Model fitting by least squares

Jon Claerbout

The first level of computer use in science and engineering is **modeling**. Beginning from physical principles and design ideas, the computer mimics nature. Then the worker looks at the result, thinks a while, alters the modeling program, and tries again. The next, deeper level of computer use is that the computer examines the results of modeling and reruns the modeling job. This deeper level is variously called "**fitting**," "**estimation**," or "**inversion**." We inspect the **conjugate-direction method** of fitting and write a subroutine for it that is used in most of the examples in this book.

UNIVARIATE LEAST SQUARES

A single parameter fitting problem arises in Fourier analysis, where we seek a "best answer" at each frequency, then combine all the frequencies to get a best signal. Thus, emerges a wide family of interesting and useful applications. However, Fourier analysis first requires us to introduce complex numbers into statistical estimation.

Multiplication in the Fourier domain is **convolution** in the time domain. Fourier-domain division is time-domain **deconvolution**. This division is challenging when F has observational error. Failure erupts if zero division occurs. More insidious are the poor results we obtain when zero division is avoided by a near miss.

Dividing by zero smoothly

Think of any real numbers x, y, and f and any program containing $x = y/f$. How can we change the program so that it never divides by zero? A popular answer is to change $x = y/f$ to $x = yf/(f^2 + \epsilon^2)$, where ϵ is any tiny value. When $|f| >> |\epsilon|$, then x is approximately y/f as expected. But when the divisor f vanishes, the result is safely zero instead of infinity. The transition is smooth, but some criterion is needed to choose the value of ϵ. This method may not be the only way or the best way to cope with **zero division**, but it is a good method, and permeates the subject of signal analysis.

To apply this method in the Fourier domain, suppose that X, Y, and F are complex numbers. What do we do then with $X = Y/F$? We multiply the top and bottom by the complex conjugate $\overline{F}$, and again add ϵ^2 to the denominator. Thus,

$$X(\omega) \quad = \quad \frac{\overline{F(\omega)}\; Y(\omega)}{\overline{F(\omega)} F(\omega) \;+\; \epsilon^2} \tag{1}$$

Now, the denominator must always be a positive number greater than zero, so division is always safe. Equation (1) ranges continuously from **inverse filter**ing, with $X = Y/F$, to filtering with $X = \overline{F}Y$, which is called "**matched filter**ing." Notice that for any complex number F, the phase of $1/F$ equals the phase of $\overline{F}$, so the filters have the same phase.

Damped solution

Another way to say $x = y/f$ is to say $fx - y$ is small, or $(fx - y)^2$ is small. This does not solve the problem of f going to zero, so we need the idea that x^2 does not get too big. To find x, we minimize the quadratic function in x.

$$Q(x) \quad = \quad (fx - y)^2 + \epsilon^2 x^2 \tag{2}$$

The second term is called a "**damping** factor," because it prevents x from going to $\pm\infty$ when $f \to 0$. Set $dQ/dx = 0$, which gives:

$$0 \quad = \quad f(fx - y) + \epsilon^2 x \tag{3}$$

Equation (3) yields our earlier common-sense guess $x = fy/(f^2 + \epsilon^2)$. It also leads us to wider areas of application in which the elements are complex vectors and matrices.

With Fourier transforms, the signal X is a complex number at each frequency ω. Therefore we generalize equation (2) to:

$$Q(\bar{X}, X) \quad = \quad (\overline{FX - Y})(FX - Y) + \epsilon^2 \bar{X} X \quad = \quad (\bar{X}\bar{F} - \bar{Y})(FX - Y) + \epsilon^2 \bar{X} X \tag{4}$$

To minimize Q, we could use a real-values approach, where we express $X = u + iv$ in terms of two real values u and v, and then set $\partial Q/\partial u = 0$ and $\partial Q/\partial v = 0$. The approach we take, however, is to use complex values, where we set $\partial Q/\partial X = 0$ and $\partial Q/\partial \bar{X} = 0$. Let us examine $\partial Q/\partial \bar{X}$:

$$\frac{\partial Q(\bar{X}, X)}{\partial \bar{X}} \quad = \quad \bar{F}(FX - Y) + \epsilon^2 X \quad = \quad 0 \tag{5}$$

The derivative $\partial Q/\partial X$ is the complex conjugate of $\partial Q/\partial \bar{X}$. Therefore, if either is zero, the other is also zero. Thus, we do not need to specify both $\partial Q/\partial X = 0$ and $\partial Q/\partial \bar{X} = 0$. I usually set $\partial Q/\partial \bar{X}$ equal to zero. Solving equation (5) for X gives equation (1).

Equation (1) solves $Y = XF$ for X, giving the solution for what is called "the **deconvolution** problem with a known wavelet F." Analogously, we can use $Y = XF$ when the filter F is unknown, but the input X and output Y are given. Simply interchange X and F in the derivation and result.

Smoothing the denominator spectrum

Equation (1) gives us one way to divide by zero. Another way is stated by the equation

$$X(\omega) = \frac{\overline{F}(\omega) Y(\omega)}{\left\langle \overline{F}(\omega) F(\omega) \right\rangle} \tag{6}$$

where the strange notation in the denominator means that the spectrum there should be smoothed a little. Such smoothing fills in the holes in the spectrum where zero-division is a danger, filling not with an arbitrary numerical value ϵ but with an average of nearby spectral values. Additionally, if the denominator spectrum $\overline{F}(\omega)F(\omega)$ is rough, the smoothing creates a shorter autocorrelation function.

Both divisions, equation (1) and equation (6), irritate us by requiring us to specify a parameter, but for the latter, the parameter has a clear meaning. In the latter case we

smooth a spectrum with a smoothing window of width, say $\Delta\omega$ which this corresponds inversely to a time interval over which we smooth. Choosing a numerical value for ϵ has not such a simple interpretation.

We jump from simple mathematical theorizing towards a genuine practical application when I grab some real data, a function of time and space from another textbook. Let us call this data $f(t,x)$ and its 2-D Fourier transform $F(\omega,k_x)$. The data and its autocorrelation are in Figure 1.

The autocorrelation $a(t,x)$ of $f(t,x)$ is the inverse 2-D Fourier Transform of $\overline{F}(\omega,k_x)F(\omega,k_x)$. Autocorrelations $a(x,y)$ satisfy the symmetry relation $a(x,y)=a(-x,-y)$. Figure 2 shows only the interesting quadrant of the two independent quadrants. We see the autocorrelation of a 2-D function has some resemblance to the function itself but differs in important ways.

Instead of messing with two different functions X and Y to divide, let us divide F by itself. This sounds like $1=F/F$ but we will watch what happens when we do the division carefully avoiding zero division in the ways we usually do.

Figure 2 shows what happens with

$$1 = F/F \quad \approx \quad \frac{\overline{F}F}{\overline{F}F+\epsilon^2} \tag{7}$$

and with

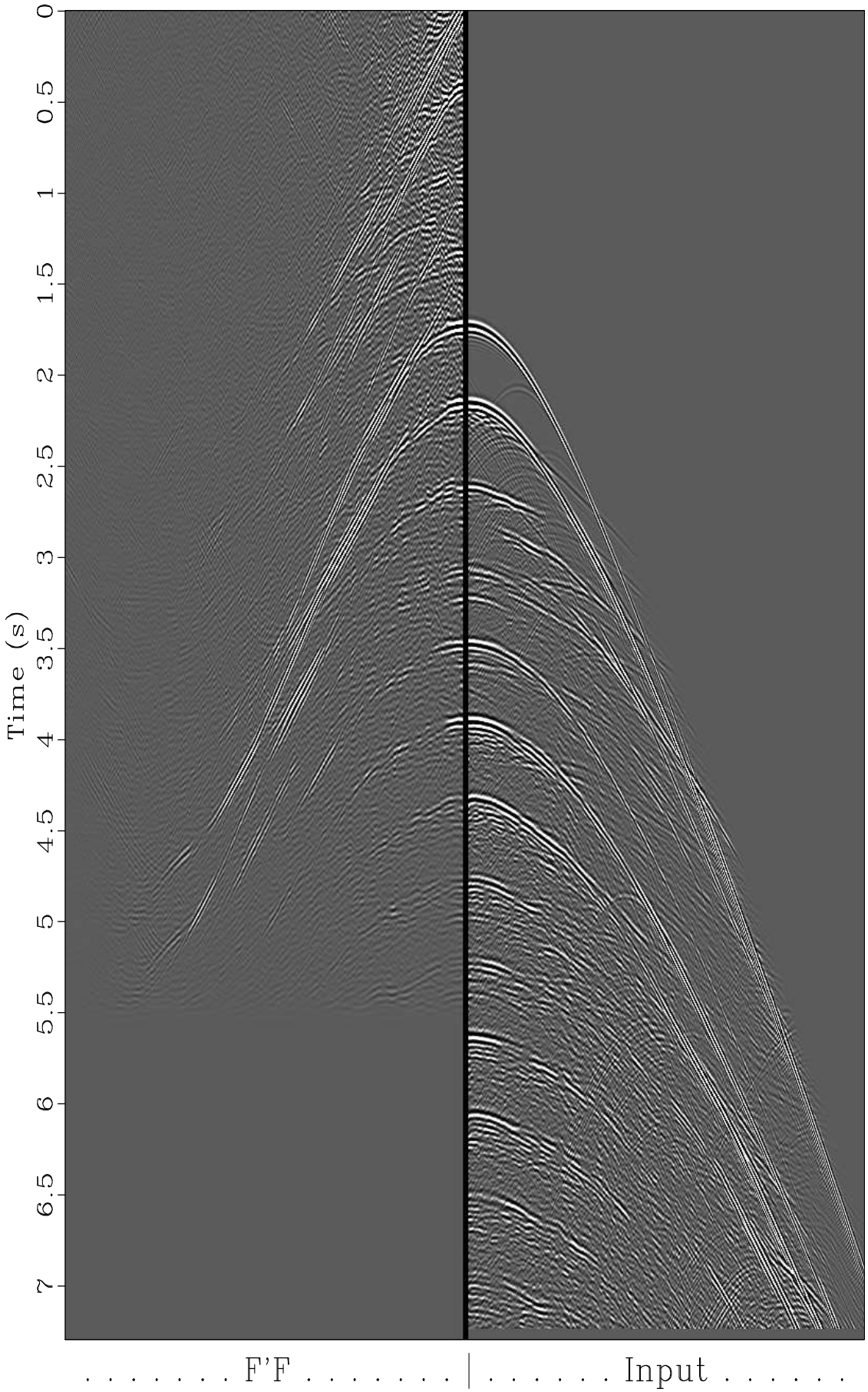
$$1 = F/F \quad \approx \quad \frac{\overline{F}F}{\left\langle \overline{F}F \right\rangle} \tag{8}$$

From Figure 2 we notice that both methods of avoiding zero division give similar results. By playing with the ϵ and the smoothing width the pictures could be made even more similar. My preference, however, is the smoothing. It is difficult to make physical sense of choosing a numerical value for ϵ. It is much easier to make physical sense of choosing a smoothing window. The smoothing window is in (ω,k_x) space, but Fourier transformation tells us its effect in (t,x) space.

Imaging

The example of dividing a function by itself ($1=F/F$) might not seem to make much sense, but it is very closely related to estimation often encountered in imaging applications. It's not my purpose here to give a lecture on imaging theory, but here is an over-brief explanation.

Imagine a downgoing wavefield $D(\omega,x,z)$. Propagating against irregularities in the medium $D(\omega,x,z)$ creates by scattering an upgoing wavefield $U(\omega,x,z)$. Given U and D, if there is a strong temporal correlation between them at any (x,z) it likely means there is a reflector nearby that is manufacturing U from D. This reflectivity could be quantified by U/D. At the Earth's surface the surface boundary condition says something like $U=D$ or $U=-D$. Thus at the surface we have something like F/F. As we go down in the Earth, the main difference is that U and D get time-shifted in opposite directions, so U and D are similar but for that time difference. Thus, a study of how we handle F/F is worthwhile.


Time (s)
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
5.5
6
6.5
7
. . . . . F'F . . . . . | . . . . . Input . . . . .

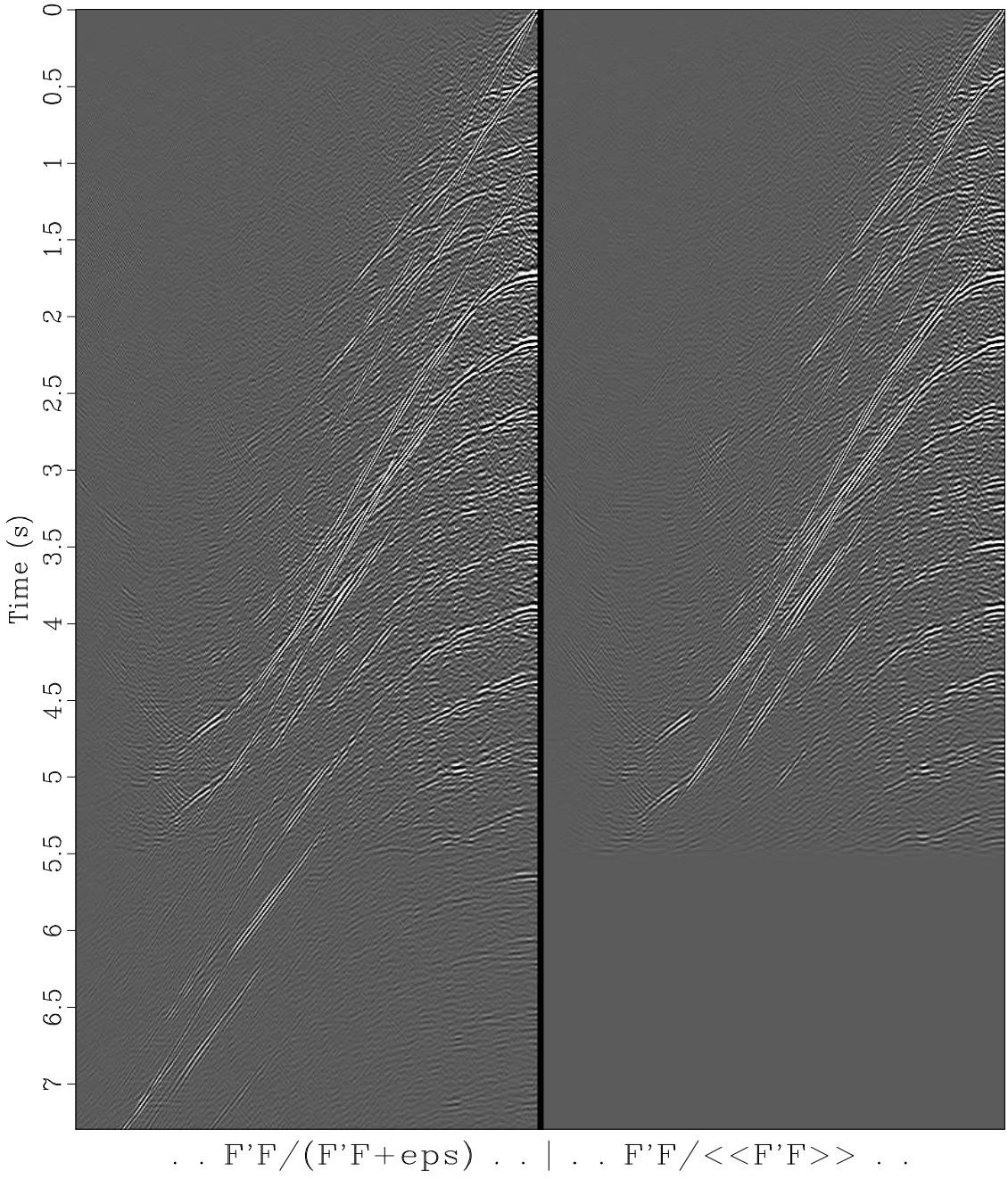


Figure 2: Equation 7 (left) and equation 8 (right). Both ways of dividing by zero give similar results.

