Geophysical tutorial: December 2016

Linear inversion

Matt Hall, matt@agilegeoscience.com

As a student geologist, I was never inducted in the world of linear algebra. Later, as a professional, I remained happily ignorant of Hessian matrices and Hermitian adjoints. But ever since reading Brian Russell's *Don't neglect your math* essay (Russell 2012), I've wanted to put things right. In particular, I have wanted to understand the well-known geophysical equation at the heart of every inversion: **d** = **Gm**. It's only an equation, how hard can it be?

In this tutorial, we're going to examine **d** = **Gm** for a familiar geophysical case: a 1D reflectivity series convolved with a wavelet, the recipe that produces a synthetic seismogram. The so-called forward problem is the convolution that produces the trace; the inverse problem is the deconvolution that recovers the reflectivity from the trace. We'll apply linear algebra to solve both of these problems.

For me, understanding comes through application in code, not through reading equations on paper. Usually in these columns we stick to Python, but today we're also going to peek at three other languages: Julia, R, and Lua. Not just to see how they differ, but also to marvel at their similarities. Find the Jupyter Notebooks accompanying this tutorial at http://github.com/seg.

Defining the problem

Now to define the components of **d** = **Gm**, starting with the data **d**. In our toy problem it's a single synthetic seismic trace with N = 25 samples. We'll assume for now that the data points are accurate: there's no noise. Its symbol is bold because it is a vector. You can think of it as a list of numbers: seismic amplitudes in this case. We'll leave **d** there for now, until we're ready to create it.

The model **m** is continuous in principle (i.e. in the earth) but discrete in practice (in the computer). We're going to think of it as a single reflectivity log with M = 50 points, a series of reflection coefficients.

We could define the reflectivities directly, but I find it more intuitive to think about the interval property acoustic impedance `imp`, then compute the interface property from that. To make a blocky model from values of P-wave velocity and bulk density:

```
import numpy as np
imp = np.ones(51) * 2550 * 2650
imp[10:15] =        2700 * 2750
imp[15:27] =        2400 * 2450
imp[27:35] =        2800 * 3000
```

Then the reflectivity model is given by:

```
m = (imp[1:] - imp[:-1]) / (imp[1:] + imp[:-1])
```

Now we move to less familiar territory: the so-called *forward operator* **G**. In general terms we can say that **G** is a linear operator that represents the experiment we're simulating. 'Linear' because it 'operates' in the same way on every sample: given inputs result in the same outputs every time. It's somewhat analogous to a kernel in a convolution — think about a running mean window smoothing a wireline log — with the difference that it also represents the way the kernel moves across the input.

Let's get specific. In our case, **G** represents convolution with a wavelet combined with a downsampling. It is an $N \times M$ matrix with one column for each sample in the input model **m**, and one row per sample in the output data **d**. The rows are shifted versions of the wavelet, as I've tried to show in Figure 1.

