Geophysical Tutorial — Coordinated by Matt Hall

The conjugate gradient method

Karl Schleicher[1]

The conjugate gradient method can be used to solve many large linear geophysical problems — for example, least-squares parabolic and hyperbolic Radon transform, traveltime tomography, least-squares migration, and full-waveform inversion (FWI) (e.g., Witte et al., 2018). This tutorial revisits the "Linear inversion tutorial" (Hall, 2016) that estimated reflectivity by deconvolving a known wavelet from a seismic trace using least squares. This tutorial solves the same problem using the conjugate gradient method. This problem is easy to understand, and the concepts apply to other applications. The conjugate gradient method is often used to solve large problems because the least-squares algorithm is much more expensive — that is, even a large computer may not be able to find a useful solution in a reasonable amount of time.

Introduction

The conjugate gradient method was originally proposed by Hestenes (1952) and extended to handle rectangular matrices by Paige and Saunders (1982). Claerbout (2012) demonstrates its application to geophysical problems. It is an iterative method. Each iteration applies the linear operator and its adjoint. The initial guess is often the zero vector, and computation may stop after very few iterations.

The adjoint of the operator $\mathbf{A}$, denoted as $\mathbf{A}^{\mathrm{H}}$, is defined as the operator that satisfies $\langle \mathbf{Ax}, \mathbf{y} \rangle = \langle \mathbf{x}, \mathbf{A}^{\mathrm{H}}\mathbf{y} \rangle$ for all vectors $\mathbf{x}$ and $\mathbf{y}$ (where $\langle \mathbf{u}, \mathbf{v} \rangle$ represents the inner product between vectors $\mathbf{u}$ and $\mathbf{v}$). For a given matrix, the adjoint is simply the complex conjugate of the transpose of the matrix; this is also sometimes known as the Hermitian transpose and is sometimes written as $\mathbf{A}^{*}$ or $\mathbf{A}^{\dagger}$. Just to muddy the notation water even further, the complex conjugate transpose is denoted by `A.H` in NumPy and `A'` in Matlab or Octave. However, we will implement the adjoint operator without forming any matrices.

Many linear operators can be programmed as functions that are more intuitive and efficient than matrix multiplication. The matrices for operators like migration and FWI would be huge, but we avoid this problem because once you have the program for the linear operator, you can write the adjoint operator without computing matrices. Implementing the conjugate gradient algorithm using functions to apply linear operators and their adjoints is practical and efficient. It is wonderful to see programs that implement linear algorithms without matrices, and the programming technique is a key theme in Claerbout's 2012 book.

This tutorial provides a quick start to the conjugate gradient method based on Guo's pseudocode (2002). Those interested in more depth can read Claerbout (2012) and Shewchuk (1994). A Jupyter Notebook with Python code to reproduce the figures in this tutorial is at https://github.com/seg/tutorials.

The forward and adjoint operators

Starting with a known reflectivity model $\mathbf{m}$, we create synthetic seismic data $\mathbf{d} = \mathbf{Fm}$, where $\mathbf{F}$ is the linear operator that performs the function "convolve with a Ricker wavelet." Given such a trace and the operator, the conjugate gradient method can be used to estimate the original reflectivity.

Hall (2016) calls the operator $\mathbf{G}$ instead of $\mathbf{F}$, and he creates a matrix by shifting the wavelet and padding with zeros. In contrast, I implement the operator using the NumPy function `convolve()`. This is advantageous because it allows us to solve the linear equation $\hat{\mathbf{m}} = \mathbf{F}^{-1}\mathbf{d}$ without ever having to construct (or invert) this matrix, which can become very large. This matrix-free approach is faster and uses less memory than the matrix implementation.

We can add one more feature to the operator and implement it with its adjoint. A convenient way to combine the two operations is to use a so-called "object-oriented programming" approach and define a Python class. Then we can have two methods (i.e., functions) defined on the class: `forward`, implementing the forward operator, and `adjoint` for the adjoint operator, which in this case is correlation.

```
class Operator(object):
    """A linear operator.
    """
    def __init__(self, wavelet):
        self.wavelet = wavelet

    def forward(self, v):
        """Defines the forward operator.
        """
        return np.convolve(v, self.wavelet, mode='same')

    def adjoint(self, v):
        """Defines the adjoint operator.
        """
        return np.correlate(v, self.wavelet, mode='same')
```

Claerbout (2012) teaches how to write this kind of symmetrical code and provides many examples of geophysical operators with adjoints (e.g., derivative versus negative derivative, causal integration versus anticausal integration, stretch versus squeeze, truncate versus zero pad). Writing functions to apply operators is more efficient than computing matrices.

