Full-waveform inversion, Part 3: Optimization

Philipp Witte[1], Mathias Louboutin[1], Keegan Lensink[1], Michael Lange[2], Navjot Kukreja[2], Fabio Luporini[2], Gerard Gorman[2], and Felix J. Herrmann[1,3]

Introduction

This tutorial is the third part of a full-waveform inversion (FWI) tutorial series with a step-by-step walkthrough of setting up forward and adjoint wave equations and building a basic FWI inversion framework. For discretizing and solving wave equations, we use Devito (http://www.opesci.org/devito-public), a Python-based domain-specific language for automated generation of finite-difference code (Lange et al., 2016). The first two parts of this tutorial (Louboutin et al., 2017, 2018) demonstrated how to solve the acoustic wave equation for modeling seismic shot records and how to compute the gradient of the FWI objective function using the adjoint-state method. With these two key ingredients, we will now build an inversion framework that can be used to minimize the FWI least-squares objective function.

FWI is a computationally and mathematically challenging problem. The computational complexity comes from the fact that an already expensive solution procedure for the wave equation needs to be repeated for a large number of source positions for each iteration of the optimization algorithm. The mathematical complexity comes from the fact that the FWI objective is known to have many local minima due to cycle skipping.

This tutorial demonstrates how we can set up a basic FWI framework with two alternative gradient-based optimization algorithms: stochastic gradient descent and the Gauss–Newton method (Nocedal and Wright, 2009).

We implement our inversion framework with the Julia Devito Inversion framework (JUDI) (https://github.com/slimgroup/JUDI.jl), a parallel software package for seismic modeling and inversion in the Julia programming language (Bezanson et al., 2012). JUDI provides abstractions and function wrappers that allow the implementation of wave-equation-based inversion problems such as FWI using code that closely follows the mathematical notation while using Devito's automatic code generation for solving the underlying wave equations.

The code to run the algorithms and generate the figures in this paper is available at http://github.com/seg/tutorials-2018.

Optimizing the FWI objective function

The goal of this tutorial series is to optimize the FWI objective function with the ℓ_2 misfit:

$$\underset{\mathbf{m}}{\text{minimize}}\, f(\mathbf{m}) = \sum_{i=1}^{n_s} \frac{1}{2} \left\| \mathbf{d}_i^{\text{pred}}(\mathbf{m}, \mathbf{q}_i) - \mathbf{d}_i^{\text{obs}} \right\|_2^2, \qquad (1)$$

where $\mathbf{d}_i^{\text{pred}}$ and $\mathbf{d}_i^{\text{obs}}$ are the predicted and observed seismic shot records of the i^{th} source location, and $\mathbf{m}$ is the velocity model (expressed as squared slowness). In Part 1, we demonstrated how to implement a forward modeling operator to generate the predicted shot records, which we will denote as $\mathbf{d}_i^{\text{pred}} = \mathbf{F}(\mathbf{m}; \mathbf{q}_i)$. In Part 2, we showed how we can compute the gradient $\nabla f(\mathbf{m})$ of the objective function and update our initial model using gradient descent.

There is a snag, however. This first-order optimization algorithm has a linear convergence rate at best and typically requires many iterations to converge. Second-order optimization methods converge considerably faster. To implement them, we first approximate the objective with a second-order Taylor expansion:

$$f(\mathbf{m}) = f(\mathbf{m}_0) + \nabla f(\mathbf{m}_0)\delta\mathbf{m} + \delta\mathbf{m}^{\mathrm{T}} \nabla^2 f(\mathbf{m}_0)\, \delta\mathbf{m} + \mathcal{O}(\delta\mathbf{m}^3), \qquad (2)$$

where $\mathcal{O}(\delta\mathbf{m}^3)$ represents the error term, $\nabla f(\mathbf{m}_0)$ is the gradient as implemented in Part 2, and $\nabla^2 f(\mathbf{m}_0)$ is the Hessian of the objective function, which we will refer to as $\mathbf{H}$. Rather than using the negative gradient to incrementally update our model, as in gradient descent, we directly calculate a model update $\delta\mathbf{m}$ that leads us to the minimum. This is called Newton's method:

$$\delta\mathbf{m} = -\mathbf{H}(\mathbf{m}_0)^{-1} \nabla f(\mathbf{m}_0). \qquad (3)$$