Formal path to the low-cut filter

This book defines many geophysical estimation applications. Many applications amount to fitting two goals. The first goal is a data-fitting goal, the goal that the model should imply some observed data. The second goal is that the model be not too big nor too wiggly. We state these goals as two residuals, each of which is ideally zero. A very simple data fitting goal would be that the model m equals the data d, thus the difference should vanish, say $0 \approx m - d$. A more interesting goal is that the model should match the data especially at high frequencies but not necessarily at low frequencies.

$$0 \quad \approx \quad -i\omega(m-d) \tag{9}$$

A danger of this goal is that the model could have a zero-frequency component of infinite magnitude as well as large amplitudes for low frequencies. To suppress such bad behavior we need the second goal, a model residual to be minimized. We need a small number ϵ. The model goal is:

$$0 \quad \approx \quad \epsilon\, m \tag{10}$$

To see the consequence of these two goals, we add the squares of the residuals:

$$Q(m) \quad = \quad \omega^2(m-d)^2 + \epsilon^2 m^2 \tag{11}$$

and then, we minimize $Q(m)$ by setting its derivative to zero:

$$0 \quad = \quad \frac{dQ}{dm} \quad = \quad 2\omega^2(m-d) + 2\epsilon^2 m \tag{12}$$

or

$$m \quad = \quad \frac{\omega^2}{\omega^2+\epsilon^2}\, d \tag{13}$$

Let us rename ϵ to give it physical units of frequency $\omega_0 = \epsilon$. Our expression says says m matches d except for low frequencies $|m| < |\omega_0|$ where it tends to zero. Now we recognize we have a low-cut filter with "cut-off frequency" ω_0.

The plane-wave destructor

We address the question of shifting signals into best alignment. The most natural approach might seem to be via cross correlations, which is indeed a good approach when signals are shifted by large amounts. Here, we assume signals are shifted by small amounts, often less than a single pixel. We take an approach closely related to differential equations. Consider this definition of a residual.

$$0 \quad \approx \quad \text{residual}(t,x) \quad = \quad \left(\frac{\partial}{\partial x} + p\frac{\partial}{\partial t}\right) u(t,x) \tag{14}$$

By taking derivatives we see the residual vanishes when the two-dimensional observation $u(t,x)$ matches the equation of moving waves $u(t-px)$. The parameter p has units inverse to velocity, the velocity of propagation.

In practice, $u(t,x)$ might not be a perfect wave but an observed field of many waves that we might wish to fit to the idea of a single wave of a single p. We seek the parameter p.

First, we need a method of discretization that allows the mesh for $\partial u/\partial t$ to overlay exactly $\partial u/\partial x$. To this end, I chose to represent the t-derivative by averaging a finite difference at x with one at $x + \Delta x$.

$$\frac{\partial u}{\partial t} \quad \approx \quad \frac{1}{2}\left(\frac{u(t+\Delta t, x) - u(t,x)}{\Delta t}\right) + \frac{1}{2}\left(\frac{u(t+\Delta t, x+\Delta x) - u(t, x+\Delta x)}{\Delta t}\right) \tag{15}$$

Likewise, there is an analogous expression for the x-derivative with t and x interchanged. The function $u(t, x)$ lies on a grid, and the differencing operator $\delta_x + p\delta_t$ lies atop it and convolves across it. The operator is a 2×2 convolution filter. We may represent equation (14) as a matrix operation,

$$\mathbf{0} \quad \approx \quad \mathbf{r} = \mathbf{A}\mathbf{u} \tag{16}$$

where the two-dimensional convolution with the difference operator is denoted $\mathbf{A}$.

Now, let us find the numerical value of p that fits a plane wave $u(t - px)$ to observations $u(t, x)$. Let $\mathbf{x}$ be an abstract vector having components with values $\partial u/\partial x$ taken everywhere on a 2-D mesh in (t, x). Likewise, let $\mathbf{t}$ contain $\partial u/\partial t$. Because we want $\mathbf{x} + p\mathbf{t} \approx \mathbf{0}$, we minimize the quadratic function of p,

$$Q(p) = (\mathbf{x} + p\mathbf{t}) \cdot (\mathbf{x} + p\mathbf{t}) \tag{17}$$

by setting to zero the derivative by p. We get:

$$p \quad = \quad -\frac{\mathbf{x} \cdot \mathbf{t}}{\mathbf{t} \cdot \mathbf{t}} \tag{18}$$

Because data does not always fit the model very well, it may be helpful to have some way to measure how good the fit is. I suggest:

$$C^2 \quad = \quad 1 \; - \; \frac{(\mathbf{x} + p\mathbf{t}) \cdot (\mathbf{x} + p\mathbf{t})}{\mathbf{x} \cdot \mathbf{x}} \tag{19}$$

which, on inserting $p = -(\mathbf{x} \cdot \mathbf{t})/(\mathbf{t} \cdot \mathbf{t})$, leads to C , where

$$C \quad = \quad \frac{\mathbf{x} \cdot \mathbf{t}}{\sqrt{(\mathbf{x} \cdot \mathbf{x})(\mathbf{t} \cdot \mathbf{t})}} \tag{20}$$

is known as the "**normalized correlation**."

To suppress noise, the quadratic functions $\mathbf{x} \cdot \mathbf{x}$, $\mathbf{t} \cdot \mathbf{t}$, and $\mathbf{x} \cdot \mathbf{t}$ were smoothed over time with a triangle filter.

Subroutine `puck2d` shows the code that generated Figures 3–5. An example based on synthetic data is shown in Figures 3 through 5. The synthetic data in Figure 3 mimics a reflection seismic field profile, including one trace that is slightly delayed as if recorded on a patch of unconsolidated **soil**.

Figure 4 shows the **residual**. The residual is small in the central region of the data; it is large where the signal is not sampled densely enough, and it is large at the transient onset of the signal. The residual is rough because of the noise in the signal, because it is made from derivatives, and the synthetic data was made by nearest-neighbor interpolation. Notice that the residual is not particularly large for the delayed trace.

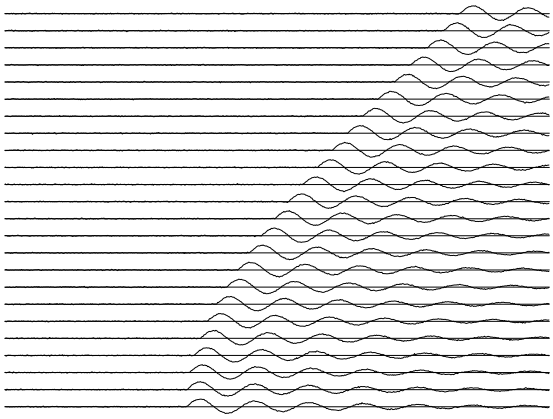


Figure 3: Input synthetic seismic data includes a low level of noise.

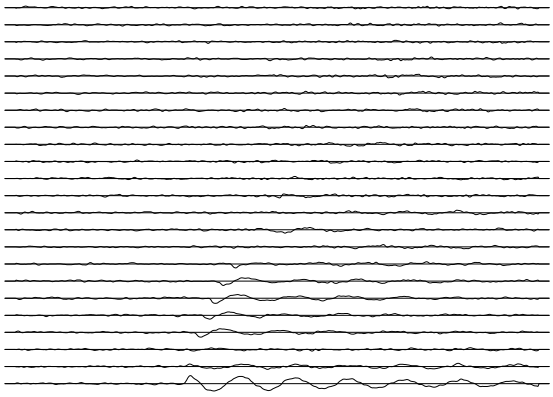


Figure 4: Residuals, i.e., an evaluation of $U_x + pU_t$.

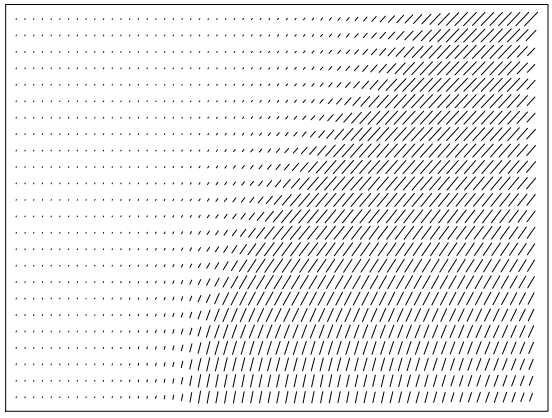


Figure 5: Output values of p are shown by the slope of short line segments.

Figure 5 shows the dips. The most significant feature of this figure is the sharp localization of the dips surrounding the delayed trace. Other methods based on "beam stacks" or Fourier concepts might lead us to conclude that the aperture must be large to resolve a wide range of angles. Here, we have a narrow aperture (two traces), but the dip can change rapidly and widely.

Once the stepout $p = dt/dx$ is known between each of the signals, it is a simple matter to integrate to get the total time shift. A real-life example is shown in Figure 6. In this

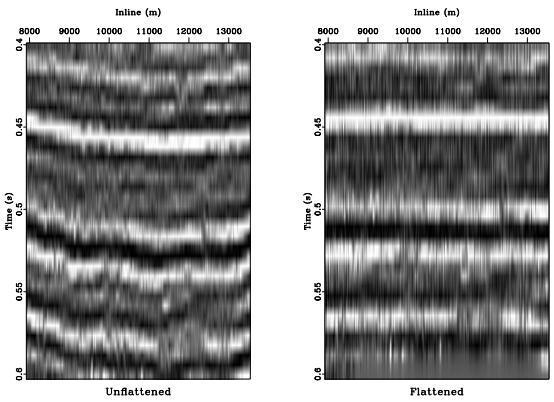


Figure 6: A seismic line before and after flattening.

case the flattening was a function of x only. More interesting (and more complicated) cases arise when the stepout $p = dt/dx$ is a function of both x and t. The code shown here should work well in such cases.

A disadvantage, well known to people who routinely work with finite-difference solutions to partial differential equations, is that for short wavelengths a finite difference operator is not the same as a differential operator; therefore, the numerical value of p is biased. This problem can be overcome in the following way. First, estimate the slope $p = dt/dx$ between each trace. Then, shift the traces to flatten arrivals. Now, there may be a residual p because of the bias in the initial estimate of p. This process can be iterated until the data is flattened. Everywhere in a plane we have solved a least squares problem for a single value p.

MULTIVARIATE LEAST SQUARES

Inside an abstract vector

In engineering uses, a vector has three scalar components that correspond to the three dimensions of the space in which we live. In least-squares data analysis, a vector is a one-dimensional array that can contain many different things. Such an array is an "**abstract vector**." For example, in **earthquake** studies, the vector might contain the time an earthquake began, as well as its latitude, longitude, and depth. Alternatively, the abstract vector might contain as many components as there are seismometers, and each component might be the arrival time of an earthquake wave. Used in signal analysis, the vector might contain the values of a signal at successive instants in time or, alternatively, a collection of signals. These signals might be "**multiplex**ed" (interlaced) or "demultiplexed" (all of each signal preceding the next). When used in image analysis, the one-dimensional array might contain an image, which is an array of signals. Vectors, including abstract vectors, are usually

denoted by **boldface letters** such as $\mathbf{p}$ and $\mathbf{s}$. Like physical vectors, abstract vectors are **orthogonal** if a dot product vanishes: $\mathbf{p} \cdot \mathbf{s} = 0$. Orthogonal vectors are well known in physical space; we also encounter orthogonal vectors in abstract vector space.

We consider first a hypothetical application with one data vector $\mathbf{d}$ and two fitting vectors $\mathbf{f}_1$ and $\mathbf{f}_2$. Each fitting vector is also known as a "**regressor**." Our first task is to approximate the data vector $\mathbf{d}$ by a scaled combination of the two regressor vectors. The scale factors m_1 and m_2 should be chosen so the model matches the data; i.e.,

$$\mathbf{d} \quad \approx \quad \mathbf{f}_1 m_1 + \mathbf{f}_2 m_2 \tag{21}$$

Notice that we could take the partial derivative of the data in (21) with respect to an unknown, say m_1, and the result is the regressor $\mathbf{f}_1$. The **partial derivative** of all modeled data d_i with respect to any particular model parameter m_j gives a **regressor**.

A **regressor** is a column in the matrix of partial-derivatives, $\partial d_i / \partial m_j$.

The fitting goal (21) is often expressed in the more compact mathematical matrix notation $\mathbf{d} \approx \mathbf{F}\mathbf{m}$, but in our derivation here, we keep track of each component explicitly and use mathematical matrix notation to summarize the final result. Fitting the observed data $\mathbf{d} = \mathbf{d}^{\text{obs}}$ to its two theoretical parts $\mathbf{f}_1 m_1$ and $\mathbf{f}_2 m_2$ can be expressed as minimizing the length of the residual vector $\mathbf{r}$, where:

$$\mathbf{0} \quad \approx \quad \mathbf{r} \quad = \quad \mathbf{d}^{\text{theor}} - \mathbf{d}^{\text{obs}} \tag{22}$$

$$\mathbf{0} \quad \approx \quad \mathbf{r} \quad = \quad \mathbf{f}_1 m_1 + \mathbf{f}_2 m_2 \; - \; \mathbf{d} \tag{23}$$

We use a dot product to construct a sum of squares (also called a "**quadratic form**") of the components of the residual vector:

$$Q(m_1, m_2) \quad = \quad \mathbf{r} \cdot \mathbf{r} \tag{24}$$

$$Q(m_1, m_2) \quad = \quad (\mathbf{f}_1 m_1 + \mathbf{f}_2 m_2 - \mathbf{d}) \cdot (\mathbf{f}_1 m_1 + \mathbf{f}_2 m_2 - \mathbf{d}) \tag{25}$$

To find the gradient of the quadratic form $Q(m_1, m_2)$, you might be tempted to expand out the dot product into all nine terms and then differentiate. It is less cluttered, however, to remember the product rule, that:

$$\frac{d}{dx} \mathbf{r} \cdot \mathbf{r} \quad = \quad \frac{d\mathbf{r}}{dx} \cdot \mathbf{r} + \mathbf{r} \cdot \frac{d\mathbf{r}}{dx} \tag{26}$$

Thus, the gradient of $Q(m_1, m_2)$ is defined by its two components:

$$\begin{array}{lcl} \frac{\partial Q}{\partial m_1} & = & \mathbf{f}_1 \cdot (\mathbf{f}_1 m_1 + \mathbf{f}_2 m_2 - \mathbf{d}) + (\mathbf{f}_1 m_1 + \mathbf{f}_2 m_2 - \mathbf{d}) \cdot \mathbf{f}_1 \\ \frac{\partial Q}{\partial m_2} & = & \mathbf{f}_2 \cdot (\mathbf{f}_1 m_1 + \mathbf{f}_2 m_2 - \mathbf{d}) + (\mathbf{f}_1 m_1 + \mathbf{f}_2 m_2 - \mathbf{d}) \cdot \mathbf{f}_2 \end{array} \tag{27}$$

Setting these derivatives to zero and using $(\mathbf{f}_1 \cdot \mathbf{f}_2) = (\mathbf{f}_2 \cdot \mathbf{f}_1)$ etc., we get

$$\begin{array}{lcl} (\mathbf{f}_1 \cdot \mathbf{d}) & = & (\mathbf{f}_1 \cdot \mathbf{f}_1) m_1 + (\mathbf{f}_1 \cdot \mathbf{f}_2) m_2 \\ (\mathbf{f}_2 \cdot \mathbf{d}) & = & (\mathbf{f}_2 \cdot \mathbf{f}_1) m_1 + (\mathbf{f}_2 \cdot \mathbf{f}_2) m_2 \end{array} \tag{28}$$

We can use these two equations to solve for the two unknowns m_1 and m_2. Writing this expression in matrix notation, we have:

$$\begin{bmatrix} (\mathbf{f}_1 \cdot \mathbf{d}) \\ (\mathbf{f}_2 \cdot \mathbf{d}) \end{bmatrix} \quad = \quad \begin{bmatrix} (\mathbf{f}_1 \cdot \mathbf{f}_1) & (\mathbf{f}_1 \cdot \mathbf{f}_2) \\ (\mathbf{f}_2 \cdot \mathbf{f}_1) & (\mathbf{f}_2 \cdot \mathbf{f}_2) \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \end{bmatrix} \tag{29}$$

It is customary to use matrix notation without dot products. For matrix notation we need some additional definitions. To clarify these definitions, we inspect vectors $\mathbf{f}_1$, $\mathbf{f}_2$, and $\mathbf{d}$ of three components. Thus,

$$\mathbf{F} \quad = \quad [\mathbf{f}_1 \quad \mathbf{f}_2] \quad = \quad \begin{bmatrix} f_{11} & f_{12} \\ f_{21} & f_{22} \\ f_{31} & f_{32} \end{bmatrix} \tag{30}$$

Likewise, the *transposed* matrix $\mathbf{F}^{\mathrm{T}}$ is defined by:

$$\mathbf{F}^{\mathrm{T}} \quad = \quad \begin{bmatrix} f_{11} & f_{21} & f_{31} \\ f_{12} & f_{22} & f_{32} \end{bmatrix} \tag{31}$$

Using this matrix $\mathbf{F}^{\mathrm{T}}$, there is a simple expression for the gradient calculated in equation (27). It is used in nearly every example in this book.

$$\mathbf{g} \quad = \quad \begin{bmatrix} \frac{\partial Q}{\partial m_1} \\ \frac{\partial Q}{\partial m_2} \end{bmatrix} \quad = \quad \begin{bmatrix} \mathbf{f}_1 \cdot \mathbf{r} \\ \mathbf{f}_2 \cdot \mathbf{r} \end{bmatrix} \quad = \quad \begin{bmatrix} f_{11} & f_{21} & f_{31} \\ f_{12} & f_{22} & f_{32} \end{bmatrix} \begin{bmatrix} r_1 \\ r_2 \\ r_3 \end{bmatrix} \quad = \quad \mathbf{F}^{\mathrm{T}} \mathbf{r} \tag{32}$$

In words this expression says, the gradient is found by putting the residual into the adjoint operator $\mathbf{g} = \mathbf{F}^{\mathrm{T}}\mathbf{r}$. Notice, the gradient $\mathbf{g}$ has the same number of components as the unknown solution $\mathbf{m}$, so we can think of the gradient as a $\Delta\mathbf{m}$, something we could add to $\mathbf{m}$ getting $\mathbf{m} + \Delta\mathbf{m}$. Later, we see how much of $\Delta\mathbf{m}$ we want to add to $\mathbf{m}$. We reach the best solution when we find the gradient $\mathbf{g} = \mathbf{0}$ vanishes, which happens as equation (32) says, when the residual is orthogonal to all the fitting functions (all the rows in the matrix $\mathbf{F}^{\mathrm{T}}$, the columns in $\mathbf{F}$, are perpendicular to $\mathbf{r}$).