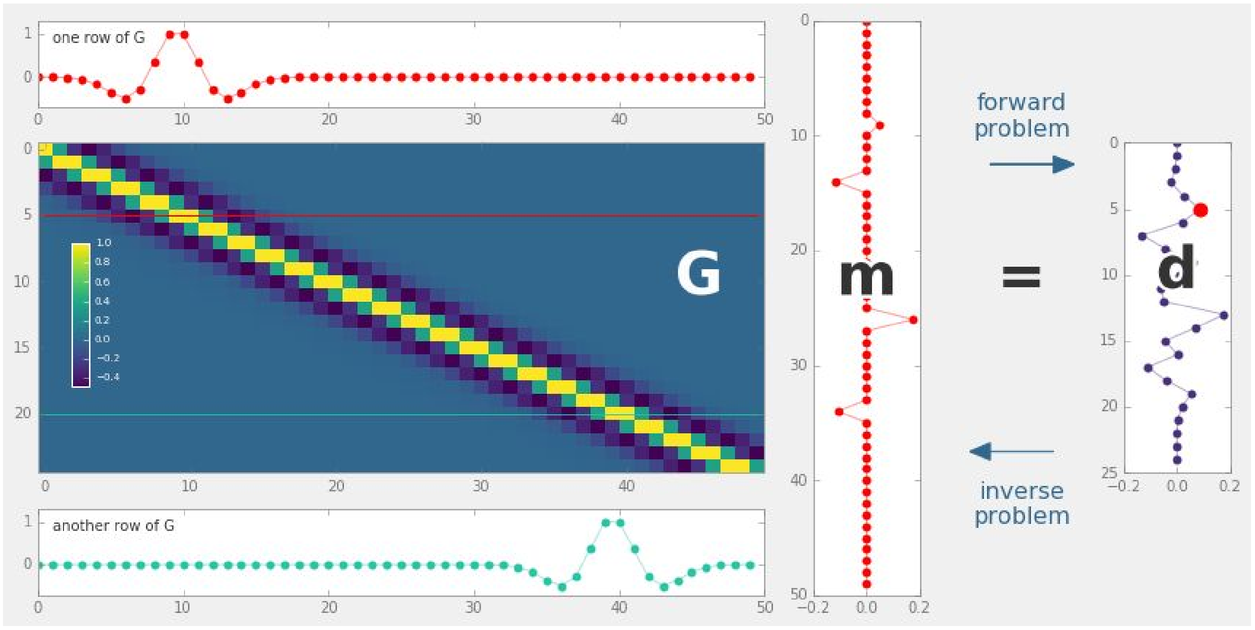


Figure 1. *The dot product of a linear operator* ***G*** *and a reflectivity model* ***m*** *results in synthetic seismic data* ***d****. In the dot product, each data point is the sum of the products of the elements of a row of* ***G*** *with the model* ***m****. In our problem, each row of* ***G*** *is a shifted version of the wavelet.*

We have what we need to make **G**. First we make a Ricker wavelet:

```
from scipy.signal import ricker
wavelet = ricker(points=20, a=2)
```

Now we use a helper function `convmtx` borrowed from MATLAB (see the notebook for details), to construct the 'convolution matrix' for the wavelet:

```
G = convmtx(wavelet, m.size)[::2, 10:60]
```

The complicated-looking slicing at the end is just to control the shape of the resulting data.

We're done! For a higher-dimensional problem or more complicated physics, **G** will be commensurately more involved. But for now this simple case seems like enough to worry about. We're ready to solve the forward problem to compute **d**.

The forward problem

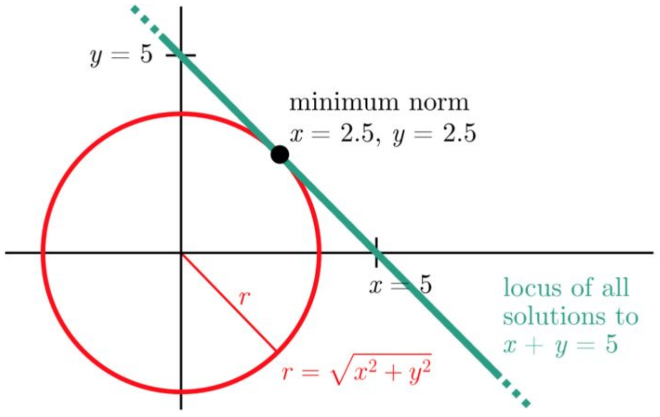
Now that we have **G** and **m**, we can use NumPy's new dot product operator `@` to produce **d**:

```
d = G @ m
```

When we calculate the dot product **Gm** we are just stepping over the rows doing elementwise multiplication of the samples of the wavelet with those of the reflectivity, and summing them. As I highlight in Figure 1, one row, combined in this way with the model, gives one data point. This process describes convolution, which I normally think of as the sum of a lot of wavelets, each scaled by a reflection coefficient. It's the same thing.

You can also think about Figure 1 as a system of equations. There are 25 linear equations, each with 50 variables (the values of the model **m**) and one answer (the value of the data **d**). Notice that there are more variables than equations. This means the system is *underdetermined* and has an infinite number of possible solutions (see box below).

Box: UNDERDETERMINED PROBLEMS



Geophysical inverse problems are usually underdetermined. A simple underdetermined problem is $x + y = 5$. There are infinitely many choices of x and y that satisfy the equation, so how shall we choose the best one? Clearly we can only do it with some prior knowledge: maybe we know that certain values are more likely, or we could require whole positive numbers only, but these still don't offer unique solutions. The criterion we will use for 'best' is 'closest to zero'. Visualize a circle growing from the origin: the first place it touches the line $x = -y + 5$ is the closest and, by our definition, best solution. But realize that there are lots of ways of defining 'best' — they correspond to choosing different types of distance or *norm*.

The estimation problem

Now we have to imagine that all we have is the data **d** and some idea of how the physical system works, represented by **G**. We would like to solve for **m**, the reflectivity, in what is called the inverse problem.

Anyone with a bit of algebra can see that if $\mathbf{d} = \mathbf{Gm}$ then $\mathbf{m} = \mathbf{G}^{-1}\mathbf{d}$ — although in linear algebra that $^{-1}$ means *inverse*, and is not the same thing as the reciprocal for scalars. Still, the notation works, and if we can find the inverse of **G**, we have a solution! Unfortunately, non-square matrices do not have an inverse, and for various other reasons we can never use this approach in geophysical problems.