Although the method converges to the minimum of the FWI objective function quickly, it comes at the cost of having to compute and invert the Hessian matrix (Nocedal and Wright, 2009). Fortunately, for least-squares problems, such as FWI, the Hessian can be approximated by the Gauss-Newton (GN) Hessian $\mathbf{J}^{\mathrm{T}}\mathbf{J}$, where $\mathbf{J}$ is the Jacobian matrix. This is the partial derivative of the forward modeling operator $\mathbf{F}(\mathbf{m}; \mathbf{q})$ with respect to $\mathbf{m}$ — something we can easily compute. Furthermore, the Jacobian can also be used to express the gradient of the FWI objective function as $\nabla f(\mathbf{m}_0) = \mathbf{J}^{\mathrm{T}}(\mathbf{d}_i^{\text{pred}} - \mathbf{d}_i^{\text{obs}})$, where $\mathbf{J}^{\mathrm{T}}$ is the adjoint (transposed) Jacobian. This is useful, because we now have a set of operators $\mathbf{F}$, $\mathbf{J}$ and $\mathbf{H}_{GN} = \mathbf{J}^{\mathrm{T}}\mathbf{J}$, through which we can express both first- and second-order optimization algorithms for FWI.

Although forming these matrices explicitly is not possible, since they can become extremely large, we only need the action of these operators on vectors. This allows us to implement these operators matrix-free. In the following section, we will demonstrate how to set up these operators in our JUDI software framework and to how to use them to implement FWI algorithms.

Implementing FWI in JUDI

We start our demonstration by reading our data set, which consists of 16 shot records and was generated with an excerpt from the SEG/EAGE Overthrust model (Aminzadeh et al., 1997). We store it as a `judiVector`:

[1]The University of British Columbia, Seismic Laboratory for Imaging and Modeling (SLIM).
[2]Imperial College London.
[3]Georgia Institute of Technology.

https://doi.org/10.1190/tle37020142.1.

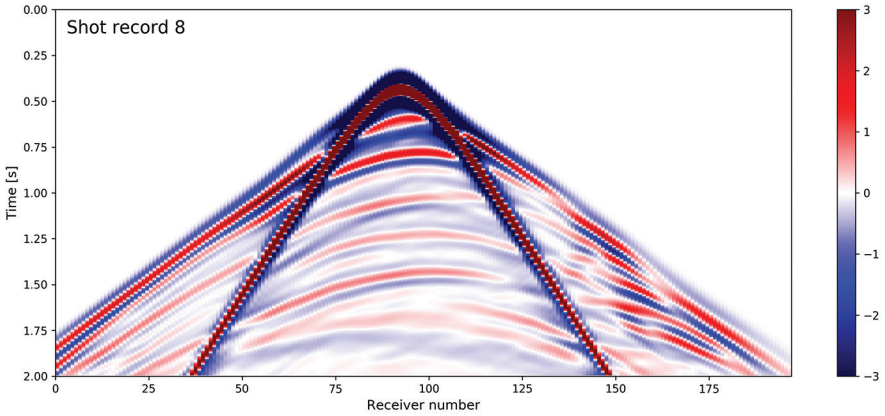

Figure 1. Observed shot record number 8.

```
block = segy_read("overthrust_shot_records.segy")
d_obs = judiVector(block);
```

JUDI vectors such as `d_obs` can be used like regular Julia vectors, so we can compute norms via `norm(d_obs)` or the inner product via `dot(d_obs, d_obs)`, but they contain the shot records in their original dimension. Shot records can be accessed via their respective shot number with `d_obs.data[shot_no]`, while the header information can be accessed with `d_obs.geometry`. We extract the source geometry from our SEG-Y file and then manually set up a source vector `q` with an 8 Hz Ricker wavelet:

```
f = 0.008  # kHz
src_geom = Geometry(block; key="source")
src_data = ricker_wavelet(src_geom.t[1], src_geom.dt[1], f)
q = judiVector(src_geom, src_data);
```

We will now set up the forward modeling operator $\mathbf{F}(\mathbf{m}; \mathbf{q})$ as a matrix-free operator for the inverse wave equation $\mathbf{A}(\mathbf{m})^{-1}$, where $\mathbf{m}$ is the current model, and source/receiver injection and sampling operators $\mathbf{P}_s$ and $\mathbf{P}_r$.