The matrix in equation (29) contains dot products. Matrix multiplication is an abstract way of representing the dot products:

$$\begin{bmatrix} (\mathbf{f}_1 \cdot \mathbf{f}_1) & (\mathbf{f}_1 \cdot \mathbf{f}_2) \\ (\mathbf{f}_2 \cdot \mathbf{f}_1) & (\mathbf{f}_2 \cdot \mathbf{f}_2) \end{bmatrix} \quad = \quad \begin{bmatrix} f_{11} & f_{21} & f_{31} \\ f_{12} & f_{22} & f_{32} \end{bmatrix} \begin{bmatrix} f_{11} & f_{12} \\ f_{21} & f_{22} \\ f_{31} & f_{32} \end{bmatrix} \tag{33}$$

Thus, equation (29) without dot products is:

$$\begin{bmatrix} f_{11} & f_{21} & f_{31} \\ f_{12} & f_{22} & f_{32} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix} \quad = \quad \begin{bmatrix} f_{11} & f_{21} & f_{31} \\ f_{12} & f_{22} & f_{32} \end{bmatrix} \begin{bmatrix} f_{11} & f_{12} \\ f_{21} & f_{22} \\ f_{31} & f_{32} \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \end{bmatrix} \tag{34}$$

which has the matrix abbreviation:

$$\mathbf{F}^{\mathrm{T}} \mathbf{d} \quad = \quad (\mathbf{F}^{\mathrm{T}} \mathbf{F}) \mathbf{m} \tag{35}$$

Equation (35) is the classic result of least-squares fitting of data to a collection of regressors. Obviously, the same matrix form applies when there are more than two regressors and each vector has more than three components. Equation (35) leads to an analytic solution for $\mathbf{m}$ using an inverse matrix. To solve formally for the unknown $\mathbf{m}$, we premultiply by the inverse matrix $(\mathbf{F}^{\mathrm{T}}\,\mathbf{F})^{-1}$:

$$\mathbf{m} \quad = \quad (\mathbf{F}^{\mathrm{T}}\,\mathbf{F})^{-1}\,\mathbf{F}^{\mathrm{T}}\,\mathbf{d} \tag{36}$$

The central result of **least-squares** theory is $\mathbf{m} = (\mathbf{F}^{\mathrm{T}}\,\mathbf{F})^{-1}\,\mathbf{F}^{\mathrm{T}}\,\mathbf{d}$. We see it everywhere.

Let us examine all the second derivatives of $Q(m_1, m_2)$ defined by equation (25). Any multiplying $\mathbf{d}$ does not survive the second derivative, therefore, the terms we are left with are:

$$Q(m_1, m_2) = (\mathbf{f}_1 \cdot \mathbf{f}_1)m_1^2 + 2(\mathbf{f}_1 \cdot \mathbf{f}_2)m_1 m_2 + (\mathbf{f}_2 \cdot \mathbf{f}_2)m_2^2 \tag{37}$$

After taking the second derivative, we can organize all these terms in a matrix:

$$\frac{\partial^2 Q}{\partial m_i \partial m_j} \quad = \quad \begin{bmatrix} (\mathbf{f}_1 \cdot \mathbf{f}_1) & (\mathbf{f}_1 \cdot \mathbf{f}_2) \\ (\mathbf{f}_2 \cdot \mathbf{f}_1) & (\mathbf{f}_2 \cdot \mathbf{f}_2) \end{bmatrix} \tag{38}$$

Comparing equation (38) to equation (33) we conclude that $\mathbf{F}^{\mathrm{T}}\,\mathbf{F}$ is a matrix of second derivatives. This matrix is also known as the **Hessian**. It often plays an important role in small problems.

Larger problems tend to have insufficient computer memory for the Hessian matrix, because it is the size of model space squared. Where model space is a multidimensional Earth image, we have a large number of values even before squaring that number. Therefore, this book rarely works with the Hessian, working instead with gradients.

Rearrange parentheses representing (34).

$$\mathbf{F}^{\mathrm{T}}\,\mathbf{d} \quad = \quad \mathbf{F}^{\mathrm{T}}\,(\mathbf{F}\mathbf{m}) \tag{39}$$

Equation (35) led to the "analytic" solution (36). In a later section on conjugate directions, we see that equation (39) expresses better than equation (36) the philosophy of iterative methods.

Notice how equation (39) invites us to cancel the matrix $\mathbf{F}^{\mathrm{T}}$ from each side. We cannot do that of course, because $\mathbf{F}^{\mathrm{T}}$ is not a number, nor is it a square matrix with an inverse. If you really want to cancel the matrix $\mathbf{F}^{\mathrm{T}}$, you may, but the equation is then only an approximation that restates our original goal (21):

$$\mathbf{d} \quad \approx \quad \mathbf{F}\mathbf{m} \tag{40}$$

Speedy problem solvers might ignore the mathematics covering the previous page, study their application until they are able to write the statement of goals (40) = (21), premultiply by $\mathbf{F}^{\mathrm{T}}$, replace $\approx$ by $=$, getting (35), and take (35) to a simultaneous equation-solving program to get $\mathbf{m}$.

What I call "**fitting goals**" are called "**regressions**" by statisticians. In common language the word regression means to "trend toward a more primitive perfect state," which vaguely resembles reducing the size of (energy in) the residual $\mathbf{r} = \mathbf{F}\mathbf{m} - \mathbf{d}$. Formally, the fitting is often written as:

$$\min_{\mathbf{m}} \quad \|\mathbf{F}\mathbf{m} - \mathbf{d}\| \tag{41}$$

The notation with two pairs of vertical lines looks like double absolute value, but we can understand it as a reminder to square and sum all the components. This formal notation is more explicit about what is constant and what is variable during the fitting.

Normal equations

An important concept is that when energy is minimum, the residual is orthogonal to the fitting functions. The fitting functions are the column vectors $\mathbf{f}_1$, $\mathbf{f}_2$, and $\mathbf{f}_3$. Let us verify only that the dot product $\mathbf{r} \cdot \mathbf{f}_2$ vanishes; to do so, we show that those two vectors are orthogonal. Energy minimum is found by:

$$0 \quad = \quad \frac{\partial}{\partial m_2} \, \mathbf{r} \cdot \mathbf{r} \quad = \quad 2 \, \mathbf{r} \cdot \frac{\partial \mathbf{r}}{\partial m_2} \quad = \quad 2 \, \mathbf{r} \cdot \mathbf{f}_2 \tag{42}$$

(To compute the derivative, refer to equation (23).) Equation (42) shows that the residual is orthogonal to a fitting function. The fitting functions are the column vectors in the fitting matrix.

The basic least-squares equations are often called the "normal" equations. The word "normal" means perpendicular. We can rewrite equation (39) to emphasize the perpendicularity. Bring both terms to the right, and recall the definition of the residual $\mathbf{r}$ from equation (23):

$$\mathbf{0} \quad = \quad \mathbf{F}^{\mathrm{T}} \, (\mathbf{F}\mathbf{m} - \mathbf{d}) \tag{43}$$

$$\mathbf{0} \quad = \quad \mathbf{F}^{\mathrm{T}} \, \mathbf{r} \tag{44}$$

Equation (44) says that the **residual** vector $\mathbf{r}$ is perpendicular to each row in the $\mathbf{F}^{\mathrm{T}}$ matrix. These rows are the **fitting function**s. Therefore, the residual, after it has been minimized, is perpendicular to *all* the fitting functions.

Differentiation by a complex vector

Complex numbers frequently arise in physical applications, particularly those with Fourier series. Let us extend the multivariable least-squares theory to the use of complex-valued unknowns $\mathbf{m}$. First, recall how complex numbers were handled with single-variable least squares; i.e., as in the discussion leading up to equation (5). Use an asterisk, such as $\mathbf{m}^{\mathrm{T}}$, to denote the complex conjugate of the transposed vector $\mathbf{m}$. Now, write the positive **quadratic form** as:

$$Q(\mathbf{m}^{\mathrm{T}}, \mathbf{m}) \quad = \quad (\mathbf{F}\mathbf{m} - \mathbf{d})^{\mathrm{T}} \, (\mathbf{F}\mathbf{m} - \mathbf{d}) \quad = \quad (\mathbf{m}^{\mathrm{T}} \, \mathbf{F}^{\mathrm{T}} - \mathbf{d}^{\mathrm{T}})(\mathbf{F}\mathbf{m} - \mathbf{d}) \tag{45}$$

Recall from equation (4), where we minimized a quadratic form $Q(\bar{X}, X)$ by setting to zero, both $\partial Q / \partial \bar{X}$ and $\partial Q / \partial X$. We noted that only one of $\partial Q / \partial \bar{X}$ and $\partial Q / \partial X$ is

necessarily zero, because these terms are conjugates of each other. Now, take the derivative of Q with respect to the (possibly complex, row) vector $\mathbf{m}^{\mathrm{T}}$. Notice that $\partial Q/\partial \mathbf{m}^{\mathrm{T}}$ is the complex conjugate transpose of $\partial Q/\partial \mathbf{m}$. Thus, setting one to zero also sets the other to zero. Setting $\partial Q/\partial \mathbf{m}^{\mathrm{T}} = \mathbf{0}$ gives the normal equations:

$$\mathbf{0} \quad = \quad \frac{\partial Q}{\partial \mathbf{m}^{\mathrm{T}}} \quad = \quad \mathbf{F}^{\mathrm{T}} (\mathbf{F}\mathbf{m} - \mathbf{d}) \tag{46}$$

The result is merely the complex form of our earlier result (43). Therefore, differentiating by a complex vector is an abstract concept, but it gives the same set of equations as differentiating by each scalar component, and it saves much clutter.

From the frequency domain to the time domain

Where data fitting uses the notation $\mathbf{m} \rightarrow \mathbf{d}$, linear algebra and signal analysis often use the notation $\mathbf{x} \rightarrow \mathbf{y}$. Equation (4) is a frequency-domain quadratic form that we minimized by varying a single parameter, a Fourier coefficient. Now, we look at the same problem in the time domain. We see that the time domain offers flexibility with boundary conditions, constraints, and weighting functions. The notation is that a filter f_t has input x_t and output y_t. In Fourier space, it is expressed $Y = XF$. There are two applications to look at, unknown filter F and unknown input X.

Unknown filter

When inputs and outputs are given, the problem of finding an unknown filter appears to be overdetermined, so we write $\mathbf{y} \approx \mathbf{X}\mathbf{f}$ where the matrix $\mathbf{X}$ is a matrix of downshifted columns like (**??**). Thus, the quadratic form to be minimized is a restatement of equation (45) with filter definitions:

$$Q(\mathbf{f}^{\mathrm{T}}, \mathbf{f}) \quad = \quad (\mathbf{X}\mathbf{f} - \mathbf{y})^{\mathrm{T}} (\mathbf{X}\mathbf{f} - \mathbf{y}) \tag{47}$$

The solution $\mathbf{f}$ is found just as we found (46), and it is the set of simultaneous equations $\mathbf{0} = \mathbf{X}^{\mathrm{T}} (\mathbf{X}\mathbf{f} - \mathbf{y})$.

Unknown input: deconvolution with a known filter

For solving the unknown-input problem, we put the known filter f_t in a matrix of downshifted columns $\mathbf{F}$. Our statement of wishes is now to find x_t so that $\mathbf{y} \approx \mathbf{F}\mathbf{x}$. We can expect to have trouble finding unknown inputs x_t when we are dealing with certain kinds of filters, such as **bandpass filter**s. If the output is zero in a frequency band, we are never able to find the input in that band and need to prevent x_t from diverging there. We prevent divergence by the statement that we wish $\mathbf{0} \approx \epsilon\, \mathbf{x}$, where ϵ is a parameter that is small with exact size chosen by experimentation. Putting both wishes into a single, partitioned matrix equation gives:

$$\begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix} \quad \approx \quad \begin{bmatrix} \mathbf{r}_1 \\ \mathbf{r}_2 \end{bmatrix} \quad = \quad \begin{bmatrix} \mathbf{F} \\ \epsilon\, \mathbf{I} \end{bmatrix} \mathbf{x} \quad - \quad \begin{bmatrix} \mathbf{y} \\ \mathbf{0} \end{bmatrix} \tag{48}$$

To minimize the residuals $\mathbf{r}_1$ and $\mathbf{r}_2$, we can minimize the scalar $\mathbf{r}^{\mathrm{T}}\,\mathbf{r} = \mathbf{r}_1^{\mathrm{T}}\,\mathbf{r}_1 + \mathbf{r}_2^{\mathrm{T}}\,\mathbf{r}_2$. Expanding:

$$\begin{aligned} Q(\mathbf{x}^{\mathrm{T}},\mathbf{x}) &= (\mathbf{F}\mathbf{x}-\mathbf{y})^{\mathrm{T}}\,(\mathbf{F}\mathbf{x}-\mathbf{y}) + \epsilon^2\mathbf{x}^{\mathrm{T}}\,\mathbf{x} \\ &= (\mathbf{x}^{\mathrm{T}}\,\mathbf{F}^{\mathrm{T}}-\mathbf{y}^{\mathrm{T}})(\mathbf{F}\mathbf{x}-\mathbf{y}) + \epsilon^2\mathbf{x}^{\mathrm{T}}\,\mathbf{x} \end{aligned} \tag{49}$$

We solved this minimization in the frequency domain (beginning from equation (4)).

Formally the solution is found just as with equation (46), but this solution looks unappealing in practice because there are so many unknowns and the problem can be solved much more quickly in the Fourier domain. To motivate ourselves to solve this problem in the time domain, we need either to find an approximate solution method that is much faster, or find ourselves with an application that needs boundaries, or needs time-variable weighting functions.

