Chapter 2

Model fitting by least squares

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

2.1 HOW TO DIVIDE NOISY SIGNALS

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.

2.1.1 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 way, and it 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,

$$X(\omega) \quad = \quad \frac{\overline{F(\omega)}\; Y(\omega)}{\overline{F(\omega)}F(\omega) \ + \ \epsilon^2} \tag{2.1}$$

Now the denominator must always be a positive number greater than zero, so division is always safe. Equation (1.1) 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.

2.1.2 Damped solution

Equation (1.1) is the solution to an optimization problem that arises in many applications. Now that we know the solution, let us formally define the problem. First, we will solve a simpler problem with real values: we will choose to minimize the **quadratic function** of x:

$$Q(x) \quad = \quad (fx - y)^2 + \epsilon^2 x^2 \tag{2.2}$$

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

$$0 \quad = \quad f(fx - y) + \epsilon^2 x \tag{2.3}$$

This yields the earlier answer $x = fy/(f^2 + \epsilon^2)$.

With Fourier transforms, the signal X is a complex number at each frequency ω. So we generalize equation (1.2) to

$$Q(\bar{X}, X) \quad = \quad (\overline{FX - Y})(FX - Y) + \epsilon^2 \bar{X}X \quad = \quad (\bar{X}\bar{F} - \bar{Y})(FX - Y) + \epsilon^2 \bar{X}X \tag{2.4}$$

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 will 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 $\partial Q/\partial \bar{X}$:

$$\frac{\partial Q(\bar{X}, X)}{\partial \bar{X}} \quad = \quad \bar{F}(FX - Y) + \epsilon^2 X \quad = \quad 0 \tag{2.5}$$

The derivative $\partial Q/\partial X$ is the complex conjugate of $\partial Q/\partial \bar{X}$. So if either is zero, the other is too. 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 (1.5) for X gives equation (1.1).

Equation (1.1) 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.

2.1.3 Formal path to the low-cut filter

This book defines many geophysical estimation problems. Many of them amount to statement of 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 or too wiggly. We will 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.

$$0 \quad \approx \quad -i\omega(m - d) \tag{2.6}$$

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 this, we need the second goal, a model residual which is to be minimized. We need a small number ϵ. The model goal is

$$0 \quad \approx \quad \epsilon\, m \tag{2.7}$$

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

$$Q(m) \quad = \quad \omega^2 (m - d)^2 + \epsilon^2 m^2 \tag{2.8}$$

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

$$0 \quad = \quad \frac{dQ}{dm} \quad = \quad 2\omega^2 (m - d) + 2\epsilon^2 m \tag{2.9}$$

or

$$m \quad = \quad \frac{\omega^2}{\omega^2 + \epsilon^2}\, d \tag{2.10}$$

which is the polarity preserving low-cut filter we found less formally earlier, equation (1.24).

2.1.4 How much damping?

Of some curiosity and significance is the numerical choice of ϵ. The general theory says we need an epsilon, but does not say how much. Our low-pass filter approach in Chapter 1 made it quite clear that ϵ is a filter cutoff which might better have been named ω_0. We experimented with some objective tests for the correct value of ω_0, a subject that we will return to later.

2.2 MULTIVARIATE LEAST SQUARES

2.2.1 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 could itself be thought of as 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** when their dot product vanishes: $\mathbf{p} \cdot \mathbf{s} = 0$. Orthogonal vectors are well known in physical space; we will also encounter them 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 x_1 and x_2 should be chosen so that the model matches the data; i.e.,

$$\mathbf{d} \quad \approx \quad \mathbf{f}_1 x_1 + \mathbf{f}_2 x_2 \tag{2.11}$$

Notice that we could take the partial derivative of the data in (1.11) with respect to an unknown, say x_1, and the result is the regressor $\mathbf{f}_1$.

> The **partial derivative** of all theoretical data with respect to any model parameter gives a **regressor**. A **regressor** is a column in the matrix of partial-derivatives, $\partial d_i / \partial m_j$.

The fitting goal (1.11) is often expressed in the more compact mathematical matrix notation $\mathbf{d} \approx \mathbf{F}\mathbf{x}$, but in our derivation here we will keep track of each component explicitly and use mathematical matrix notation to summarize the final result. Fitting the observed data $\mathbf{d} = \mathbf{d}^{\rm obs}$ to its two theoretical parts $\mathbf{f}_1 x_1$ and $\mathbf{f}_2 x_2$ can be expressed as minimizing the length of the residual vector $\mathbf{r}$, where

$$\mathbf{0} \quad \approx \quad \mathbf{r} \quad = \quad \mathbf{d}^{\rm theor} - \mathbf{d}^{\rm obs} \tag{2.12}$$

$$\mathbf{0} \quad \approx \quad \mathbf{r} \quad = \quad \mathbf{f}_1 x_1 + \mathbf{f}_2 x_2 \; - \; \mathbf{d} \tag{2.13}$$