Since the dimensions of the inverse wave equation operator depend on the number of computational time steps, we calculate this number using the `get_computational_nt` function and set up an `info` object that contains some dimensionality information required by all operators.

Then we can define `Pr` and `Ps` as matrix-free operators implementing Devito sparse point injection and interpolation (Louboutin et al., 2017). Multiplications with `Ps` and `Pr` represent sampling the wavefield at source/receiver locations, while their adjoints `Ps', Pr'` denote injecting either source wavelets or shot records into the computational grid.

These projection and modeling operators are set up in Julia in the following way:

```
ntComp = get_computational_nt(q.geometry, d_obs.geometry, model0)
info = Info(prod(model0.n), d_obs.nsrc, ntComp)
Pr = judiProjection(info, d_obs.geometry)
Ps = judiProjection(info, q.geometry)
Ainv = judiModeling(info, model0);
```

The forward modeling step can be expressed mathematically as

$$F(\mathbf{m};\mathbf{q}) = \mathbf{P}_r\, \mathbf{A}^{-1}\,(\mathbf{m})\mathbf{P}_s^{\top}\, \mathbf{q}\,, \tag{4}$$

which is expressed in Julia as

```
d_pred = Pr * Ainv * Ps' * q
```

This forward models all 16 predicted shot records in parallel. Notice that, in instantiating `Ainv`, we made the wave equation solver implicitly dependent on `model0`.

Finally, we set up the matrix-free Jacobian operator `J` and the Gauss–Newton Hessian `J' * J`. As mentioned in the introduction, `J` is the partial derivative of the forward modeling operator $\mathbf{F}(\mathbf{m}; \mathbf{q})$ with respect to the model `m` and is therefore directly constructed from our modeling operator `Pr * Ainv * Ps'` and a specified source vector `q`:

```
op = Pr * Ainv * Ps'
J = judiJacobian(op, q);
```

In the context of seismic inversion, the Jacobian is also called the linearized modeling or demigration operator, and its adjoint `J'` is the migration operator. One drawback of this notation is that the forward wavefields for the gradient calculation have to be recomputed since the forward modeling operator only returns the shot records and not the complete wavefields. For this reason, JUDI has an additional function for computing the gradients of the FWI objective function `f,g = fwi_objective(model0,q[i], d_obs[i])`, which takes the current model, source and data vectors as an input and computes the objective value and gradient in parallel without having to recompute the forward wavefields.

FWI via gradient descent

With expressions for modeling operators, Jacobians and gradients of the FWI objective, we can now implement different FWI algorithms in a few lines of code. We will start with a basic gradient descent example with a line search. To reduce the computational cost of full gradient descent, we will use a stochastic approach in which we only compute the gradient and function value for a randomized subset of source locations. In JUDI, this is accomplished by choosing a random vector of integers between 1 and 16

and indexing the data vectors as described earlier. Furthermore, we will apply a projection operator `proj(x)`, which prevent velocities (or squared slownesses) becoming negative or too large by clipping values outside the allowed range.

A few extra variables are defined in the notebook, but the full algorithm for FWI with stochastic gradient descent and box constraints is implemented as follows:

```
for j=1:maxiter
    # FWI objective function value and gradient.
    i = randperm(d_obs.nsrc)[1:batchsize]
    fval, grad = fwi_objective(model0, q[i], d_obs[i])

    # Line search and update model.
    update = backtracking_linesearch(model0,
                                     q[i],
                                     d_obs[i],
                                     fval,
                                     grad,
                                     proj;
                                     alpha=1f0)
    model0.m += reshape(update, model0.n)

    # Apply box constraints.
    model0.m = proj(model0.m)
end
```

JUDI's `backtracking_linesearch` function performs an approximate line search and returns a model update that leads to a decrease of the objective function value (Armijo condition; Nocedal and Wright, 2009). The result after 10 iterations of SGD with box constraints is shown in Figure 2. In practice, where starting models are typically less accurate than in our example, FWI is often performed from low to high frequencies, since the objective function has less local minima for lower frequencies (Bunks et al., 1995). In this multiscale FWI approach, a low-pass-filtered version of the data is used to invert for a low-resolution velocity model first, and higher frequencies are added in subsequent iterations.