KRYLOV SUBSPACE ITERATIVE METHODS

The **solution time** for simultaneous **linear equations** grows cubically with the number of unknowns. There are three regimes for solution; which regime is applicable depends on the number of unknowns in m. For m three or less, we use analytical methods. We also sometimes use analytical methods on matrices of size 4×4 if the matrix contains many zeros. My 1988 desktop workstation solved a 100×100 system in a minute. Ten years later, it would do a 600×600 system in roughly a minute. A nearby more powerful computer would do 1,000 × 1,000 in a minute. Because the computing effort increases with the third power of the size, and because $4^3 = 64 \approx 60$, an hour's work solves a four times larger matrix, namely 4,000 × 4,000 on the more powerful machine. For significantly larger values of m, exact numerical methods must be abandoned and **iterative method**s must be used.

The compute time for a rectangular matrix is slightly more pessimistic. It is the product of the number of data points n times the number of model points squared m^2 which is also the cost of computing the matrix $\mathbf{F}^{\mathrm{T}}\,\mathbf{F}$ from $\mathbf{F}$. Because the number of data points generally exceeds the number of model points $n > m$ by a substantial factor (to allow averaging of noises), it leaves us with significantly fewer than 4,000 points in model space.

A square image packed into a 4,096-point vector is a 64×64 array. The computer power for linear algebra to give us solutions that fit in a $k \times k$ image is thus proportional to k^6, which means that even though computer power grows rapidly, imaging resolution using "exact numerical methods" hardly grows at all from our 64×64 current practical limit.

The retina in our eyes captures an image of size roughly 1,000 × 1,000 which is a lot bigger than 64 × 64. Life offers us many occasions in which final images exceed the 4,000 points of a 64 × 64 array. To make linear algebra (and inverse theory) relevant to such applications, we investigate special techniques. A numerical technique known as the "**conjugate-direction method**" works well for all values of m and is our subject here. As with most simultaneous equation solvers, an exact answer (assuming exact arithmetic) is attained in a finite number of steps. And, if n and m are too large to allow enough iterations, the iterative methods can be interrupted at any stage, the partial result often proving useful. Whether or not a partial result actually is useful is the subject of much research; naturally, the results vary from one application to the next.

Sign convention

On the last day of the survey, a storm blew up, the sea got rough, and the receivers drifted further downwind. The data recorded that day had a larger than usual difference from that predicted by the final model. We could call $(\mathbf{d} - \mathbf{Fm})$ the ***experimental error***. (Here, $\mathbf{d}$ is data, $\mathbf{m}$ is model parameters, and $\mathbf{F}$ is their linear relation.)

The alternate view is that our theory was too simple. It lacked model parameters for the waves and the drifting cables. Because of this model oversimplification, we had a ***modeling error*** of the opposite polarity $(\mathbf{Fm} - \mathbf{d})$.

Strong experimentalists prefer to think of the error as experimental error, something for them to work out. Likewise, a strong analyst likes to think of the error as a theoretical problem. (Weaker investigators might be inclined to take the opposite view.)

Opposite to common practice, I define the **sign convention** for the error (or residual) as $(\mathbf{Fm} - \mathbf{d})$. Here is why. Minus signs are a source of confusion and errors. Putting the minus sign on the field data limits it to one location, while putting it in model space would spread it into as many parts as model space has parts.

> Beginners often feel disappointment when the data does not fit the model very well. They see it as a defect in the data instead of an opportunity to discover what our data contains that our theory does not.

Method of random directions and steepest descent

Let us minimize the sum of the squares of the components of the **residual** vector given by:

$$\text{residual} \quad = \quad \text{transform} \quad \text{model space} \quad - \quad \text{data space} \tag{50}$$

$$\begin{bmatrix} \\ \mathbf{r} \\ \\ \end{bmatrix} = \begin{bmatrix} \\ \mathbf{F} \\ \\ \end{bmatrix} \begin{bmatrix} \\ \mathbf{x} \\ \\ \end{bmatrix} - \begin{bmatrix} \\ \mathbf{d} \\ \\ \end{bmatrix} \tag{51}$$

A contour plot is based on an altitude function of space. The altitude is the **dot product** $\mathbf{r} \cdot \mathbf{r}$. By finding the lowest altitude, we are driving the residual vector $\mathbf{r}$ as close as we can to zero. If the residual vector $\mathbf{r}$ reaches zero, then we have solved the simultaneous equations $\mathbf{d} = \mathbf{Fx}$. In a two-dimensional world, the vector $\mathbf{x}$ has two components, (x_1, x_2). A contour is a curve of constant $\mathbf{r} \cdot \mathbf{r}$ in (x_1, x_2)-space. These contours have a statistical interpretation as contours of uncertainty in (x_1, x_2), with measurement errors in $\mathbf{d}$.

Let us see how a random search-direction can be used to reduce the residual $\mathbf{0} \approx \mathbf{r} = \mathbf{Fx} - \mathbf{d}$. Let $\Delta\mathbf{x}$ be an abstract vector with the same number of components as the solution $\mathbf{x}$, and let $\Delta\mathbf{x}$ contain arbitrary or random numbers. We add an unknown quantity α of vector $\Delta\mathbf{x}$ to the vector $\mathbf{x}$, and thereby create $\mathbf{x}_{\text{new}}$:

$$\mathbf{x}_{\text{new}} \quad = \quad \mathbf{x} + \alpha\Delta\mathbf{x} \tag{52}$$

The new $\mathbf{x}$ gives a new residual:

$$\mathbf{r}_{\text{new}} = \mathbf{F}\ \mathbf{x}_{\text{new}} - \mathbf{d} \tag{53}$$

$$\mathbf{r}_{\text{new}} = \mathbf{F}(\mathbf{x} + \alpha\Delta\mathbf{x}) - \mathbf{d} \tag{54}$$

$$\mathbf{r}_{\text{new}} = \mathbf{r} + \alpha\Delta\mathbf{r} = (\mathbf{F}\mathbf{x} - \mathbf{d}) + \alpha\mathbf{F}\Delta\mathbf{x} \tag{55}$$

which defines $\Delta\mathbf{r} = \mathbf{F}\Delta\mathbf{x}$.

Next, we adjust α to minimize the dot product: $\mathbf{r}_{\text{new}} \cdot \mathbf{r}_{\text{new}}$

$$(\mathbf{r} + \alpha\Delta\mathbf{r}) \cdot (\mathbf{r} + \alpha\Delta\mathbf{r}) = \mathbf{r}\cdot\mathbf{r} + 2\alpha(\mathbf{r}\cdot\Delta\mathbf{r}) + \alpha^2\Delta\mathbf{r}\cdot\Delta\mathbf{r} \tag{56}$$

Set to zero its derivative with respect to α:

$$0 = 2\mathbf{r}\cdot\Delta\mathbf{r} + 2\alpha\Delta\mathbf{r}\cdot\Delta\mathbf{r} \tag{57}$$

which says that the new residual $\mathbf{r}_{\text{new}} = \mathbf{r} + \alpha\Delta\mathbf{r}$ is perpendicular to the "fitting function" $\Delta\mathbf{r}$. Solving gives the required value of α.

$$\alpha = -\frac{(\mathbf{r}\cdot\Delta\mathbf{r})}{(\Delta\mathbf{r}\cdot\Delta\mathbf{r})} \tag{58}$$

A "computation **template**" for the method of random directions is:

$\mathbf{r} \longleftarrow \mathbf{F}\mathbf{x} - \mathbf{d}$
iterate {
$\Delta\mathbf{x} \longleftarrow$ random numbers
$\Delta\mathbf{r} \longleftarrow \mathbf{F}\ \Delta\mathbf{x}$
$\alpha \longleftarrow -(\mathbf{r}\cdot\Delta\mathbf{r})/(\Delta\mathbf{r}\cdot\Delta\mathbf{r})$
$\mathbf{x} \longleftarrow \mathbf{x} + \alpha\Delta\mathbf{x}$
$\mathbf{r} \longleftarrow \mathbf{r} + \alpha\Delta\mathbf{r}$
}

A nice thing about the method of random directions is that you do not need to know the adjoint operator $\mathbf{F}^{\mathrm{T}}$.

In practice, random directions are rarely used. It is more common to use the **gradient** direction than a random direction. Notice that a vector of the size of $\Delta\mathbf{x}$ is:

$$\mathbf{g} = \mathbf{F}^{\mathrm{T}}\,\mathbf{r} \tag{59}$$

Recall this vector can be found by taking the gradient of the size of the residuals:

$$\frac{\partial}{\partial\mathbf{x}^{\mathrm{T}}}\ \mathbf{r}\cdot\mathbf{r} = \frac{\partial}{\partial\mathbf{x}^{\mathrm{T}}}\ (\mathbf{x}^{\mathrm{T}}\,\mathbf{F}^{\mathrm{T}} - \mathbf{d}^{\mathrm{T}})\,(\mathbf{F}\,\mathbf{x} - \mathbf{d}) = \mathbf{F}^{\mathrm{T}}\ \mathbf{r} \tag{60}$$

Choosing $\Delta\mathbf{x}$ to be the gradient vector $\Delta\mathbf{x} = \mathbf{g} = \mathbf{F}^{\mathrm{T}}\,\mathbf{r}$ is called "the method of **steepest descent**."

Starting from a model $\mathbf{x} = \mathbf{m}$ (which may be zero), the following is a **template** of pseudocode for minimizing the residual $\mathbf{0} \approx \mathbf{r} = \mathbf{F}\mathbf{x} - \mathbf{d}$ by the steepest-descent method:

$$
\begin{array}{lll}
\mathbf{r} & \longleftarrow & \mathbf{F}\mathbf{x} - \mathbf{d} \\
\text{iterate } \{ & & \\
\quad \Delta\mathbf{x} & \longleftarrow & \mathbf{F}^{\mathrm{T}}\,\mathbf{r} \\
\quad \Delta\mathbf{r} & \longleftarrow & \mathbf{F}\,\Delta\mathbf{x} \\
\quad \alpha & \longleftarrow & -(\mathbf{r}\cdot\Delta\mathbf{r})/(\Delta\mathbf{r}\cdot\Delta\mathbf{r}) \\
\quad \mathbf{x} & \longleftarrow & \mathbf{x} + \alpha\Delta\mathbf{x} \\
\quad \mathbf{r} & \longleftarrow & \mathbf{r} + \alpha\Delta\mathbf{r} \\
\quad \} & &
\end{array}
$$

Good science and engineering is finding something unexpected. Look for the unexpected both in data space and in model space. In data space, you look at the residual $\mathbf{r}$. In model space, you look at the residual projected there $\mathbf{F}^{\mathrm{T}}\,\mathbf{r}$. What does it mean? It is simply Δm, the changes you need to make to your model. It means more in later chapters, where the operator $\mathbf{F}$ is a column vector of operators that are fighting with one another to grab the data.

Why steepest descent is so slow

Before we can understand why the **conjugate-direction method** is so fast, we need to see why the **steepest-descent method** is so slow. The process of selecting α is called "**line search**," but for a linear problem like the one we have chosen here, we hardly recognize choosing α as searching a line. A more graphic understanding of the whole process is possible from considering a two-dimensional space, where the vector of unknowns $\mathbf{x}$ has just two components, x_1 and x_2. Then, the size of the residual vector $\mathbf{r}\cdot\mathbf{r}$ can be displayed with a contour plot in the plane of (x_1, x_2). Figure 7 shows a contour plot of the penalty function of $(x_1, x_2) = (m_1, m_2)$. The gradient is perpendicular to the contours. Contours and gradients are *curved lines*. When we use the steepest-descent method, we start at a point and compute the gradient direction at that point. Then, we begin a *straight-line* descent in that direction. The gradient direction curves away from our direction of travel, but we continue on our straight line until we have stopped descending and are about to ascend. There we stop, compute another gradient vector, turn in that direction, and descend along a new straight line. The process repeats until we get to the bottom or until we get tired.

What could be wrong with such a direct strategy? The difficulty is at the stopping locations. These locations occur where the descent direction becomes *parallel* to the contour lines. (There the path becomes level.) So, after each stop, we turn 90° from parallel to perpendicular to the local contour line for the next descent. What if the final goal is at a 45° angle to our path? A 45° turn cannot be made. Instead of moving like a rain drop down the centerline of a rain gutter, we move along a fine-toothed zigzag path, crossing and recrossing the centerline. The gentler the slope of the rain gutter, the finer the teeth on the zigzag path.

Null space and iterative methods

In applications where we fit $\mathbf{d} \approx \mathbf{F}\mathbf{x}$, there might exist a vector (or a family of vectors) defined by the condition $\mathbf{0} = \mathbf{F}\mathbf{x}_{\text{null}}$. This family is called a **null space**. For example, if the

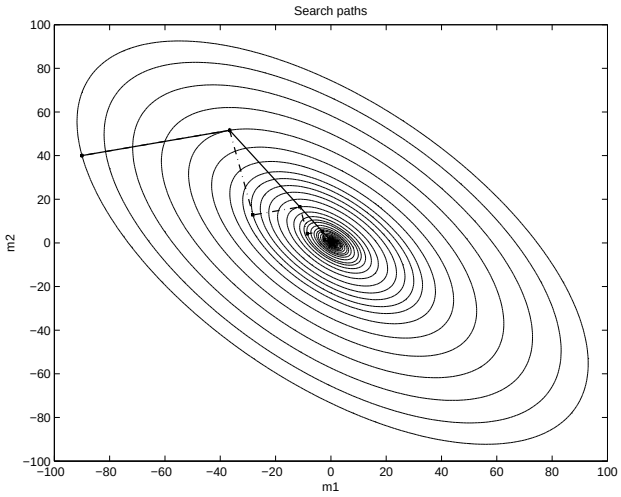


Figure 7: Route of steepest descent (black) and route of conjugate direction (light grey or red).

operator $\mathbf{F}$ is a time derivative, then the null space is the constant function; if the operator is a second derivative, then the null space has two components, a constant function and a linear function, or combinations of both. The null space is a family of model components that have no effect on the data.

When we use the steepest-descent method, we iteratively find solutions by this updating:

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha \Delta \mathbf{x} \tag{61}$$

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha \mathbf{F}^{\mathrm{T}} \mathbf{r} \tag{62}$$

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha \mathbf{F}^{\mathrm{T}} (\mathbf{F}\mathbf{x} - \mathbf{d}) \tag{63}$$

After we have iterated to convergence, the gradient $\Delta \mathbf{x} = \mathbf{F}^{\mathrm{T}} \mathbf{r}$ vanishes. Adding any $\mathbf{x}_{\mathrm{null}}$ to $\mathbf{x}$ does not change the residual $\mathbf{r} = \mathbf{F}\mathbf{x} - \mathbf{d}$. Because $\mathbf{r}$ is unchanged, $\Delta \mathbf{x} = \mathbf{F}^{\mathrm{T}} \mathbf{r}$ remains zero and $\mathbf{x}_{i+1} = \mathbf{x}_i$. Thus, we conclude that any null space in the initial guess $\mathbf{x}_0$ remains there unaffected by the gradient-descent process. So, in the presense of null space, the answer we get from an iterative method depends on the starting guess. Oops! The analytic solution does not do any better. It needs to deal with a singular matrix. Existence of a null space destroys the uniqueness of any resulting model.

Linear algebra theory enables us to dig up the entire null space should we so desire. On the other hand, the computer demands might be vast. Even the memory for holding the many $\mathbf{x}$ vectors could be prohibitive. A much simpler and more practical goal is to find out if the null space has any members, and if so, to view some members. To try to see a member of the null space, we take two starting guesses, and we run our iterative solver for each. If the two solutions, $\mathbf{x}_1$ and $\mathbf{x}_2$, are the same, there is no null space. If the solutions differ, the difference is a member of the null space. Let us see why: Suppose after iterating to minimum residual we find:

$$\mathbf{r}_1 = \mathbf{F}\mathbf{x}_1 - \mathbf{d} \tag{64}$$

$$\mathbf{r}_2 = \mathbf{F}\mathbf{x}_2 - \mathbf{d} \tag{65}$$

We know that the residual squared is a convex quadratic function of the unknown $\mathbf{x}$. Mathematically that means the minimum value is unique, so $\mathbf{r}_1 = \mathbf{r}_2$. Subtracting, we find

$0 = \mathbf{r}_1 - \mathbf{r}_2 = \mathbf{F}(\mathbf{x}_1 - \mathbf{x}_2)$ proving that $\mathbf{x}_1 - \mathbf{x}_2$ is a model in the null space. Adding $\mathbf{x}_1 - \mathbf{x}_2$ to any to any model $\mathbf{x}$ does not change the modeled data.

