# NOT FOR MANUSCRIPT  
import sys  
  
import numpy as np  
import scipy as sp  
%matplotlib inline  
import matplotlib.pyplot as plt  
  
print ("sys.version=", sys.version)  
print ("np.\_\_version\_\_=", np.\_\_version\_\_)

sys.version= 3.6.3 |Anaconda, Inc.| (default, Oct 6 2017, 12:04:38)   
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE\_401/final)]  
np.\_\_version\_\_= 1.13.3

### Geophysical Tutorial – coordinated by Matt Hall

# The conjugate gradient method

Karl Schleicher, k\_schleicher@hotmail.com

The conjugate gradient method can be used to solve many large linear geophysical problems. For example, least squares parabolic and hyperbolic radon transform, travel time tomography, least squares migration, and full waveform inversion (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 & Saunders (1982). Claerbout (2012) demonstrated 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 , denoted as , is defined as the operator that satisfies = for all vectors and (where represents the inner product between vectors and ). 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 sometimes written as or . 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.

# NOT FOR MANUSCRIPT  
A = np.matrix([[2,3], [(6+3j),-7]])  
  
print("matrix")  
print(A)  
print("\ntranspose")  
print(A.T)  
print("\ncomplex conjugate transpose, aka Hermitian transpose")  
print(A.H)

matrix  
[[ 2.+0.j 3.+0.j]  
 [ 6.+3.j -7.+0.j]]  
  
transpose  
[[ 2.+0.j 6.+3.j]  
 [ 3.+0.j -7.+0.j]]  
  
complex conjugate transpose, aka Hermitian transpose  
[[ 2.-0.j 6.-3.j]  
 [ 3.-0.j -7.-0.j]]

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.

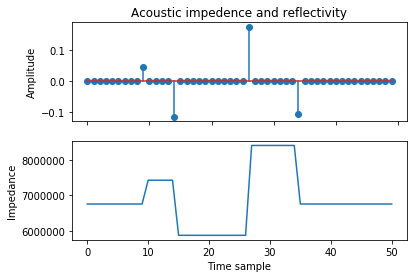
# OMIT SET-UP FROM MANUSCRIPT

## SET-UP — Construct the model m

We start with the slightly odd-seeming number of samples 51. This is because when we calculate the impedance contrasts (reflectivities), we’ll lose a sample. Since I’d like 50 samples in the final reflectivity model m, we have to start with 50 + 1 samples in the impedance model.

# NOT FOR MANUSCRIPT  
  
# Impedance, imp VP RHO  
imp = np.ones(51) \* 2550 \* 2650  
imp[10:15] = 2700 \* 2750  
imp[15:27] = 2400 \* 2450  
imp[27:35] = 2800 \* 3000  
  
# Compute reflectivity.  
m = (imp[1:] - imp[:-1]) / (imp[1:] + imp[:-1])

# NOT FOR MANUSCRIPT  
fig, axs = plt.subplots(nrows=2)  
  
ax = axs[0]  
ax.set\_title('Acoustic impedence and reflectivity')  
ax.set\_ylabel('Amplitude')  
ax.set\_xticklabels([])  
ax.stem(m)  
  
ax = axs[1]  
ax.plot(imp)  
ax.set\_xlabel('Time sample')  
ax.set\_ylabel('Impedance')  
  
plt.show()



png

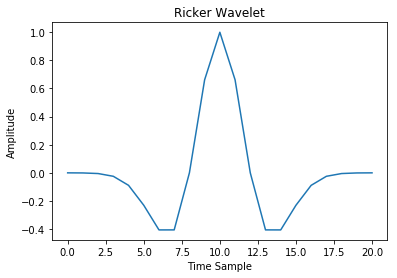
We lost a sample, leaving 50. This is exactly we wanted.

## SET-UP — Make a wavelet

Now we make a Ricker wavelet to convolve on the reflectivity.