FWI via the Gauss–Newton method

As discussed earlier, the convergence rate of GD depends on the objective function, but requires many FWI iterations necessary to reach an acceptable solution. Using our matrix-free operator for the Jacobian J, we can modify the above code to implement the Gauss–Newton method (equation 3) to improve the convergence rate. In practice, directly inverting the Gauss–Newton Hessian `J' * J` should be avoided, because the matrix is badly conditioned and takes many iterations to invert. Instead, we perform a few iterations of a least-squares solver, `lsqr()`, to approximately solve `J * p = d_pred - d_obs` and obtain the update direction `p`. `lsqr`, from the Julia `IterativeSolvers` package, is a conjugate-gradient type algorithm for solving least-squares problems and is mathematically equivalent to inverting `J' * J`, but has better numerical properties (Paige and Saunders, 1982). We implement the Gauss-Newton method as follows:

```
for j=1:maxiter
    # Model predicted data.
    d_pred = Pr * Ainv * Ps' * q

    # GN update direction.
    p = lsqr(J, d_pred - d_obs; maxiter=6)

    # update model and box constraints.
    model0.m = model0.m - reshape(p, model0.n)
end
```

In contrast to our SGD algorithm, we use all shot records in every iteration, since stochastic methods for second-order algorithms are less well understood, making this approach considerably more expensive than our previous algorithm. However, as shown in Figures 2 and 3, it achieves a superior result, with a considerably lower misfit

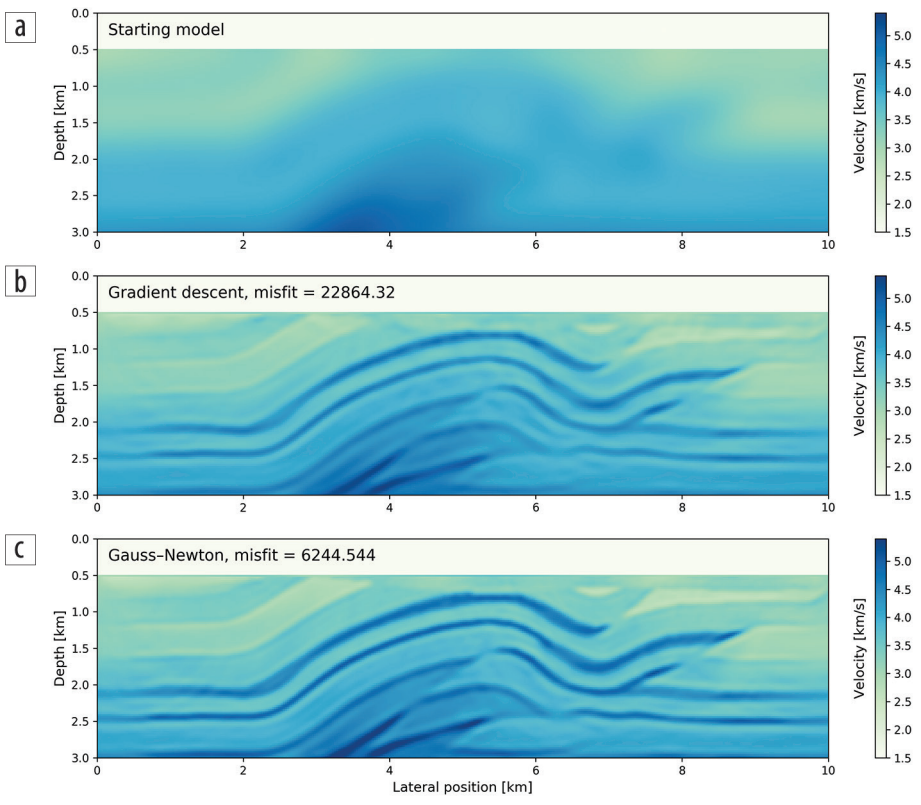

Figure 2. (a) Initial model. (b) Recovered velocity model after 10 iterations of stochastic gradient descent with box constraints and a batch size of eight shots. (c) Recovered velocity model after 10 iterations of the Gauss–Newton method, with six iterations of LSQR for the Gauss–Newton subproblem, and using all shots in every iteration. The resulting misfit is substantially lower than what was achieved with gradient descent.