A practical way to learn about the existence of null spaces and see samples is to try gradient-descent methods beginning from various different starting guesses.

"Did I fail to run my iterative solver long enough?" is a question you might have. If two residuals from two starting solutions are not equal, $\mathbf{r}_1 \neq \mathbf{r}_2$, then you should be running your solver through more iterations.

If two different starting solutions produce two different residuals, then you did not run your solver through enough iterations.

The magical property of the conjugate direction method

In the **conjugate-direction method**, not a line, but rather a plane, is searched. A plane is made from an arbitrary linear combination of two vectors. One vector is chosen to be the gradient vector, say $\mathbf{g}$. The other vector is chosen to be the previous descent step vector, say $\mathbf{s} = \mathbf{x}_j - \mathbf{x}_{j-1}$. Instead of $\alpha\,\mathbf{g}$, we need a linear combination, say $\alpha\mathbf{g} + \beta\mathbf{s}$. For minimizing quadratic functions the **plane search** requires only the solution of a two-by-two set of linear equations for α and β.

The conjugate-direction (CD) method described in this book has a magical property shared by the more famous conjugate-gradient method. This magical property is not proven in this book, but it may be found in many sources. Although both methods are iterative methods, both converge on the exact answer (assuming perfect numerical precision) in a fixed number of steps. That number is the number of components in model space $\mathbf{x}$.

Where we benefit from iterative methods is where convergence is more rapid than the theoretical requirement. Whether or not that happens, depends on the problem at hand. Reflection seismology has many problems so massive they are said to be solved simply by one application of the adjoint operator. The idea that such solutions might be improved by a small number of iterations is very appealing.

Conjugate-direction theory for programmers

Fourier-transformed variables are often capitalized. This convention is helpful here, so in this subsection only, we capitalize vectors transformed by the $\mathbf{F}$ matrix. As everywhere, a matrix, such as $\mathbf{F}$, is printed in **boldface** type but in this subsection, vectors are *not* printed in boldface. Thus, we define the solution, the solution step (from one iteration to the next), and the gradient by:

$$X = \mathbf{F}\, x \qquad \text{modeled data } = \mathbf{F} \text{ model} \tag{66}$$

$$S_j = \mathbf{F}\, s_j \qquad \text{solution change} \tag{67}$$

$$G_j = \mathbf{F}\, g_j \qquad \Delta\mathbf{r} = \mathbf{F}\Delta\mathbf{m} \tag{68}$$

A linear combination in solution space, say $s+g$, corresponds to $S+G$ in the conjugate space, the data space, because $S+G = \mathbf{F}s + \mathbf{F}g = \mathbf{F}(s+g)$. According to equation (51), the residual is the modeled data minus the observed data.

$$R \quad = \quad \mathbf{F}x \;-\; D \quad = \quad X \;-\; D \tag{69}$$

The solution x is obtained by a succession of steps s_j, say:

$$x \quad = \quad s_1 \;+\; s_2 \;+\; s_3 \;+\; \cdots \tag{70}$$

The last stage of each iteration is to update the solution and the residual:

$$\text{solution update}: \qquad x \;\leftarrow x \quad + s \tag{71}$$

$$\text{residual update}: \qquad R \;\leftarrow R \quad + S \tag{72}$$

The *gradient* vector g is a vector with the same number of components as the solution vector x. A vector with this number of components is:

$$g \;=\; \mathbf{F}^{\mathrm{T}}\, R \quad = \quad \text{gradient} \tag{73}$$

$$G \;=\; \mathbf{F}\; g \quad = \quad \text{conjugate gradient} \;=\; \Delta r \tag{74}$$

The gradient g in the transformed space is G, also known as the **conjugate gradient**.

What is our solution update $\Delta\mathbf{x} = \mathbf{s}$? It is some unknown amount α of the gradient $\mathbf{g}$ plus another unknown amount β of the previous step $\mathbf{s}$. Likewise in residual space.

$$\Delta\mathbf{x} \;=\; \alpha\mathbf{g} + \beta\mathbf{s} \qquad \text{model space} \tag{75}$$

$$\Delta\mathbf{r} \;=\; \alpha\mathbf{G} + \beta\mathbf{S} \qquad \text{data space} \tag{76}$$

The minimization (56) is now generalized to scan not only in a line with α, but simultaneously another line with β. The combination of the two lines is a plane. We now set out to find the location in this plane that minimizes the quadratic Q.

$$Q(\alpha,\beta) \quad = \quad (R + \alpha G + \beta S) \;\cdot\; (R + \alpha G + \beta S) \tag{77}$$

The minimum is found at $\partial Q/\partial\alpha \;=\; 0$ and $\partial Q/\partial\beta \;=\; 0$, namely,

$$0 \;=\; G \;\cdot\; (R + \alpha G + \beta S) \tag{78}$$

$$0 \;=\; S \;\cdot\; (R + \alpha G + \beta S) \tag{79}$$

$$-\quad \begin{bmatrix} (G\cdot R) \\ (S\cdot R) \end{bmatrix} \quad = \quad \begin{bmatrix} (G\cdot G) & (S\cdot G) \\ (G\cdot S) & (S\cdot S) \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \tag{80}$$

Equation (81) is a set of two equations for α and β. Recall the inverse of a 2×2 matrix, equation (111) and get:

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix} \;=\; \frac{-1}{(G\cdot G)(S\cdot S) - (G\cdot S)^2} \begin{bmatrix} (S\cdot S) & -(S\cdot G) \\ -(G\cdot S) & (G\cdot G) \end{bmatrix} \begin{bmatrix} (G\cdot R) \\ (S\cdot R) \end{bmatrix} \tag{81}$$

The many applications in this book all need to find α and β with (81), and then update the solution with (71) and update the residual with (72). Thus, we package these activities in a subroutine named `cgstep()`. To use that subroutine, we have a computation **template** with repetitive work done by subroutine `cgstep()`. This template (or pseudocode) for minimizing the residual $\mathbf{0} \approx \mathbf{r} = \mathbf{F}\mathbf{x} - \mathbf{d}$ by the conjugate-direction method is:

$$
\begin{array}{lll}
\mathbf{r} & \longleftarrow & \mathbf{F}\mathbf{x} - \mathbf{d} \\
\text{iterate } \{ & & \\
\quad \Delta\mathbf{x} & \longleftarrow & \mathbf{F}^{\mathrm{T}}\,\mathbf{r} \\
\quad \Delta\mathbf{r} & \longleftarrow & \mathbf{F}\,\Delta\mathbf{x} \\
\quad (\mathbf{x}, \mathbf{r}) & \longleftarrow & \text{cgstep}(\mathbf{x}, \mathbf{r}, \Delta\mathbf{x}, \Delta\mathbf{r}) \\
\quad \} & &
\end{array}
$$

where the subroutine `cgstep()` remembers the previous iteration and works out the step size and adds in the proper proportion of the $\Delta\mathbf{x}$ of the previous step.

Routine for one step of conjugate-direction descent

The conjugate vectors G and S in the program are denoted by `gg` and `ss`. The inner part of the conjugate-direction task is in function `cgstep()`.

Observe the `cgstep()` function has a logical parameter called `forget`. This parameter does not need to be input. In the normal course of things, `forget` is true on the first iteration and false on subsequent iterations. On the first iteration there is no previous step, so the conjugate direction method is reduced to the steepest descent method. At any iteration, however, you have the option to set `forget=true`, which amounts to restarting the calculation from the current location, something we rarely find reason to do.

A basic solver program

There are many different methods for iterative least-square estimation some of which are discussed later in this book. The conjugate-gradient (CG) family (including the first order conjugate-direction method previously described) share the property that theoretically they achieve the solution in n iterations, where n is the number of unknowns. The various CG methods differ in their numerical errors, memory required, adaptability to nonlinear optimization, and their requirements on accuracy of the adjoint. What we do in this section is to show you the generic interface.

None of us is an expert in both geophysics and in optimization theory (OT), yet we need to handle both. We would like to have each group write its own code with a relatively easy interface. The problem is that the OT codes must invoke the physical operators yet the OT codes should not need to deal with all the data and parameters needed by the physical operators.

In other words, if a practitioner decides to swap one solver for another, the only thing needed is the name of the new solver.

The operator entrance is for the geophysicist, who formulates the estimation application. The solver entrance is for the specialist in numerical algebra, who designs a new optimization method. The C programming language allows us to achieve this design goal by means of generic function interfaces.

A generic solver subroutine is `tinysolver`. It is simplified substantially from the library version, which has a much longer list of optional arguments. (The `forget` parameter is not needed by the solvers we discuss first.)

api/c/cgstep.c

```
51      if (forget) {
52          for (i = 0; i < nx; i++) S[i] = 0.;
53          for (i = 0; i < ny; i++) Ss[i] = 0.;
54          beta = 0.0;
55          alfa = cblas_dsdot( ny, gg, 1, gg, 1);
56          /* Solve G . ( R + G*alfa) = 0 */
57          if (alfa <= 0.) return;
58          alfa = - cblas_dsdot( ny, gg, 1, rr, 1) / alfa;
59      } else {
60          /* search plane by solving 2-by-2
61             G . (R + G*alfa + S*beta) = 0
62             S . (R + G*alfa + S*beta) = 0 */
63          gdg = cblas_dsdot( ny, gg, 1, gg, 1);
64          sds = cblas_dsdot( ny, Ss, 1, Ss, 1);
65          gds = cblas_dsdot( ny, gg, 1, Ss, 1);
66          if (gdg == 0. || sds == 0.) return;
67          determ = 1.0 - (gds/gdg)*(gds/sds);
68          if (determ > EPSILON) determ *= gdg * sds;
69          else determ = gdg * sds * EPSILON;
70          gdr = - cblas_dsdot( ny, gg, 1, rr, 1);
71          sdr = - cblas_dsdot( ny, Ss, 1, rr, 1);
72          alfa = ( sds * gdr - gds * sdr ) / determ;
73          beta = (-gds * gdr + gdg * sdr ) / determ;
74      }
75      cblas_sscal(nx,beta,S,1);
76      cblas_saxpy(nx,alfa,g,1,S,1);
77
78      cblas_sscal(ny,beta,Ss,1);
79      cblas_saxpy(ny,alfa,gg,1,Ss,1);
80
81      for (i = 0; i < nx; i++) {
82          x[i] +=  S[i];
83      }
84      for (i = 0; i < ny; i++) {
85          rr[i] += Ss[i];
86      }
```

api/c/tinysolver.c

```
23 void sf_tinysolver (sf_operator Fop       /* linear operator */,
24                    sf_solverstep stepper /* stepping function */,
25                    int nm                /* size of model */,
26                    int nd                /* size of data */,
27                    float* m              /* estimated model */,
28                    const float* m0       /* starting model */,
29                    const float* d        /* data */,
30                    int niter             /* iterations */)
31 /*< Generic linear solver. Solves oper{x} =~ dat >*/
32 {
33     int i, iter;
34     float *g, *rr, *gg;
35 
36     g =  sf_floatalloc (nm);
37     rr = sf_floatalloc (nd);
38     gg = sf_floatalloc (nd);
39 
40     for (i=0; i < nd; i++) rr[i] = - d[i];
41     if (NULL==m0) {
42         for (i=0; i < nm; i++) m[i] = 0.0;
43     } else {
44         for (i=0; i < nm; i++) m[i] = m0[i];
45         Fop (false, true, nm, nd, m, rr);
46     }
47 
48     for (iter=0; iter < niter; iter++) {
49         Fop (true, false, nm, nd, g, rr);
50         Fop (false, false, nm, nd, g, gg);
51 
52         stepper (false, nm, nd, m, g, rr, gg);
53     }
54 
55     free (g);
56     free (rr);
57     free (gg);
58 }
```

The two most important arguments in `tinysolver()` are the operator function `Fop`, which is defined by the interface from Chapter **??**, and the solver function `stepper`, which implements one step of an iterative estimation. For example, a practitioner who choses to use our new `cgstep()` on page 23 for iterative solving the operator `matmult` on page ?? would write the call

```
tinysolver ( matmult_lop, cgstep, ...
```

The other required parameters to `tinysolver()` are `dat` (the data we want to fit), `x` (the model we want to estimate), and `niter` (the maximum number of iterations). There are also a couple of optional arguments. For example, `x0` is the starting guess for the model. If this parameter is omitted, the model is initialized to zero. To output the final residual vector, we include a parameter called `res`, which is optional as well. We will watch how the list of optional parameters to the generic solver routine grows as we attack more and more complex applications in later chapters.

Fitting success and solver success

Every time we run a data modeling program, we have access to two publishable numbers $1 - |\mathbf{r}|/|\mathbf{d}|$ and $1 - |\mathbf{F}^{\mathrm{T}}\,\mathbf{r}|/|\mathbf{F}^{\mathrm{T}}\,\mathbf{d}|$. The first says how well the model fits the data. The second says how well we did the job of finding out.

Define the residual $\mathbf{r} = \mathbf{F}\mathbf{m} - \mathbf{d}$ and the "size" of any vector, such as the data vector, as $|\mathbf{d}| = \sqrt{\mathbf{d} \cdot \mathbf{d}}$. The number $1 - |\mathbf{r}|/|\mathbf{d}|$ is called the "success at fitting data." (Any data-space weighting function should have been incorporated in both $\mathbf{F}$ and $\mathbf{d}$.)

While the data fitting success is of interest to everyone, the second number $1 - |\mathbf{F}^{\mathrm{T}}\,\mathbf{r}|/|\mathbf{F}^{\mathrm{T}}\,\mathbf{d}|$ is of interest in QA (quality analysis). In giant problems, especially those arising in seismology, running iterations to completion is impractical. A question always of interest is whether or not enough iterations have been run. This number gives us guidance to where more effort could be worthwhile.

$$0 \quad \leq \quad \text{Success} \quad \leq \quad 1$$
$$\text{Fitting success:} \quad 1 - |\mathbf{r}|/|\mathbf{d}|$$
$$\text{Numerical success:} \quad 1 - |\mathbf{F}^{\mathrm{T}}\,\mathbf{r}|/|\mathbf{F}^{\mathrm{T}}\,\mathbf{d}|$$

Roundoff

Surprisingly, as a matter of practice, the simple conjugate-direction method defined in this book is more reliable than the conjugate-gradient method defined in the formal professional literature. I know this sounds unlikely, but I can tell you why.

In large applications, numerical roundoff can be a problem. Calculations need to be done in higher precision. The conjugate gradient method depends on you to supply an operator with the adjoint correctly computed. Any roundoff in computing the operator should somehow be matched by the roundoff in the adjoint. But that is unrealistic. Thus, optimization may diverge while theoretically converging. The conjugate direction method does not mind the roundoff; it simply takes longer to converge.

Let us see an example of a situation in which roundoff becomes a problem. Suppose we add 100 million 1.0s. You expect the sum to be 100 million. I got a sum of 16.7 million. Why? After the sum gets to 16.7 million, adding a one to it adds nothing. The extra 1.0 disappears in single precision roundoff.

```
        real function one(sum);  one=1.;  return; end
        integer i;   real sum
        do i=1, 100000000
                sum = sum + one(sum)
        write (0,*) sum;    stop;  end
1.6777216E+07
```

The previous code must be a little more complicated than I had hoped because modern compilers are so clever. When told to add all the values in a vector the compiler wisely adds the numbers in groups, and then adds the groups. Thus, I had to hide the fact I was adding ones by getting those ones from a subroutine that seems to depend upon the sum (but really does not).