Now that we have the operator, we can instantiate the class with a wavelet. This wavelet will be "built in" to the instance `F`.

```
F = Operator(wavelet)
```

[1] University of Texas, Bureau of Economic Geology, Jackson School of Geosciences.

https://doi.org/10.1190/tle37040296.1.

Now we can compute $\mathbf{d} = \mathbf{Fm}$ simply by passing the model `m` to the method `F.forward()`, which already has the wavelet:

```
d = F.forward(m)
```

This results in the synthetic seismogram shown in Figure 1.

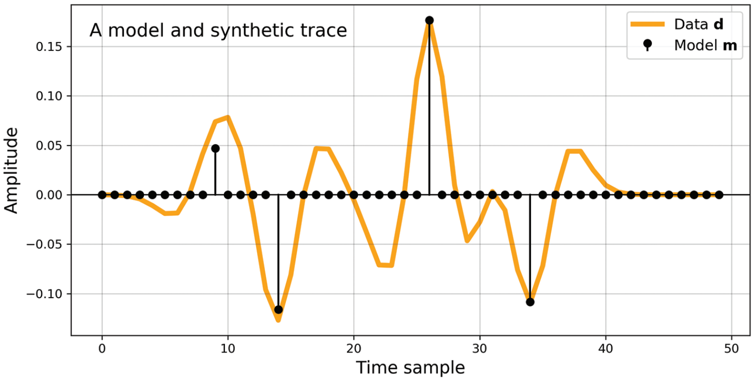

Figure 1. A plot of reflectivity model **m** (black) and the synthetic seismic data **d** (orange).

The conjugate gradient method

Now that we have a synthetic, we wish to solve the linear inverse problem and estimate the model **m** from the data **d**. The model can not be completely recovered because the Ricker wavelet is band limited, so some information is lost.

One way to solve linear problems is to start with an initial guess and iteratively improve the solution. The next few paragraphs derive an iterative method. You do not need to understand all the derivation, so you might want to lightly read it and move to the section about the pseudocode.

We start with an initial estimate for the model, $\hat{\mathbf{m}}_0 = \mathbf{0}$ (the zero vector) and compute the residual $\mathbf{r}_0 = \mathbf{d} - \mathbf{F}\hat{\mathbf{m}}_0$ (i.e., the difference between the data and the action of the forward operator on the model estimate). A good measure of the error in the initial solution is the inner product $\langle \mathbf{r}_0, \mathbf{r}_0 \rangle$ or `np.dot(r0, r0)` in code. This is equivalent to the squared norm (length) of the residual vector $\mathbf{r}_0$, and constitutes our cost function. If the cost is 0, or within some small tolerance, then we have a solution.

We can improve the estimate, $\hat{\mathbf{m}}_0$ by selecting a direction $\mathbf{s}_0$ and a scale α_0 to move $\hat{\mathbf{m}}_0$ to a new guess, $\hat{\mathbf{m}}_1 = \hat{\mathbf{m}}_0 + \alpha_0\, \mathbf{s}_0$. You can compute the direction in model space that most rapidly decreases the error. This is the gradient of $\langle \mathbf{d} - \mathbf{F}\hat{\mathbf{m}}, \mathbf{d} - \mathbf{F}\hat{\mathbf{m}} \rangle$ or $\langle \mathbf{r}_0, \mathbf{r}_0 \rangle$. If you grind through the mathematics, the gradient $\mathbf{g}_0$ turns out to be given by the action of the adjoint operator on the residual: $\mathbf{g}_0 = \mathbf{F}^{\mathrm{H}}\mathbf{r}_0$.

The scalar α_0 is computed to minimize the residual, $\mathbf{r}_1 = \mathbf{d} - \mathbf{F}\hat{\mathbf{m}} = \mathbf{d} - \mathbf{F}(\hat{\mathbf{m}} + \alpha_0\, \mathbf{s}_0) = \mathbf{r}_0 - \alpha_0 \mathbf{F}\mathbf{s}_0$. We can then compute α_0 by taking the derivative of $\langle \mathbf{r}_1, \mathbf{r}_1 \rangle$ with respect to α_0, setting the derivative to 0, and solving for α_0. The result is:

$$\alpha_0 = \langle \mathbf{g}_0, \mathbf{g}_0 \rangle \,/\, \langle \mathbf{F}\mathbf{s}_0, \mathbf{F}\mathbf{s}_0 \rangle. \qquad (1)$$

These equations define a reasonable approach to iteratively improve an estimated solution, and it is called "the steepest descent method." The conjugate gradient algorithm builds on this. It is not much harder to implement, has similar cost per iteration, and faster convergence.