# NOT FOR MANUSCRIPT  
from scipy.signal import ricker  
  
# Use an odd number so we get a centered peak.  
wavelet = ricker(points=21, a=2)  
  
# Normalize the wavelet amplitude to 1 so that the amplitude  
# relates directly to the reflectivity.  
wavelet /= np.amax(wavelet)

# NOT FOR MANUSCRIPT  
fig, ax = plt.subplots()  
  
ax.set\_title('Ricker Wavelet')  
ax.set\_xlabel('Time Sample')  
ax.set\_ylabel('Amplitude')  
ax.plot(wavelet)  
  
plt.show()



png

# END OF SET-UP

## The forward and adjoint operators

Starting with a known reflectivity model , we create synthetic seismic data , where 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) called the operator instead of , and he created 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 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')

# NOT FOR MANUSCRIPT  
# Dot product test.  
w = np.array([1.0, -2.0, 0.5, 0, 0])  
x = np.random.rand(50)  
y = np.random.rand(50)  
  
F = Operator(w)  
  
convx = F.forward(x)  
cadjy = F.adjoint(y)  
  
print('<Fx, y> =', np.dot(convx, y))  
print('<x, F\*y> =', np.dot(x, cadjy))  
print('diff =', np.dot(convx, y) - np.dot(x, cadjy))

<Fx, y> = -8.604494486  
<x, F\*y> = -8.604494486  
diff = 0.0

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

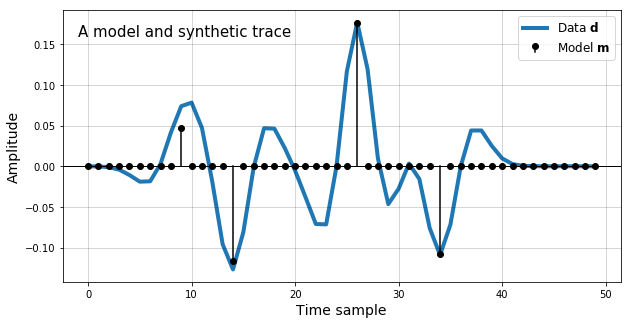
F = Operator(wavelet)

Now we can compute 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.

# NOT FOR MANUSCRIPT  
fig, ax = plt.subplots(figsize=(10, 5))  
  
plt.stem(m, label="Model $\mathbf{m}$", basefmt='none', linefmt='k', markerfmt='ko')  
plt.axhline(c='k', lw=1)  
ax.plot(d, lw=4, zorder=1, label="Data $\mathbf{d}$")  
ax.text(-1, 0.16, 'A model and synthetic trace', size=15)  
ax.set\_xlabel('Time sample', size=14)  
ax.set\_ylabel('Amplitude', size=14)  
ax.legend(fontsize=12)  
ax.grid(color='k', alpha=0.2)  
  
plt.savefig('figure1\_model\_data.png', dpi=300)  
plt.show()



png

**Figure 1**: A plot of reflectivity model (black) and the synthetic seismic data (blue).

## The conjugate gradient method

Now that we have a synthetic, we wish to solve the linear inverse problem and estimate the model from the data . The model can not be completely recovered because the Ricker wavelet is bandlimited, 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 pseudo code.

We start with an initial estimate for the model, (the zero vector) and compute the residual (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 or np.dot(r0, r0) in code. This is equivalent to the squared norm (length) of the residual vector , 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, by selecting a direction and a scale to move to a new guess, . You can compute the direction in model space that most rapidly decreases the error. This is the gradient of $ -  , -  $ or . If you grind through the mathematics, the gradient turns out to be given by the action of the adjoint operator on the residual: .