It turns out that many non-invertible (also known as *singular* or *degenerate*) matrices do have a *pseudo*- or *generalized inverse* $\mathbf{G}^{+}$ (or sometimes $\mathbf{G}^{g}$ or $\mathbf{G}^{-g}$ for 'generalized'), which does the same job. Our matrix is *fat* — it has more columns than rows — and the rows are independent, so we can use the so-called *right inverse*, given by $\mathbf{G}^{\mathrm{T}}(\mathbf{G}\mathbf{G}^{\mathrm{T}})^{-1}$ — the inverse is on the right. When we use this instead of $\mathbf{G}^{-1}$ in the inverse equation above, the code to compute the estimated model `m_est` is:

```
m_est = G.T @ np.linalg.inv(G @ G.T) @ d
```

This results in a so-called *minimum norm* estimate for the model, denoted $\hat{\mathbf{m}}$ (the hat means 'estimate'). It is a *least squares* solution, meaning it minimizes $||\mathbf{G}\hat{\mathbf{m}} - \mathbf{d}||_2$, but the system is underdetermined and there are lots of least squares solutions that fit the data (see box). This is the one with the minimum norm, or (in Euclidean terms) length. In linear algebra jargon, we say that $\hat{\mathbf{m}}$ is the solution to the following *optimization* problem:

$$\begin{array}{ll} \text{minimize} & ||\hat{\mathbf{m}}||_2 \\ \text{subject to} & \mathbf{G}\hat{\mathbf{m}} = \mathbf{d} \end{array}$$

The notation $||\mathbf{x}||_2$ means 'the L2 norm of $\mathbf{x}$', where the L2 norm (sometimes written ℓ_2 norm and often just called 'the norm') is just a mathematical way of saying length. We'll look at norms in a future tutorial.

The appraisal problem

We'd like to know how well we did. In our synthetic situation, we can check how good the estimate $\hat{\mathbf{m}}$ is by comparing it to the known model $\mathbf{m}$, but this is not an option in a real inversion problem, where the whole point is that we are trying to find the model.

We already constrained the model by choosing the solution with the smallest norm. We can ensure that it meets other criteria we may have, drawing on our prior knowledge. For example, we might want the result to 'look geological' in some specific way. But this must be carefully formulated, or we risk begging the question.

Instead we must focus on the data. We use `m_est` in a forward model, and check if the data it generates looks like the 'real' data we started with (our data isn't real, but we made it deterministically from the real model). The check we use is a *misfit function*, which is often chosen to be the squared Euclidean distance between `d` and `d_pred`:

```
difference = d - d_pred
misfit = np.linalg.norm(difference)**2
misfit = difference.T @ difference  # Equivalent.
```

The comparison is shown in Figure 2. In this very simple, noise-free case, we find that the misfit is almost zero: the minimum norm estimate almost perfectly reproduces the data. This is expected with perfect data; the real test comes when we add noise.

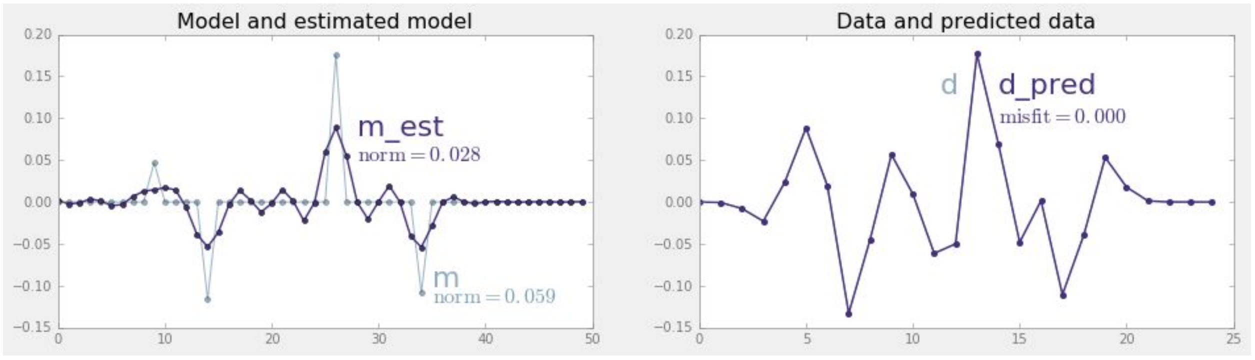
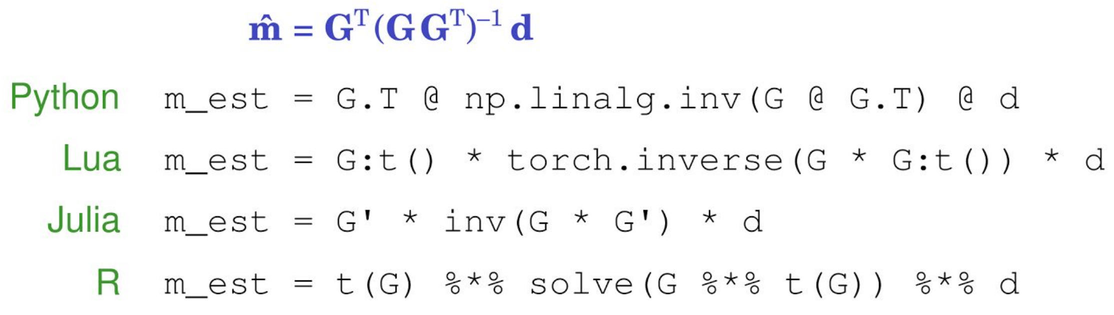