The first iteration of the conjugate gradient method is the same as the steepest descent method. The second (and later) iterations compute the gradient $\mathbf{g}_1$ and $\mathbf{F}\mathbf{g}_1$. With $\mathbf{s}_0$ and $\mathbf{F}\mathbf{s}_0$ from the previous iteration we can then compute the step direction. Scalars α and β are computed to minimize

$$\mathbf{r}_2 = \mathbf{d} - \mathbf{F}(\hat{\mathbf{m}}_1 + \alpha\, \mathbf{s}_0 + \beta\, \mathbf{g}_1) \qquad (2)$$

Some mathematical manipulations determine the best direction is $\mathbf{s}_1 = \mathbf{g}_1 + \beta\, \mathbf{s}_0$ where $\beta = \langle \mathbf{g}_1, \mathbf{g}_1 \rangle \,/\, \langle \mathbf{g}_0, \mathbf{g}_0 \rangle$. The conjugate gradient algorithm is guaranteed to converge when the number of iterations is equal to the dimension of $\hat{\mathbf{m}}$, but only a few iterations often give sufficient accuracy. For our implementation, we'll start with a simplifed version of the pseudocode provided by Guo (2002), with its implementation in Python on the right:

```
$\hat{\mathbf{m}} = \mathbf{0}$
$\mathbf{r} = \mathbf{d} - \mathbf{F}\hat{\mathbf{m}}$
$\mathbf{s} = \mathbf{0}$
$\beta = 0$

iterate $n$ times:
    $\mathbf{g} = \mathbf{F}^{\mathrm{H}}\mathbf{r}$
    if not first iteration :
        $\beta = \langle \mathbf{g}, \mathbf{g} \rangle / \gamma$
    $\gamma = \langle \mathbf{g}, \mathbf{g} \rangle$
    $\mathbf{s} = \mathbf{g} + \beta\mathbf{s}$
    $\Delta\mathbf{r} = \mathbf{F}\mathbf{s}$
    $\alpha_0 = \langle \mathbf{g}_0, \mathbf{g}_0 \rangle / \langle \Delta\mathbf{r}, \Delta\mathbf{r} \rangle$
    $\hat{\mathbf{m}} = \hat{\mathbf{m}} - \alpha\mathbf{s}$
    $\mathbf{r} = \mathbf{r} + \alpha\Delta\mathbf{r}$
```

```
m_est = np.zeros_like(d)
r = d - F.forward(m_est)
s = np.zeros_like(d)
beta = 0

for i in range(n):
    g = F.adjoint(r)
    if i != 0:
        beta = np.dot(g, g) / gamma
    gamma = np.dot(g, g)
    s = g + beta * s
    deltar = F.forward(s)
    alpha = gamma / np.dot(deltar, deltar)
    m_est = m_est + alpha * s
    r = r - alpha * deltar
```

Results

The Python code in the previous section was used to invert for reflectivity. Figure 2 shows the five iterations of the conjugate gradient method. The conjugate gradient method converged in only four iterations; the results of the fourth and fifth iteration almost exactly overlay on the plot. Fast convergence is important for a practical algorithm. Convergence is guaranteed in 50 iterations (the dimension of the model).

Figure 3 compares the original model and the model estimated using conjugate gradient inversion. Conjugate gradient inversion does not completely recover the model because the Ricker wavelet is band limited, but side lobes are reduced compared to the data.

Finally, we compute the predicted data from the estimated model:

```
d_pred = F.forward(m_est)
```

Figure 4 compares the predicted data with the original data to show that we have done a good job of the estimation. This demonstrates more than one reflectivity sequence when convolved with the Ricker wavelet fits the data, in particular the original model and the model estimated by the conjugate gradient method. It may be interesting to explore preconditioning operators that promote a sparse solution.

The Jupyter notebook provided with this tutorial further explores finding least-squares solutions using the conjugate gradient method. The notebook demonstrates how preconditioning can be used to promote a sparse solution. It also provides examples using the solver provided in the SciPy package.