Test case: solving some simultaneous equations

Now we assemble a module `cgtest` for solving simultaneous equations. Starting with the conjugate-direction module `cgstep` on page 23 we insert the module `matmult` on page ?? as the linear operator.

user/pwd/cgtest.c

```
23 void cgtest(int nx, int ny, float *x,
24              const float *yy, float **fff, int niter)
25 /*< testing conjugate gradients with matrix multiplication >*/
26 {
27     matmult_init( fff);
28     sf_tinysolver( matmult_lop, sf_cgstep,
29                    nx, ny, x, NULL, yy, niter);
30     sf_cgstep_close();
31 }
```

The following shows the solution to a 5×4 set of simultaneous equations. Observe that the "exact" solution is obtained in the last step. Because the data and answers are integers, it is quick to check the result manually.

```
d transpose
     3.00      3.00      5.00      7.00      9.00

F transpose
     1.00      1.00      1.00      1.00      1.00
     1.00      2.00      3.00      4.00      5.00
     1.00      0.00      1.00      0.00      1.00
```

```
      0.00       0.00       0.00       1.00       1.00

for iter = 0, 4
x     0.43457383  1.56124675  0.27362058  0.25752524
res -0.73055887  0.55706739  0.39193487 -0.06291389 -0.22804642
x     0.51313990  1.38677299  0.87905121  0.56870615
res -0.22103602  0.28668585  0.55251014 -0.37106210 -0.10523783
x     0.39144871  1.24044561  1.08974111  1.46199656
res -0.27836466 -0.12766013  0.20252672 -0.18477242  0.14541438
x     1.00001287  1.00004792  1.00000811  2.00000739
res  0.00006878  0.00010860  0.00016473  0.00021179  0.00026788
x     1.00000024  0.99999994  0.99999994  2.00000024
res -0.00000001 -0.00000001  0.00000001  0.00000002 -0.00000001
```

INVERSE NMO STACK

To illustrate an example of solving a huge set of simultaneous equations without ever writing down the matrix of coefficients, we consider how ***back projection*** can be upgraded toward ***inversion*** in the application called **moveout and stack**.

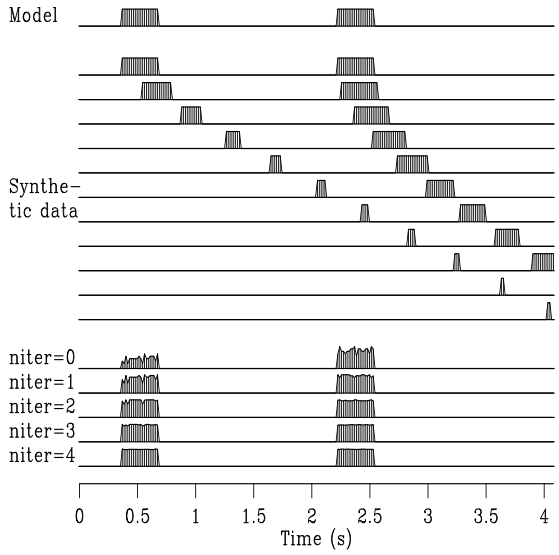


Figure 8: Top is a model trace $\mathbf{m}$. Next, are the synthetic data traces, $\mathbf{d} = \mathbf{Fm}$. Then, labeled `niter=0` is the **stack**, a result of processing by adjoint modeling. Increasing values of `niter` show $\mathbf{m}$ as a function of iteration count in the fitting goal $\mathbf{d} \approx \mathbf{Fm}$. (Carlos Cunha-Filho)

The seismograms at the bottom of Figure 8 show the first four iterations of conjugate-direction inversion. You see the original rectangle-shaped waveform returning as the iterations proceed. Notice also on the **stack** that the early and late events have unequal amplitudes, but after enough iterations match the original model. Mathematically, we can denote the top trace as the model $\mathbf{m}$, the synthetic data signals as $\mathbf{d} = \mathbf{Fm}$, and the stack as $\mathbf{F}^{\mathrm{T}}\,\mathbf{d}$. The conjugate-gradient algorithm optimizes the fitting goal $\mathbf{d} \approx \mathbf{Fx}$ by variation of $\mathbf{x}$, and the figure shows $\mathbf{x}$ converging to $\mathbf{m}$. Because there are 256 unknowns in $\mathbf{m}$, it is gratifying to see good convergence occurring after the first four iterations. The fitting is done by module `invstack`, which is just like `cgtest` on the preceding page except that the matrix-multiplication operator `matmult` on page ?? has been replaced by `imospray`. Studying the program, you can deduce that, except for a scale factor, the output at `niter=0` is identical to the stack $\mathbf{M}^{\mathrm{T}}\mathbf{d}$. All the signals in Figure 8 are intrinsically the same scale.

This simple inversion is inexpensive. Has anything been gained over conventional stack?

user/gee/invstack.c

```
24 void invstack(int nt, float *model, int nx, const float *gather,
25                float t0, float x0,
26                float dt, float dx, float slow, int niter)
27 /*< NMO stack by inverse of forward modeling */
28 {
29     imospray_init( slow, x0,dx, t0,dt, nt, nx);
30     sf_tinysolver( imospray_lop, sf_cgstep,
31                    nt, nt*nx, model, NULL, gather, niter);
32     sf_cgstep_close ();
33     imospray_close ();  /* garbage collection */
34 }
```

First, though we used nearest-neighbor interpolation, we managed to preserve the spectrum of the input, apparently all the way to the Nyquist frequency. Second, we preserved the true amplitude scale without ever bothering to think about (1) dividing by the number of contributing traces, (2) the amplitude effect of NMO stretch, or (3) event truncation.

With depth-dependent velocity, wave fields become much more complex at wide offset. NMO soon fails, but wave-equation forward modeling offers interesting opportunities for inversion.

FLATTENING 3-D SEISMIC DATA

Here is an expression that on first sight seems to say nothing:

$$\nabla\tau \quad = \quad \begin{bmatrix} \frac{\partial\tau}{\partial x} \\ \\ \frac{\partial\tau}{\partial y} \end{bmatrix} \tag{82}$$

Equation (82) looks like a tautology, a restatement of basic mathematical notation. But it is a tautology only if $\tau(x,y)$ is known and the derivatives come from it. When $\tau(x,y)$ is not known but the partial derivatives are observed, then, we have two measurements at each (x,y) location for the one unknown τ at that location. In Figure **??**, we have seen how to flatten 2-D seismic data. The 3-D process (x,y,τ) is much more interesting because of the possibility encountering a vector field that cannot be derived from a scalar field.

The easy case is when you can move around the (x,y) plane adding up τ by steps of $d\tau/dx$ and $d\tau/dy$ and find upon returning to your starting location that the total time change τ is zero. When $d\tau/dx$ and $d\tau/dy$ are derived from noisy data, such sums around a path often are not zero. Old time seismologists would say, "The survey lines don't tie." Mathematically, it is like an electric field vector that may be derived from a potential field unless the loop encloses a changing **magnetic** field.

We would like a solution for τ that gives the best fit of all the data (the stepouts $d\tau/dx$ and $d\tau/dy$) in a volume. Given a volume of data $d(t,x,y)$, we seek the best $\tau(x,y)$ such that $w(t,x,y) = d(t-\tau(x,y),x,y)$ is flattened. Let us get it.

We write a regression, a residual $\mathbf{r}$ that we minimize to find a best fitting $\tau(x,y)$ or maybe $\tau(x,y,t)$. Let d be the measurements in the vector in equation (82), the measurements throughout the (t,x,y)-volume. Expressed as a regression, equation (82) becomes:

$$\mathbf{0} \quad \approx \quad \mathbf{r} \quad = \quad \nabla\tau \; - \; \mathbf{d} \tag{83}$$

Figure 9 shows slices through a cube of seismic data. A paper book is inadequate to display all the images required to compare before and after (one image of output is blended over multiple images of input), therefore, we move on to a radar application of much the same idea, but in 2-D instead of 3-D.

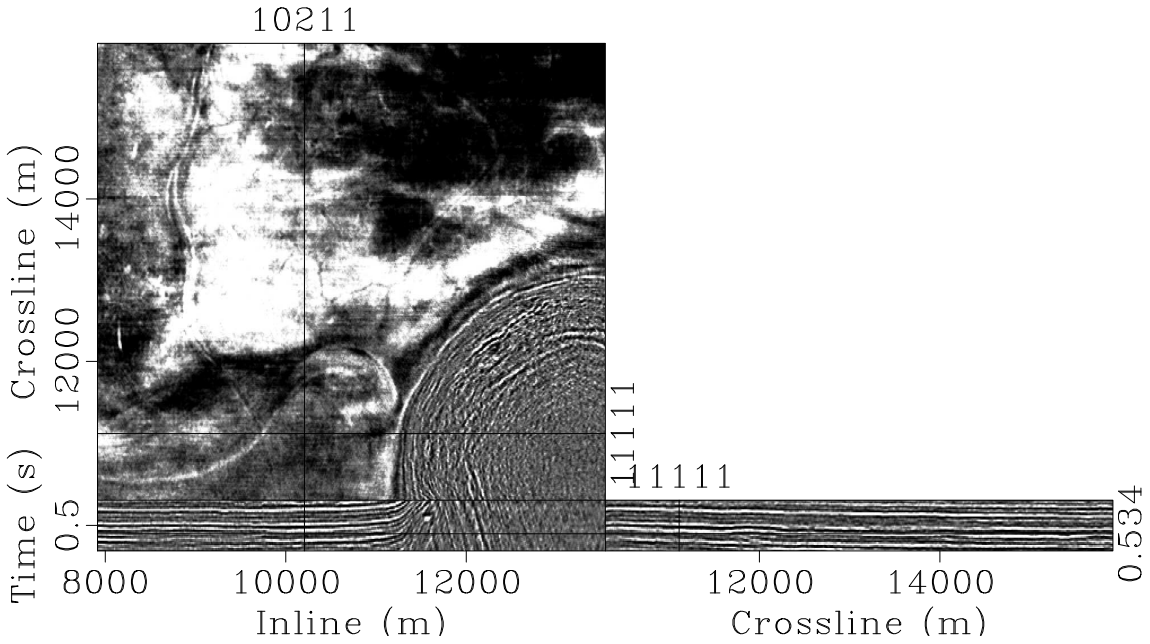


Figure 9: [Jesse Lomask] Chevron data cube from the Gulf of Mexico. Shown are three planes within the cube. A salt dome (lower right corner in the top plane) has pushed upward, dragging bedding planes (seen in the bottom two orthogonal planes) along with it.

Let us see how the coming 3-D illustrations were created. First we need code for vector gradient with its adjoint, negative vector divergence. Here it is: In a kind of magic, all we need to fit our regression (82) is to pass the `igrad2` module to the Krylov subspace solver, simple solver using `cgstep`, but first we need to compute $\mathbf{d}$ by calculating dt/dx and dt/dy between all the mesh points.

```
do iy=1,ny { # Calculate x-direction dips: px
             call puck2d(dat(:,:,iy),coh_x,px,res_x,boxsz,nt,nx)
           }
do ix=1,nx { # Calculate y-direction dips: py
             call puck2d(dat(:,ix,:),coh_y,py,res_y,boxsz,nt,ny)
           }
do it=1,nt { # Integrate dips: tau
             call dipinteg(px(it,:,:),py(it,:,:),tau,niter,verb,nx,ny)
           }
```

api/c/igrad2.c

```
48      for (i2=0; i2 < n2-1; i2++) {
49          for (i1=0; i1 < n1-1; i1++) {
50              i = i1+i2*n1;
51              if (adj) {
52                  p[i+1]  += r[i];
53                  p[i+n1] += r[i+n12];
54                  p[i]    -= (r[i] + r[i+n12]);
55              } else {
56                  r[i]     += (p[i+1]  - p[i]);
57                  r[i+n12] += (p[i+n1] - p[i]);
58              }
59          }
60      }
```

The code says first to initialize the gradient operator. Convert the 2-D plane of dt/dx to a vector. Likewise dt/dy. Concatenate these two vectors into a single column vector $\mathbf{d}$ like in equation (82). Tell the simple solver to make its steps with to use `cgstep` with the linear operator `igrad2`.

Gulf of Mexico Salt Piercement Example (Jesse Lomask)

Figure 9 shows a 3-D seismic data cube from the Gulf of Mexico provided by Chevron. A volume of data cannot be displayed on the page of a book. The display here consists of three slices from the volume. Top is a (t_0, x, y) slice, also called a "time slice." Beneath it is a (t, x, y_0) slice; aside that is a (t, x_0, y) slice, depth slices in orthogonal directions. Intersections of the slices within the cube are shown by the heavy black lines on the faces of the cube. The circle in the lower right corner of the top slice is an eruption of salt (which, like ice, under high pressure will flow like a liquid). Inside the salt there are no reflections so the data should be ignored there. Outside the salt we see layers, simple horizons of sedimentary rock. As the salt has pushed upward it has dragged bedding planes upward with it. Presuming the bedding to contain permeable sandstones and impermeable shales, the pushed up bedding around the salt is a prospective oil trap. The time slice in the top panel shows ancient river channels, some large, some small, that are now deeply buried. These may also contain sand. Being natural underground "oil pipes" they are of great interest. To see these pipes as they approach the salt dome we need a picture not at a constant t, but at a constant $t - \tau(x, y)$.

Figure 10 shows a time slice of the original cube and the flattened cube of Figure 9. The first thing to notice on the plane before flattening is that the panel drifts from dark to light in place to place. This is because the horizontal layers are not fully horizontal. Approaching the dome the change from dark to light and back again happens so rapidly that the dome appears surrounded by rings. After flattening, the drift and rings disappear. The reflection horizons are no longer cutting across the image. Channels no longer drift off (above or below) the viewable time slice. Carefully viewing the salt dome it seems smaller

after flattening because the rings are replace by a bedding plane.

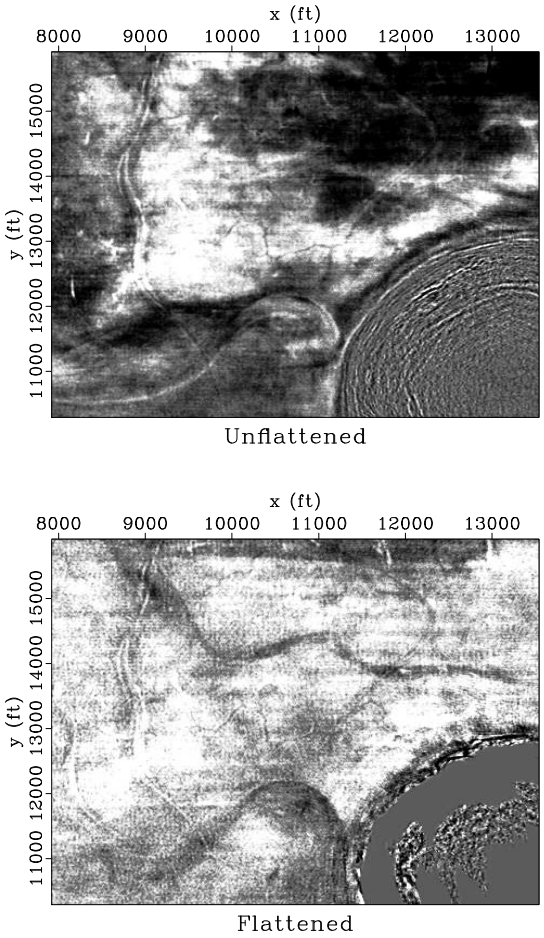


Figure 10: Slices of constant time before and after flattening. Notice the rings surrounding the dome are gone giving the dome a reduced diameter. (Ignore the inside of the dome.)

VESUVIUS PHASE UNWRAPPING

Figure 11 shows radar images of Mt. Vesuvius[1] in Italy. These images are made from backscatter signals $s_1(t)$ and $s_2(t)$, recorded along two **satellite orbit**s 800-km high and 54-m apart. The signals are very high frequency (the radar wavelength being 2.7 cm). The signals were Fourier transformed and one multiplied by the complex conjugate of the other, getting the product $Z = S_1(\omega)\bar{S}_2(\omega)$. The product's amplitude and phase are shown in Figure 11. Examining the data, you can notice that where the signals are strongest (darkest on the left), the phase (on the right) is the most spatially consistent. Pixel by pixel evaluation with the two frames in a movie program shows that there are a few somewhat large local amplitudes (clipped in Figure 11) but because these generally have spatially consistent phase, I would not describe the data as containing noise bursts.

To reduce the time needed for analysis and printing, I reduced the data size two different ways, by decimation and local averaging, as shown in Figure 12. The decimation was to about 1 part in 9 on each axis, and the local averaging was done in 9×9 windows giving the same spatial resolution in each case. The local averaging was done independently in

[1] A web search engine quickly finds you other views.