The scalar is computed to minimize the residual, . We can them compute by taking the derivative of with respect to , setting the derivative to 0, and solving for . The result is:

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 and . With and from the previous iteration we can then compute the step direction. Scalars and are computed to minimize

Some mathematical manipulations determine the best direction is where . The conjugate gradient algorithm is guaranteed to converge when the number of iterations is equal to the dimension of , but only a few iterations often give sufficient accuracy. For our implementation, we’ll start with a simplifed version of the pseudo-code provided by Guo (2002):

# NOT FOR MANUSCRIPT  
n = 5

# NOT FOR MANUSCRIPT  
# This version has the cache, which is just a convenient  
# way to store the results of the iterations.  
m\_est = np.zeros\_like(d)  
r = d - F.forward(m\_est)  
s = np.zeros\_like(d)  
beta = 0  
  
cache = [m\_est]  
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 = -np.dot(r, deltar) / np.dot(deltar, deltar)  
 m\_est = m\_est - alpha \* s  
 r = r + alpha \* deltar  
 cache.append(m\_est)

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 = -np.dot(r, deltar) / np.dot(deltar, deltar)  
 m\_est = m\_est - alpha \* s  
 r = r + alpha \* deltar

## Results

The conjugate gradient pseudo-code in the previous section was implemented in Python and used to invert for the 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.

Figure 3 compares the original model, the data, and the model estimated using conjugate gradient inversion. Conjugate gradient inversion does not completely recover the model because the Ricker wavelet is bandlimitted, but sidelobes are reduced (for example samples 17 through 23).

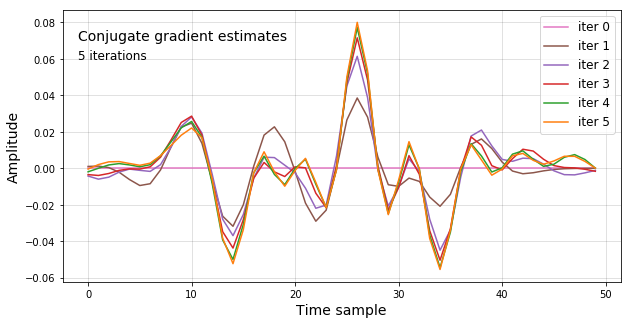
Finally, we compute the predicted data from the estimated model:

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 there is more than one reflectivity sequence that when convolved with the Ricker wavelet that will fit 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 sparse solvers provided in the SciPy package. You can find the notebooks at https://github.com/seg/tutorials.

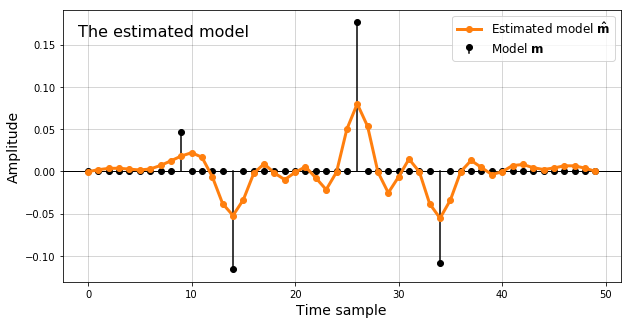
# NOT FOR MANUSCRIPT  
fig, ax = plt.subplots(figsize=(10, 5))  
  
#plt.stem(m, label=r'Model $\mathbf{m}$', basefmt='none', linefmt='k', markerfmt='ko')  
#plt.axhline(color='k', lw=1)  
for i, m\_est in enumerate(cache):  
 plt.plot(m\_est, color='C'+str(6-i), label='iter {}'.format(i))  
plt.legend(fontsize=12)  
plt.grid(color='k', alpha=0.15)  
ax.text(-1, 0.07, 'Conjugate gradient estimates', size=14)  
ax.text(-1, 0.06, '{} iterations'.format(n), size=12)  
ax.set\_xlabel('Time sample', size=14)  
ax.set\_ylabel('Amplitude', size=14)  
  
plt.savefig('figure2\_cg\_est\_iter.png', dpi=300)  
plt.show()



png

**Figure 2** Model estimated by the first five iterations of conjugate gradient plotted with the original model. Four iterations can be seen and the fifth nearly overlays the fourth. Mathematically, convergence is ensured in 50 iterations (the dimension of the model), but it hardly changes after the fourth.