Figure 2. (a) The known model `m` *with the estimated model* `m_est` *and (b) the forward-modeled data* `d` *(obscured) with the predicted data* `d_pred`*. The norms of the models and the misfit of the predicted data are given.*

Multilingual least squares

These operations are so fundamental that they are implemented in every linear algebra package, regardless of the language. In fact they all really just implement functions in LAPACK, the ubiquitous Fortran 90 linear algebra library.

Let's look at how we can express the minimum norm least squares estimate in four different scientific computing languages:



$$\hat{\mathbf{m}} = \mathbf{G}^{\mathrm{T}}(\mathbf{G}\,\mathbf{G}^{\mathrm{T}})^{-1}\,\mathbf{d}$$

Python	`m_est = G.T @ np.linalg.inv(G @ G.T) @ d`
Lua	`m_est = G:t() * torch.inverse(G * G:t()) * d`
Julia	`m_est = G' * inv(G * G') * d`
R	`m_est = t(G) %*% solve(G %*% t(G)) %*% d`

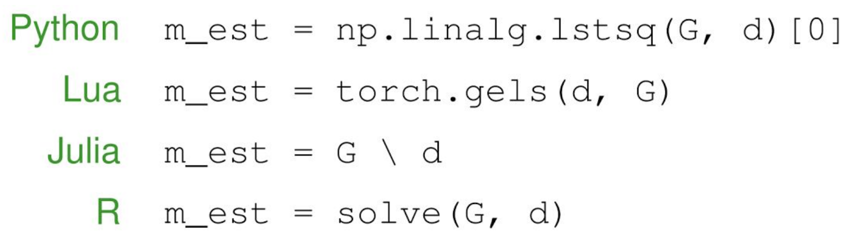
```
m_est = G.T @ np.linalg.inv(G @ G.T) @ d
m_est = G:t() * torch.inverse(G * G:t()) * d
m_est = G' * inv(G * G') * d
m_est = t(G) %*% solve(G %*% t(G)) %*% d
```

The differences are just as interesting as the similarities. I'm used to Python so I find the Lua version fairly intuitive. The mathematically inclined often seem to like Julia, and you can see why. I find R quite unintuitive.

What next?

There are lots of things we can do from here. For one thing, computing the inverse of a matrix is usually a bad idea. There are more computationally efficient and accurate methods, for example using matrix factorization. The accompanying notebooks look at some of these approaches.

We can go further. Most languages implement solvers that try to take the most efficient approach, given the shape and internal character of the matrix **G**. For example, returning to the languages we surveyed before, the surest path to the minimum norm least squares solution is:



Python	`m_est = np.linalg.lstsq(G, d)[0]`
Lua	`m_est = torch.gels(d, G)`
Julia	`m_est = G \ d`
R	`m_est = solve(G, d)`

```
m_est = np.linalg.lstsq(G, d)[0]
m_est = torch.gels(d, G)
m_est = G \ d
m_est = solve(G, d)
```

We can go further still. We can make the problem more realistic, for example by adding noise. In this case the problem is fundamentally the same, but we allow for some tolerance in the exactness solutions. This introduces the concept of regularization, which we'll look at in a future tutorial.

References and acknowledgments

Russell, B (2013). Don't neglect your math. In: M Hall and E Bianco (eds.), *52 Things You Should Know About Geophysics*, Agile Libre. Online at http://ageo.co/Russell2013

This tutorial was inspired by Mauricio Sacchi's *Notes on linear inverse theory, minimum norm solutions, and quadratic regularization with MATLAB examples.* Online at http://ageo.co/GEOPH431

To find out more, I strongly Doug Oldenburg and Francis Jones's online course notes Inversion for Applied Geophysics: Learning resources about geophysical inversion. Online at http://ageo.co/ageo.co/IAG2007

Heartfelt thanks to Graham Ganssle, Leo Uieda, Rowan Cockett, David Holmes, Ben Bougher, Diego Castañeda, Victor Aarre, and Evan Bianco for the many helpful conversations leading to my enlightenment.