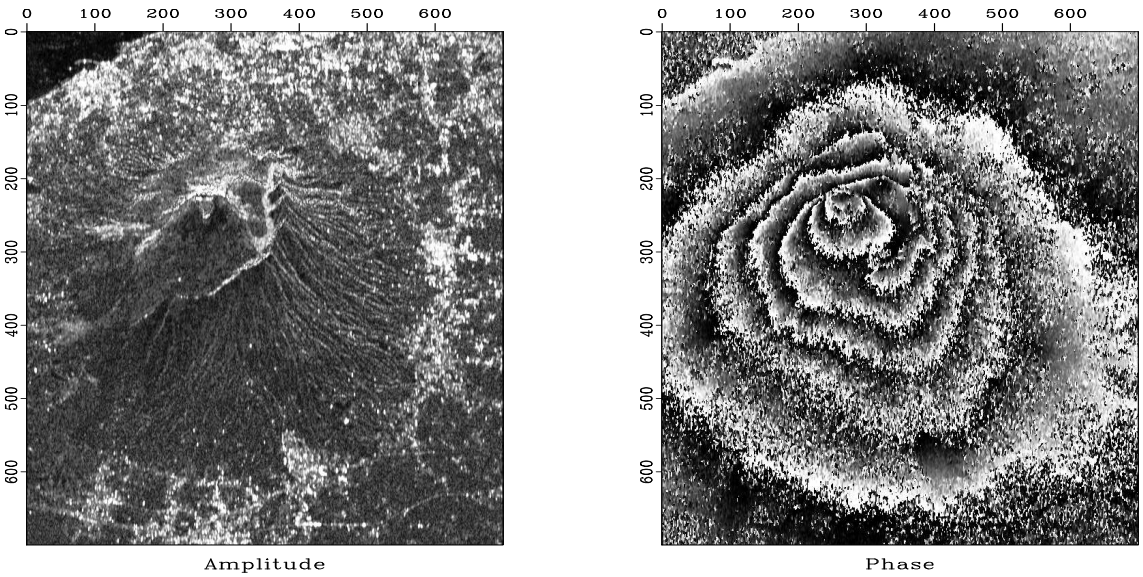


Figure 11: Radar image of Mt. Vesuvius. Left is the amplitude $|Z(x,y)|$. Nonreflecting ocean in upper-left corner. Right is the phase $\arctan(\Re Z(x,y), \Im Z(x,y)\ )$. (European Space Agency via Umberto Spagnolini)

the plane of the real part and the plane of the imaginary part. On the smoothed data the phase is less noisy.

From Figures 11 and 12, we see that contours of constant phase appear to be contours of constant altitude; this conclusion leads us to suppose that a study of radar theory would lead us to a relation like $Z(x,y) = e^{ih(x,y)}$, where $h(x,y)$ is altitude. We nonradar specialists often think of phase in $e^{i\phi} = e^{i\omega t_0(x,y)}$ as being caused by some time delay and being defined for some constant frequency ω. Knowledge of this ω (as well as some angle parameters) would define the physical units of $h(x,y)$.

Because the flat land away from the mountain is all at the same phase (as is the altitude), the distance as revealed by the phase does not represent the distance from the ground to the satellite viewer. We are accustomed to measuring altitude along a vertical line to a datum; but here, the distance seems to be measured from the ground along a 23° angle from the vertical to a datum at the satellite height.

Phase is a troublesome measurement, because we generally see it modulo 2π. Marching up the mountain, we see the phase getting lighter and lighter until it suddenly jumps to black, which then continues to lighten as we continue up the mountain to the next jump. Let us undertake to compute the phase, including all its jumps of 2π. Begin with a complex number Z representing the complex-valued image at any location in the (x,y)-plane.

$$re^{i\phi} = Z \tag{84}$$

$$\ln|r| + i\phi = \ln Z \tag{85}$$

$$\phi(x,y) = \Im \ln Z(x,y) \ + \ 2\pi N(x,y) \tag{86}$$

Computers find the imaginary part of the logarithm with the arctan function of two arguments, `atan2(y,x)`, which puts the phase in the range $-\pi < \phi \leq \pi$, although any multiple

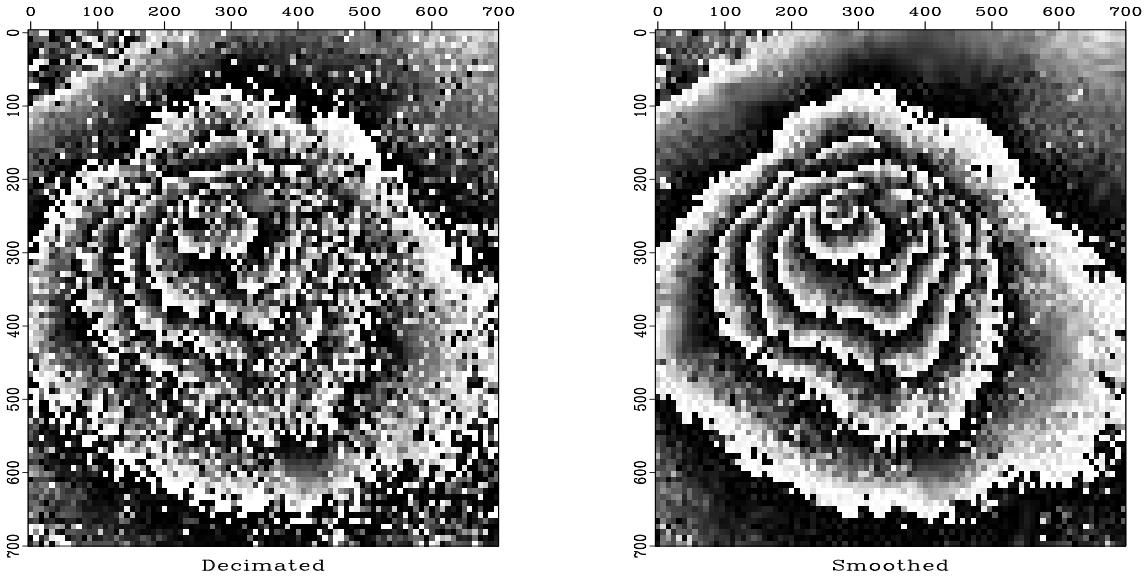


Figure 12: Phase based on decimated data (left) and smoothed data (right).

of 2π could be added. We seem to escape the $2\pi N$ phase ambiguity by differentiating:

$$\frac{\partial \phi}{\partial x} \quad = \quad \Im \frac{1}{Z} \frac{\partial Z}{\partial x} \quad = \quad \frac{\Im \bar{Z} \frac{\partial Z}{\partial x}}{\bar{Z} Z} \tag{87}$$

For every point on the y-axis, equation (87) is a differential equation on the x-axis. We could integrate them all to find $\phi(x, y)$. That sounds easy. On the other hand, the same equations are valid when x and y are interchanged, therefore we get twice as many equations as unknowns. Ideally either of these sets of equations is equivalent to the other; but for real data, we expect to be fitting this fitting goal:

$$\nabla \phi \quad \approx \quad \frac{\Im \bar{Z} \nabla Z}{\bar{Z} Z} \tag{88}$$

where $\nabla = (\frac{\partial}{\partial x}, \frac{\partial}{\partial y})$. Mathematically, computing phase this way is like our previous seismic flattening with $\nabla \tau \approx \mathbf{d}$. Taking measurements to be phase differences between neighboring mesh points, it is more correct to interpret Equation (88) as a difference equation than a differential equation. Because we measure phase differences only over tiny distances (one pixel), we hope not to worry about phases greater than 2π. But, if such jumps do occur, the jumps contribute to overall error.

Let us consider a typical location in the (x, y) plane where the complex numbers $Z_{i,j}$ are given. Define a shorthand a, b, c, and d as follows:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad = \quad \begin{bmatrix} Z_{i,j} & Z_{i,j+1} \\ Z_{i+1,j} & Z_{i+1,j+1} \end{bmatrix} \tag{89}$$

With this shorthand, the difference equation representation of the fitting goal (88) is:

$$\begin{array}{rcl} \phi_{i+1,j} - \phi_{i,j} & \approx & \Delta \phi_{ac} \\ \phi_{i,j+1} - \phi_{i,j} & \approx & \Delta \phi_{ab} \end{array} \tag{90}$$

Now, let us find the phase jumps between the various locations. Complex numbers a and b may be expressed in polar form, say $a = r_a e^{i\phi_a}$ and $b = r_b e^{i\phi_b}$. The complex number $\bar{a}b = r_a r_b e^{i(\phi_b - \phi_a)}$ has the desired phase $\Delta\phi_{ab}$. To obtain it we take the imaginary part of the complex logarithm $\ln |r_a r_b| + i\Delta\phi_{ab}$:

$$\begin{array}{rcccl} \phi_b - \phi_a & = & \Delta\phi_{ab} & = & \Im \ln \bar{a}b \\ \phi_d - \phi_c & = & \Delta\phi_{cd} & = & \Im \ln \bar{c}d \\ \phi_c - \phi_a & = & \Delta\phi_{ac} & = & \Im \ln \bar{a}c \\ \phi_d - \phi_b & = & \Delta\phi_{bd} & = & \Im \ln \bar{b}d \end{array} \tag{91}$$

which gives the information needed to fill in the right side of (90), as done by subroutine `igrad2init()` from module `unwrap` on this page.

The operator needed is `igrad2`, gradient with its adjoint, the divergence.

Estimating the inverse gradient

To optimize the fitting goal (90), module `unwrap()` uses the conjugate-direction method like the modules `cgmeth()` and `invstack()`: An open question is whether or not the required

user/gee/unwrap.c

```
65      sf_igrad2_init(n1,n2);
66      sf_tinysolver(sf_igrad2_lop,sf_cgstep,
67                    n1*n2,n1*n2*2,hh,NULL,rt[0][0],niter);
```

number of iterations is reasonable or if we need to uncover a preconditioner or more rapid solution method. I adjusted the frame size (by the amount of smoothing in Figure 12) so that I would get the solution in about ten seconds with 400 iterations. Results are shown in Figure 13.

Revise `igrad2` to make a module called `tgrad2` which has transient boundaries.

Analytical solutions

We have found a numerical solution to fitting applications, such as:

$$\mathbf{0} \quad \approx \quad \nabla\tau \; - \; \mathbf{d} \tag{92}$$

An analytical solution is much faster. From any regression, we get the least squares solution when we multiply by the transpose of the operator. Thus,

$$\mathbf{0} \quad = \quad \nabla^{\mathrm{T}} \, \nabla\tau \; - \; \nabla^{\mathrm{T}} \, \mathbf{d} \tag{93}$$

We need to understand what is the transpose of the gradient operator. Recall the finite difference representation of a derivative in Chapter 1. Ignoring end effects, the transpose of a derivative is the negative of a derivative. Because the transpose of a column vector is a row vector, the adjoint of a gradient ∇, namely, ∇^{T} is more commonly known as the vector divergence $(\nabla\cdot)$. Likewise, $\nabla^{\mathrm{T}} \, \nabla$ is a positive definite matrix, the negative of the Laplacian

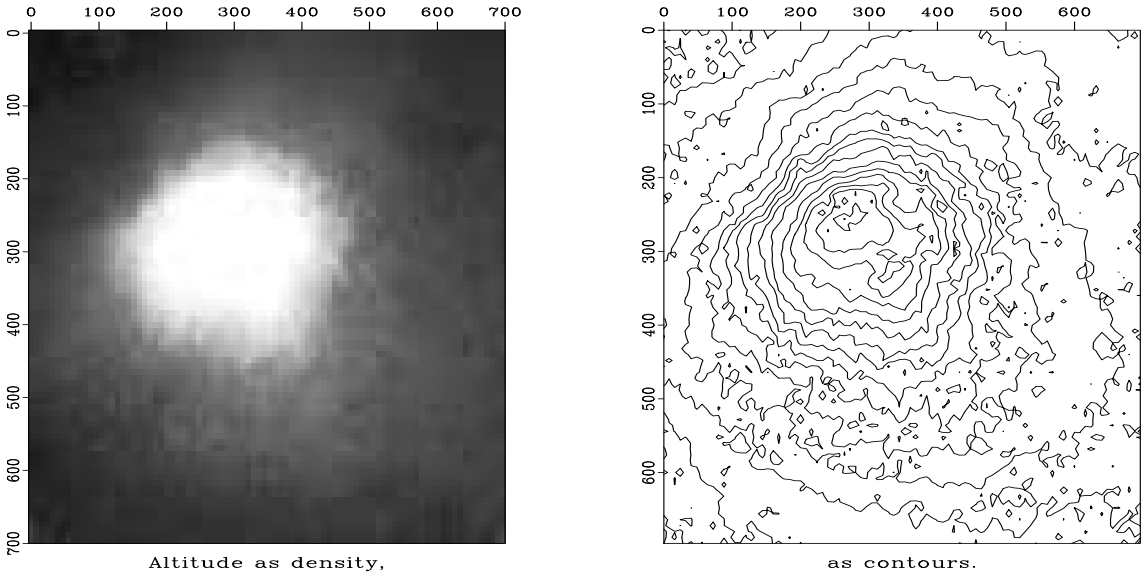


Figure 13: Estimated altitude.

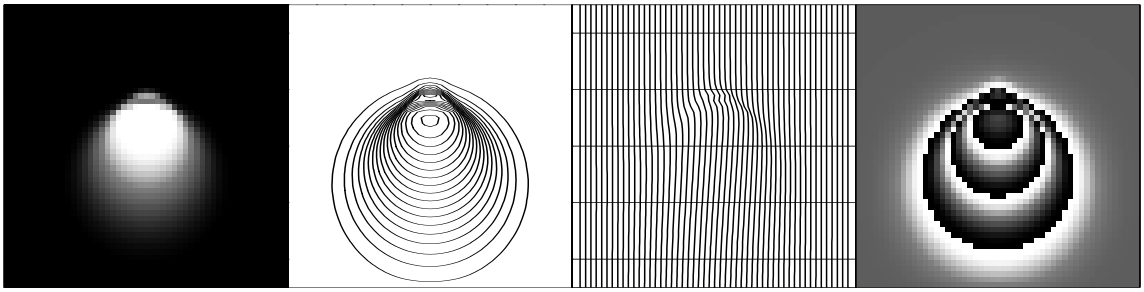


Figure 14: Synthetic mountain with hidden backside. For your estimation enjoyment.

∇^2. Thus, in more conventional mathematical notation, the solution τ is that of Poisson's equation.

$$\nabla^2 \tau \quad = \quad -\nabla \cdot \mathbf{d} \tag{94}$$

In the Fourier domain, we can have an analytic solution. There, $-\nabla^2 = k_x^2 + k_y^2$ where (k_x, k_y) are the Fourier frequencies on the (x, y) axes. Instead of thinking of equation (94) as a convolution in physical space, think of it as a product in Fourier space. Thus, the analytic solution is:

$$\tau(x,y) \quad = \quad \mathbf{FT}^{-1} \; \frac{\mathbf{FT} \;\; \nabla \cdot \mathbf{d}}{k_x^2 + k_y^2} \tag{95}$$

where **FT** denotes two-dimensional Fourier transform over x and y . Here is a trick from numerical analysis that gives better results: Instead of representing the denominator $k_x^2 + k_y^2$ in the most obvious way, let us represent it in a manner consistent with the finite-difference way we expressed the numerator $\nabla \cdot \mathbf{d}$. Recall that $-i\omega\Delta t \approx i\hat{\omega}\Delta t = 1 - Z = 1 - \exp(-i\omega\Delta t)$, which is a Fourier domain way of saying that difference equations tend to differential equations at low frequencies. Likewise, a symmetric second time derivative has a finite-difference representation proportional to $(-2 + Z + 1/Z)$ and in a two-dimensional space, a finite-difference representation of the Laplacian operator is proportional to $(-4 + X + 1/X + Y + 1/Y)$, where $X = \exp(ik_x\Delta x)$ and $Y = \exp(ik_y\Delta y)$. Fourier solutions have peculiarities (periodic boundary conditions) that are not always appropriate in practice, but having these solutions available is often a nice place to start from when solving an application that cannot be solved in Fourier space.