# NOT FOR MANUSCRIPT  
fig, ax = plt.subplots(figsize=(10, 5))  
  
plt.stem(m, label='Model $\mathbf{m}$', basefmt='none', linefmt='k', markerfmt='ko')  
plt.axhline(color='k', lw=1)  
ax.plot(m\_est, 'o-', color='C1', lw=3, label='Estimated model $\mathbf{\hat{m}}$')  
ax.legend(fontsize=12)  
ax.set\_xlabel('Time sample', size=14)  
ax.set\_ylabel('Amplitude', size=14)  
ax.text(-1, 0.16, 'The estimated model', color='k', size=16)  
ax.grid(color='k', alpha=0.2)  
  
plt.savefig('figure3\_data\_model\_estmodel.png', dpi=300)  
plt.show()



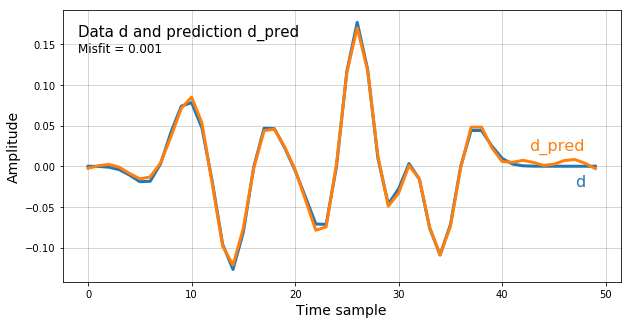
png

**Figure 3**: Comparison of the model (black) and the model estimated using conjugate gradient inversion (orange).

# NOT FOR MANUSCRIPT  
  
# Check the norm  
difference = d - d\_pred  
misfit = np.linalg.norm(difference)\*\*2  
misfit

0.00073174262505606676

# NOT FOR MANUSCRIPT  
fig, ax = plt.subplots(figsize=(10, 5))  
  
ax.plot(d, label='d', color='C0', lw=3)  
ax.plot(d\_pred, label="d\_pred", color='C1', lw=3)  
ax.text(-1, 0.16, 'Data d and prediction d\_pred', color='k', size=15)  
ax.text(-1, 0.14, 'Misfit = {:.3f}'.format(misfit), color='k', size=12)  
ax.text(48, -0.025, 'd', ha='right', color='C0', size=16)  
ax.text(48, 0.02, 'd\_pred', ha='right', color='C1', size=16)  
ax.set\_xlabel('Time sample', size=14)  
ax.set\_ylabel('Amplitude', size=14)  
ax.grid(color='k', alpha=0.2)  
  
plt.savefig('figure4\_data\_cg\_pred.png', dpi=300)  
plt.show()



png

**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’s *Linear inversion* tutorial 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 August 9 to 13 in Houston inspired this tutorial. For more information visit: http://ahay.org/wiki/Houston\_2017

## References

Claerbout, J., Fomel, S., 2012, Image Estimation by Example, http://sepwww.stanford.edu/sep/prof/gee1-2012.pdf

Guo, J., Hongbo, Z., Young, J., Gray, S., 2002, Merits and challenges for accurate velocity model building by 3D gridded tomography, Geophysics. https://doi.org/10.1190/1.1817395

Hall, M., 2016, Linear Inversion, The Leading Edge, pages 1085-1087. https://doi.org/10.1190/tle35121085.1

Hestenes, Magnus R.; Stiefel, Eduard, 1952, “Methods of Conjugate Gradients for Solving Linear Systems”. Journal of Research of the National Bureau of Standards. 49 (6). doi:10.6028/jres.049.044

Paige, C. C., Saunders, M. A., 1982, LSQR: An Algorithm for Sparse Linear Equations and Sparse Least Squares, ACM Transactions on Mathematical Software, http://web.stanford.edu/class/cme324/paige-saunders2.pdf

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

Witte, P, Mathias Louboutin, Keegan Lensink, Michael Lange, Navjot Kukreja, Fabio Luporini, Gerard Gorman, Felix J. Herrman, 2018, Full-waveform inversion, Part 3: Optimization. The Leading Edge 37, 2(2018); pp. 142-145. https://doi.org/10.1190/tle37020142.1