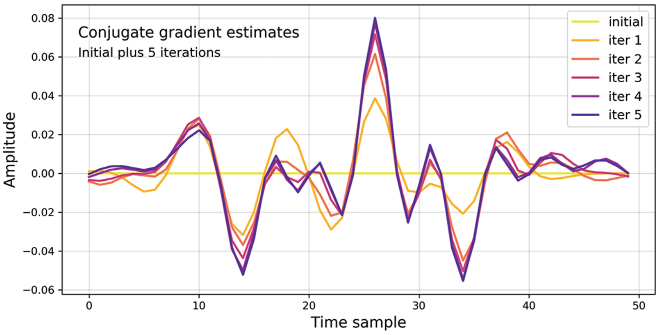

Figure 2. The initial estimate and the first 5 iterations of conjugate gradient. The fifth iteration almost exactly overlays the fourth.

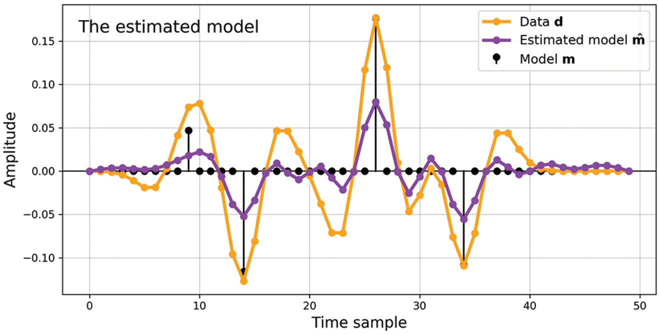

Figure 3. Comparison of the model (black) and the model estimated using conjugate gradient inversion (purple), along with the data (orange) for comparison.

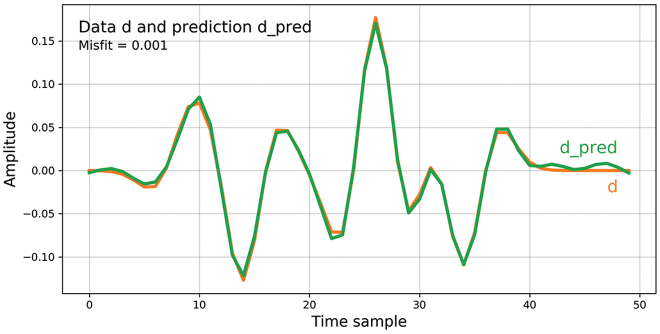

Figure 4. Comparison of the predicted data `d_pred` from the conjugate gradient inversion with the original data `d`. It overplots the data almost exactly.

Conclusions

I described the conjugate gradient algorithm and presented an implementation. This is an iterative method that requires functions to apply the linear operator and its adjoint. Many linear operators that are familiar geophysical operations like convolution are more efficiently implemented without matrices. The reflectivity estimation problem described in Hall (2016) was solved using the conjugate gradient method. Convergence only took four iterations. The conjugate gradient method is often used to solve large problems because well-known solvers like least squares are much more expensive.

Acknowledgments

The SEG Seismic Working Workshop on Reproducible Tutorials held 9–13 August 2017 in Houston inspired this tutorial. For more information, visit http://ahay.org/wiki/Houston_2017.

Corresponding author: k_schleicher@hotmail.com

References

Claerbout, J., and S. Fomel, 2012, Image estimation by example, http://sepwww.stanford.edu/sep/prof/gee1-2012.pdf, accessed 6 March 2018.

Guo, J., H. Zhou, J. Young, and S. Gray, 2002, Merits and challenges for accurate velocity model building by 3D gridded tomography: 72nd Annual International Meeting, SEG, Expanded Abstracts, 854–857, https://doi.org/10.1190/1.1817395.

Hall, M., 2016, Linear Inversion: The Leading Edge, **35**, no. 12, 1085–1087, https://doi.org/10.1190/tle35121085.1.

Hestenes, M. R., and E. Stiefel, 1952, Methods of conjugate gradients for solving linear systems: Journal of Research of the National Bureau of Standards, **49**, no. 6, https://doi.org/10.6028/jres.049.044.

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.

Shewchuk, J. R., 1994, An introduction to the conjugate gradient method without the agonizing pain, http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf, accessed 6 March 2018.

Witte, P, M. Louboutin, K. Lensink, M. Lange, N. Kukreja, F. Luporini, G. Gorman, and F. J. Herrman, 2018, Full-waveform inversion, Part 3: Optimization: The Leading Edge, **37**, no. 2, 142–145, https://doi.org/10.1190/tle37020142.1.