For example, suppose we feel some data values are bad, and we would like to throw out the regression equations involving the bad data points. At Vesuvius, we might consider the strength of the radar return (which we have previously ignored) and use it as a weighting function **W**. Now, our regression (92) becomes:

$$\mathbf{0} \quad \approx \quad \mathbf{W} \; (\nabla\phi - \mathbf{d}) \quad = \quad (\mathbf{W}\nabla)\phi \; - \; \mathbf{W}\mathbf{d} \tag{96}$$

which is a regression with an operator $\mathbf{W}\nabla$ and data $\mathbf{W}\mathbf{d}$. The weighted problem is not solvable in the Fourier domain, because the operator $(\mathbf{W}\nabla)^{\mathrm{T}} \; \mathbf{W}\nabla$ has no simple expression in the Fourier domain. Thus, we would use the analytic solution to the unweighted problem as a starting guess for the iterative solution to the real problem.

With the Vesuvius data, we could construct a weight **W** from the signal strength. We also have available the curl, which should vanish. Vanishing is an indicator of questionable data that should be weighted down relative to other data.

OPERATOR SCALING (BINORMALIZATION)

We can accept model **m** and data **d** as they arise from the geometry or geophysics of an application, or we can transform them to forms that are computationally convenient, work with them, and finally transform back. Say we transform them to new variables, **u** and **v**.

$$\mathbf{u} \quad = \quad \mathbf{M}\mathbf{m} \tag{97}$$

$$\mathbf{v} \quad = \quad \mathbf{D}\mathbf{d} \tag{98}$$

These transformations change the $\mathbf{F}$ operator to $\mathbf{DFM}^{-1}$ because

$$\mathbf{d} = \mathbf{Fm} \tag{99}$$
$$\mathbf{Dd} = \mathbf{DFM}^{-1}\mathbf{Mm} \tag{100}$$
$$\mathbf{v} = \mathbf{DFM}^{-1}\mathbf{u} \tag{101}$$

The game is looking for the best $\mathbf{M}$ and $\mathbf{D}$.

If I were able and willing to handle linear algebra in a modern way, I would show you this matrix iteration

$$\lambda \begin{bmatrix} \mathbf{d} \\ \mathbf{m} \end{bmatrix}_{i+1} = \begin{bmatrix} \mathbf{0} & \mathbf{F} \\ \mathbf{F}^{\mathrm{T}} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{d} \\ \mathbf{m} \end{bmatrix}_{i} \tag{102}$$

What is λ? It is a scale factor that you use to keep the vectors normalized as the iteration proceeds. You may recognize a path leading to eigenvectors, eigenvalues, Hermitian matrices, and singular-value decomposition. You'll need to find other sources to go further on that path because it has not led me to a solution to the problem at hand which is how to choose the best $\mathbf{M}$ and $\mathbf{D}$.

If you have an operator that you are using millions of times it is worth seeking good choices. Good choices are those that make the adjoint of your new operator $\mathbf{DFM}^{-1}$ a good approximation to its inverse. These are the two conditions we seek:

$$\mathbf{I} \approx (\mathbf{DFM}^{-1})^{\mathrm{T}} (\mathbf{DFM}^{-1}) \tag{103}$$
$$\mathbf{I} \approx (\mathbf{DFM}^{-1}) (\mathbf{DFM}^{-1})^{\mathrm{T}} \tag{104}$$

If these were true, we could probe with any test vectors we wished

$$\hat{\mathbf{t}}_m = (\mathbf{M}^{-1})^{\mathrm{T}} \text{ (something) } \mathbf{M}^{-1}\, \mathbf{t}_{\mathrm{m}} \tag{105}$$
$$\hat{\mathbf{t}}_d = \mathbf{D} \text{ (something else) } \mathbf{D}^{\mathrm{T}}\, \mathbf{t}_d \tag{106}$$

and find both $\hat{\mathbf{t}}_m \approx \mathbf{t}_m$ and $\hat{\mathbf{t}}_d \approx \mathbf{t}_d$. So, the game is to play with $\mathbf{M}$ and $\mathbf{D}$ to try to get this to happen.

About the only trick I know is to try $\mathbf{M}$ and $\mathbf{D}$ as diagonal matrices. For test functions $\mathbf{t}$, I generally use a pattern of moderately spaced impulses. In physical space we may see places where $\hat{\mathbf{t}}$ is smaller than $\mathbf{t}$. Those are the places to boost the corresponding diagonal.

There are many test functions you could use. You could use all ones. You could use random numbers. You could use a pile of random old data, though I'm not sure what you would use for old models. Take the output. Take its absolute value. Maybe smooth it. Take the square root since the half you put in $\mathbf{F}$ appears a second time in $\mathbf{F}^{\mathrm{T}}$.

I know one more trick. In seismology many operators appear as integrals. One of many such operators is called "Kirchhoff migration". Because these operators and their adjoints contain integrations they boost low frequencies. We can attenuate them back to their original size by having $\mathbf{M}$ or $\mathbf{D}$ apply $\sqrt{-i\omega}$ (known in the time domain as the "Hankel tail").

What, may we ask is the interpretation of the $(\mathbf{u}, \mathbf{v})$ variables? They feel like "energy conservation" variables, though it makes no sense to say the physical energy in $\mathbf{m}$ or $\mathbf{d}$ should be conserved in the way of Parseval's theorem of Fourier transforms. I imagined the $(\mathbf{u}, \mathbf{v})$ variables might be especially suitable for display (like preconditioned variables) but now I am less certain.

THE WORLD OF CONJUGATE GRADIENTS

Nonlinearity arises in two ways: First, modeled data might be a nonlinear function of the model parameters. Second, observed data could contain imperfections that force us to use **nonlinear methods** of statistical estimation.

Physical nonlinearity

Methods of physics may relate modeled data $\mathbf{d}_{\text{theor}}$ to model parameters $\mathbf{m}$, with a nonlinear relation, say $\mathbf{d}_{\text{theor}} = \mathbf{f}(\mathbf{m})$. The power-series approach then leads to representing modeled data as:

$$\mathbf{d}_{\text{theor}} \quad = \quad \mathbf{f}(\mathbf{m}_0 + \Delta\mathbf{m}) \quad \approx \quad \mathbf{f}(\mathbf{m}_0) + \mathbf{F}\Delta\mathbf{m} \tag{107}$$

where $\mathbf{F}$ is the matrix of partial derivatives of data values by model parameters, say $\partial d_i/\partial m_j$, evaluated at $\mathbf{m}_0$. The modeled data $\mathbf{d}_{\text{theor}}$ minus the observed data $\mathbf{d}_{\text{obs}}$ is the residual we minimize.

$$\mathbf{0} \quad \approx \quad \mathbf{d}_{\text{theor}} - \mathbf{d}_{\text{obs}} \quad = \quad \mathbf{F}\boldsymbol{\Delta}\mathbf{m} + [\mathbf{f}(\mathbf{m}_0) - \mathbf{d}_{\text{obs}}] \tag{108}$$

$$\mathbf{r}_{\text{new}} \quad = \quad \mathbf{F}\boldsymbol{\Delta}\mathbf{m} + \mathbf{r}_{\text{old}} \tag{109}$$

It is worth noticing that the residual updating (109) in a nonlinear application is the same as that in a linear application (55). If you make a large step $\Delta\mathbf{m}$, however, the new residual is different from that expected by (109). Thus, you should always re-evaluate the residual vector at the new location, and if you are reasonably cautious, you should be sure the residual norm has actually decreased before you accept a large step.

The pathway of inversion with physical nonlinearity is well developed in the academic literature, and Bill **Symes** at Rice University has a particularly active group.

There are occasions to change the weighting function during model fitting. Then, one simply restarts the calculation from the current model. In the code, you would flag a restart with the expression `forget=true`.

Coding nonlinear fitting problems

An adaptation of a linear method gives us a nonlinear solver by simply recomputing the gradient at each iteration. Omitting the weighting function (for simplicity) the **template** is:

iterate {
$\mathbf{r} \quad \longleftarrow \quad \mathbf{f}(\mathbf{m}) - \mathbf{d}$
Define $\mathbf{F} = \partial\mathbf{d}/\partial\mathbf{m}$.
$\Delta\mathbf{m} \quad \longleftarrow \quad \mathbf{F}^{\mathrm{T}}\ \mathbf{r}$
$\Delta\mathbf{r} \quad \longleftarrow \quad \mathbf{F}\ \Delta\mathbf{m}$
$(\mathbf{m}, \mathbf{r}) \quad \longleftarrow \quad \text{step}(\mathbf{m}, \mathbf{r}, \Delta\mathbf{m}, \Delta\mathbf{r})$
}

A formal theory for the optimization exists, but we are not using it here. The assumption we make is that the step size is small, so that familiar line-search and plane-search

approximations can succeed in reducing the residual. Unfortunately, this assumption is not reliable. What we should do is test that the residual really does decrease, and if it does not, we should revert to smaller step size. Perhaps, we should test an incremental variation on the status quo: where inside `solver`, we check to see if the residual diminished in the *previous* step; and if it did not, restart the iteration (choose the *current* step to be steepest descent instead of CD).

Experience shows that nonlinear applications have many pitfalls. Start with a linear problem, add a minor physical improvement or abnormal noise, and the problem becomes nonlinear and probably has another solution far from anything reasonable. When solving such a nonlinear problem, we cannot arbitrarily begin from zero, as we do with linear problems. We must choose a reasonable starting guess. Chapter **??** on the topic of regularization offers an additional way to reduce the dangers of nonlinearity.

Inverse of a 2×2 matrix

$$\mathbf{A}^{-1} \qquad \mathbf{A} \qquad = \qquad \mathbf{I} \tag{110}$$

$$\frac{1}{ad-bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \quad \begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad = \quad \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{111}$$

Standard methods

The conjugate-direction method is really a family of methods. Mathematically, where there are n unknowns, these algorithms all converge to the answer in n (or fewer) steps. The various methods differ in numerical accuracy, treatment of underdetermined systems, accuracy in treating ill-conditioned systems, space requirements, and numbers of dot products. Technically, the method of CD used in the `cgstep` module is not the conjugate-gradient method itself, but is equivalent to it. This method is more properly called the **conjugate-direction method** with a memory of one step. I chose this method for its clarity and flexibility. If you would like a free introduction and summary of conjugate-gradient methods, I particularly recommend *An Introduction to Conjugate Gradient Method Without Agonizing Pain* by Jonathon Shewchuk, which you can download[2].

I suggest you skip over the remainder of this section and return after you have seen many examples and have developed some expertise, and have some technical problems.

The **conjugate-gradient method** was introduced by **Hestenes** and **Stiefel** in 1952. To read the standard literature and relate it to this book, you should first realize that when I write fitting goals like

$$0 \quad \approx \quad \mathbf{W}(\mathbf{F}\mathbf{m} - \mathbf{d}) \tag{112}$$

$$0 \quad \approx \quad \mathbf{A}\mathbf{m}, \tag{113}$$

they are equivalent to minimizing the quadratic form:

$$\mathbf{m}: \qquad \min_{\mathbf{m}} Q(\mathbf{m}) \quad = \quad (\mathbf{m}^{\mathrm{T}}\,\mathbf{F}^{\mathrm{T}} - \mathbf{d}^{\mathrm{T}})\mathbf{W}^{\mathrm{T}}\,\mathbf{W}(\mathbf{F}\mathbf{m} - \mathbf{d}) \;+\; \mathbf{m}^{\mathrm{T}}\,\mathbf{A}^{\mathrm{T}}\,\mathbf{A}\mathbf{m} \tag{114}$$

[2] http://www.cs.cmu.edu/afs/cs/project/quake/public/papers/painless-conjugate-gradient.ps

The optimization theory (OT) literature starts from a minimization of

$$\mathbf{x}: \qquad \min_{\mathbf{x}} Q(\mathbf{x}) \quad = \quad \mathbf{x}^{\mathrm{T}}\,\mathbf{H}\mathbf{x} - \mathbf{b}^{\mathrm{T}}\,\mathbf{x} \tag{115}$$

To relate equation (114) to (115) we expand the parentheses in (114) and abandon the constant term $\mathbf{d}^{\mathrm{T}}\,\mathbf{d}$. Then gather the quadratic term in $\mathbf{m}$ and the linear term in $\mathbf{m}$. There are two terms linear in $\mathbf{m}$ that are transposes of each other. They are scalars so they are equal. Thus, to invoke "standard methods," you take your problem-formulation operators $\mathbf{F}$, $\mathbf{W}$, $\mathbf{A}$ and create two subroutines that apply:

$$\mathbf{H} \quad = \quad \mathbf{F}^{\mathrm{T}}\,\mathbf{W}^{\mathrm{T}}\,\mathbf{W}\mathbf{F} + \mathbf{A}^{\mathrm{T}}\,\mathbf{A} \tag{116}$$

$$\mathbf{b}^{\mathrm{T}} \quad = \quad 2(\mathbf{F}^{\mathrm{T}}\,\mathbf{W}^{\mathrm{T}}\,\mathbf{W}\mathbf{d})^{\mathrm{T}} \tag{117}$$

The operators $\mathbf{H}$ and $\mathbf{b}^{\mathrm{T}}$ operate on model space. Standard procedures do not require their adjoints because $\mathbf{H}$ is its own adjoint and $\mathbf{b}^{\mathrm{T}}$ reduces model space to a scalar. You can see that computing $\mathbf{H}$ and $\mathbf{b}^{\mathrm{T}}$ requires one temporary space the size of data space (whereas `cgstep` requires two).

When people have trouble with conjugate gradients or conjugate directions, I always refer them to the **Paige and Saunders algorithm** `LSQR`. Methods that form $\mathbf{H}$ explicitly or implicitly (including both the standard literature and the book3 method) square the condition number, that is, they are twice as susceptible to rounding error as is `LSQR`. The Paige and Saunders method is reviewed by Nolet in a geophysical context.

EXERCISES:

1 It is possible to reject two dips with the operator:

$$(\partial_x + p_1\partial_t)(\partial_x + p_2\partial_t) \tag{118}$$

This is equivalent to:

$$\left(\frac{\partial^2}{\partial x^2} + a\frac{\partial^2}{\partial x \partial t} + b\frac{\partial^2}{\partial t^2}\right) u(t,x) \quad = \quad v(t,x) \quad \approx \quad 0 \tag{119}$$

where u is the input signal, and v is the output signal. Show how to solve for a and b by minimizing the energy in v.

2 Given a and b from the previous exercise, what are p_1 and p_2?

3 Reduce $\mathbf{d} = \mathbf{F}\mathbf{m}$ to the special case of one data point and two model points like this:

$$d \quad = \quad \begin{bmatrix} 2 & 1 \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \end{bmatrix} \tag{120}$$

What is the null space?

4 In 1695, 150 years before Lord Kelvin's absolute temperature scale, 120 years before Sadi Carnot's PhD. thesis, 40 years before Anders Celsius, and 20 years before Gabriel Fahrenheit, the French physicist Guillaume Amontons, deaf since birth, took a mercury manometer (pressure gauge) and sealed it inside a glass pipe (a constant volume of air).

He heated it to the boiling point of water at 100°C. As he lowered the temperature to freezing at 0°C, he observed the pressure dropped by 25% . He could not drop the temperature any further, but he supposed that if he could drop it further by a factor of three, the pressure would drop to zero (the lowest possible pressure), and the temperature would have been the lowest possible temperature. Had he lived after Anders Celsius, he might have calculated this temperature to be −300°C (Celsius). Absolute zero is now known to be −273°C.

It is your job to be Amontons' lab assistant. You make your i-th measurement of temperature T_i with Issac Newton's thermometer; and you measure pressure P_i and volume V_i in the metric system. Amontons needs you to fit his data with the regression $0 \approx \alpha(T_i - T_0) - P_i V_i$ and calculate the temperature shift T_0 that Newton should have made when he defined his temperature scale. Do not solve this problem! Instead, cast it in the form of equation (**??**), identifying the data d and the two column vectors f_1 and f_2 that are the fitting functions. Relate the model parameters x_1 and x_2 to the physical parameters α and T_0 . Suppose you make ALL your measurements at room temperature, can you find T_0 ? Why or why not?

5 One way to remove a mean value m from signal $s(t) = \mathbf{s}$ is with the fitting goal $\mathbf{0} \approx \mathbf{s} - m$. What operator matrix is involved?

6 What linear operator subroutine from Chapter **??** can be used for finding the mean?

7 How many CD iterations should be required to get the exact mean value?

8 Write a mathematical expression for finding the mean by the CG method.