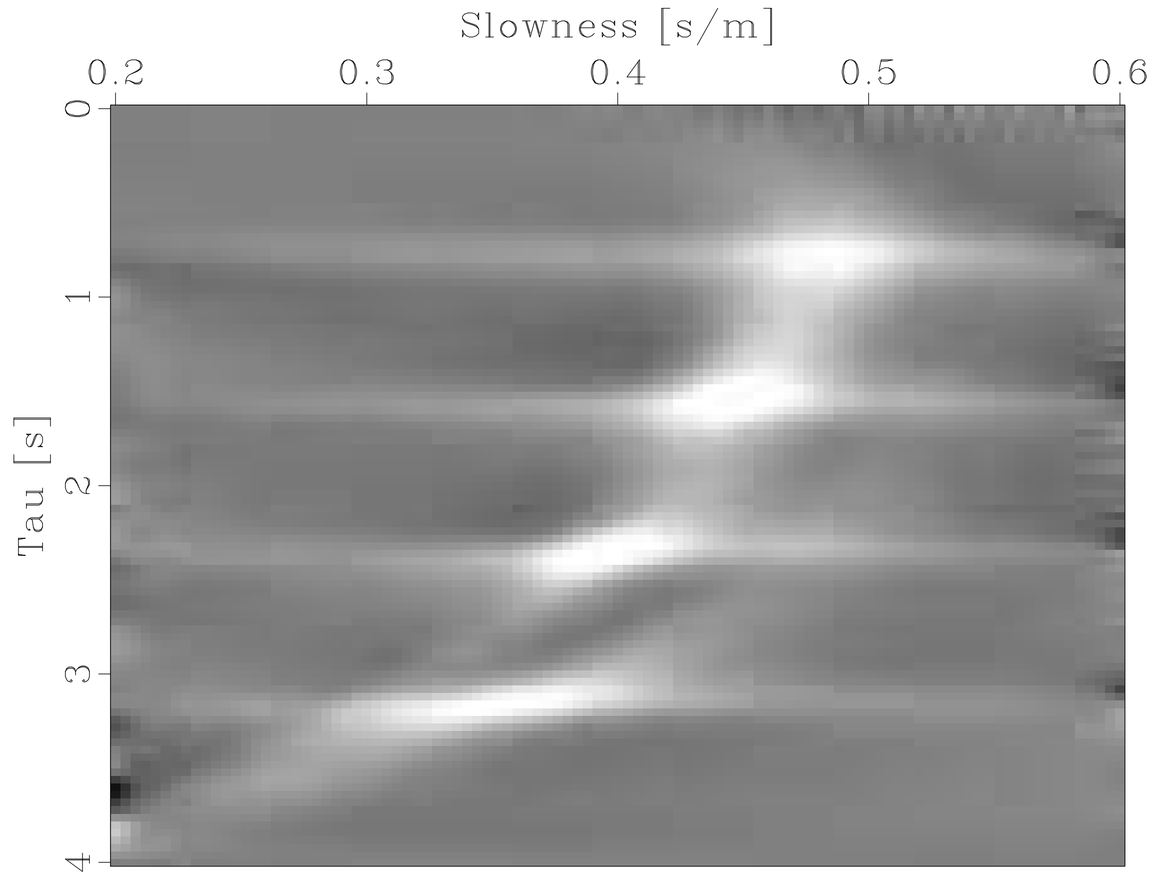

Figure 6: The result of performing 25 iterations of conjugate gradient inversion using the NMO operator and regularizing with the LaPlacian. [**ER**]

to C++. At this stage the labs for the class have been convereted but many of the examples have yet to be finished. We are also considering adding a python interface to the library. This would allow the class to be taught in ipython notebooks while still introducing all of the concepts needed for students to later use the C++ library.

CONCLUSION

In this paper we described the current state of a C++ non-linear inversion library. It allows the user to solve to store in-core linear and non-linear problems using an approach similar to SEP's Fortran 2003 and python equivalents.

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**, 39–70.

Biondi, E. and G. Barnier, 2017, A flexible out-of-core solver for linear/non-linear problems: SEP-Report, **168**, 1–15.

Claerbout, J., 2014, GEOPHYSICAL IMAGE ESTIMATION BY EXAMPLE.: LULU COM. (OCLC: 986953183).

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

Fomel, S. and J. Claerbout, 1996, Simple linear operators in Fortran 90: SEP-Report, **93**, 317–328.

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

Martin, E., R. G. Clapp, H. Le, C. Leader, and D. Nichols, 2014, SEPVector: A C++ inversion library: SEP-Report, **152**, 359–364.

Nichols, D., H. Urdaneta, H. I. Oh, J. Claerbout, L. Laane, M. Karrenbach, and M. Schwab, 1993, Programming geophysics in C++: SEP-Report, **79**, 313–471.

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