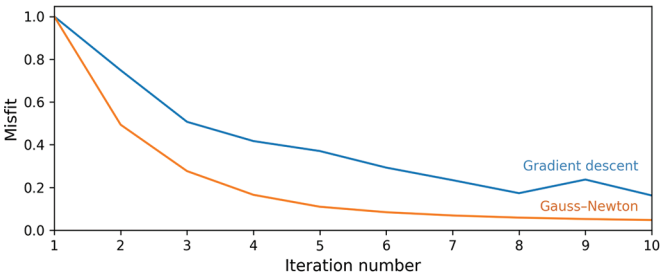

Figure 3. Normalized function values for the FWI inversion example with stochastic gradient descent and the Gauss-Newton method.

compared to the known model. Furthermore, Figure 3 shows that it achieves the improved result in relatively few iterations.

An alternative to (Gauss–)Newton methods are quasi-Newton methods, which build up an approximation of the Hessian from previous gradients only and require no additional PDE solves or matrix inversions. Implementing an efficient and correct version of this method, such as the L-BFGS algorithm, exceeds a few lines of code, and we therefore leave this exercise to the reader. Instead of implementing more complicated algorithms by hand, it is also possible to interface third-party Julia optimization libraries; an example for this is given in the notebook **fwi_example_NLopt.ipynb**.

Even though all examples shown here are two-dimensional, to make them reproducible on a laptop or desktop PC, JUDI can be used for 3D modeling and inversion without having to change the code, since the number of dimensions are automatically inferred from the velocity model and data dimensions.

Conclusions

In this final part of our FWI tutorial series, we demonstrated how to set up basic optimization algorithms for waveform inversion using JUDI. The methods shown here are all gradient based and differ in the way update directions for the velocity model are computed. Our numerical examples can serve for the reader as a basis for developing more advanced FWI workflows, which usually include additional data preprocessing, frequency continuation techniques, or further model constraints. TLE

Acknowledgments

This research was carried out as part of the SINBAD II project with the support of the member organizations of the SINBAD Consortium. This work was financially supported in part by EPSRC grant EP/L000407/1 and the Imperial College London Intel Parallel Computing Centre.

Corresponding author: pwitte@eoas.ubc.ca

References

Aminzadeh, F., J. Brac, and T. Kunz, 1997, 3D salt and overthrust models, SEG/EAGE Modeling Series No. 1: SEG.

Bezanson, J., S. Karpinski, V. B. Shah, and A. Edelman, 2012, Julia: A fast dynamic language for technical computing: CoRR. Retrieved from http://arxiv.org/abs/1209.5145.

Bunks, C., F. M. Saleck, S. Zaleski, and G. Chavent, 1995, Multiscale seismic waveform inversion: Geophysics, **60**, no. 5, 1457–1473, https://doi.org/10.1190/1.1443880.

Lange, M., N. Kukreja, M. Louboutin, F. Luporini, F. V. Zacarias, V. Pandolfo, V. Paulius, K. Paulius, and G. Gorman, 2016, Devito: Towards a generic finite difference DSL using symbolic python: 6th Workshop on Python for High-performance and Scientific Computing, https://doi.org/10.1109/PyHPC.2016.013.

Louboutin, M., P. Witte, M. Lange, N. Kukreja, F. Luporini, G. Gorman, and F. J. Herrmann, Full-waveform inversion, Part 1: Forward modeling, The Leading Edge, **36**, no. 12, 1033–1036, https://doi.org/10.1190/tle36121033.1.

Louboutin, M., P. Witte, M. Lange, N. Kukreja, F. Luporini, G. Gorman, and F. J. Herrmann, 2018, Full-waveform inversion, Part 2: Adjoint modeling, The Leading Edge, **37**, no. 1, 69–72, https://doi.org/10.1190/tle37010069.1.

Nocedal, J., and S. Wright, 2006, Numerical optimization, 2nd edition: Springer, https://doi.org/10.1007/978-0-387-40065-5.

Paige, C. C., and M. A. Saunders, 1982, LSQR: An algorithm for sparse linear equations and sparse least squares: ACM Transactions on Mathematical Software, **8**, no. 1, 43–71, https://doi.org/10.1145/355984.355989.