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

$$Q(x_1, x_2) = \mathbf{r} \cdot \mathbf{r} \tag{2.14}$$
$$= (\mathbf{f}_1 x_1 + \mathbf{f}_2 x_2 - \mathbf{d}) \cdot (\mathbf{f}_1 x_1 + \mathbf{f}_2 x_2 - \mathbf{d}) \tag{2.15}$$

To find the gradient of the quadratic form $Q(x_1, x_2)$, 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

$$\frac{d}{dx} boldr \cdot \mathbf{r} = \frac{d\mathbf{r}}{dx} \cdot \mathbf{r} + \mathbf{r} \cdot \frac{d\mathbf{r}}{dx} \tag{2.16}$$

Thus, the gradient of $Q(x_1, x_2)$ is defined by its two components:

$$\frac{\partial Q}{\partial x_1} = \mathbf{f}_1 \cdot (\mathbf{f}_1 x_1 + \mathbf{f}_2 x_2 - \mathbf{d}) + (\mathbf{f}_1 x_1 + \mathbf{f}_2 x_2 - \mathbf{d}) \cdot \mathbf{f}_1 \tag{2.17}$$
$$\frac{\partial Q}{\partial x_2} = \mathbf{f}_2 \cdot (\mathbf{f}_1 x_1 + \mathbf{f}_2 x_2 - \mathbf{d}) + (\mathbf{f}_1 x_1 + \mathbf{f}_2 x_2 - \mathbf{d}) \cdot \mathbf{f}_2 \tag{2.18}$$

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

$$(\mathbf{f}_1 \cdot \mathbf{d}) = (\mathbf{f}_1 \cdot \mathbf{f}_1) x_1 + (\mathbf{f}_1 \cdot \mathbf{f}_2) x_2 \tag{2.19}$$
$$(\mathbf{f}_2 \cdot \mathbf{d}) = (\mathbf{f}_2 \cdot \mathbf{f}_1) x_1 + (\mathbf{f}_2 \cdot \mathbf{f}_2) x_2 \tag{2.20}$$

We can use these two equations to solve for the two unknowns x_1 and x_2. Writing this expression in matrix notation, we have

$$\begin{bmatrix} (\mathbf{f}_1 \cdot \mathbf{d}) \\ (\mathbf{f}_2 \cdot \mathbf{d}) \end{bmatrix} = \begin{bmatrix} (\mathbf{f}_1 \cdot \mathbf{f}_1) & (\mathbf{f}_1 \cdot \mathbf{f}_2) \\ (\mathbf{f}_2 \cdot \mathbf{f}_1) & (\mathbf{f}_2 \cdot \mathbf{f}_2) \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{2.21}$$

It is customary to use matrix notation without dot products. To do this, 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

$$\mathbf{F} = [\mathbf{f}_1 \quad \mathbf{f}_2] = \begin{bmatrix} f_{11} & f_{12} \\ f_{21} & f_{22} \\ f_{31} & f_{32} \end{bmatrix} \tag{2.22}$$

Likewise, the *transposed* matrix $\mathbf{F}'$ is defined by

$$\mathbf{F}' = \begin{bmatrix} f_{11} & f_{21} & f_{31} \\ f_{12} & f_{22} & f_{32} \end{bmatrix} \tag{2.23}$$

The matrix in equation (1.21) contains dot products. Matrix multiplication is an abstract way of representing the dot products:

$$\begin{bmatrix} (\mathbf{f}_1 \cdot \mathbf{f}_1) & (\mathbf{f}_1 \cdot \mathbf{f}_2) \\ (\mathbf{f}_2 \cdot \mathbf{f}_1) & (\mathbf{f}_2 \cdot \mathbf{f}_2) \end{bmatrix} = \begin{bmatrix} f_{11} & f_{21} & f_{31} \\ f_{12} & f_{22} & f_{32} \end{bmatrix} \begin{bmatrix} f_{11} & f_{12} \\ f_{21} & f_{22} \\ f_{31} & f_{32} \end{bmatrix} \tag{2.24}$$

Thus, equation (1.21) without dot products is

$$\begin{bmatrix} f_{11} & f_{21} & f_{31} \\ f_{12} & f_{22} & f_{32} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix} \quad = \quad \begin{bmatrix} f_{11} & f_{21} & f_{31} \\ f_{12} & f_{22} & f_{32} \end{bmatrix} \begin{bmatrix} f_{11} & f_{12} \\ f_{21} & f_{22} \\ f_{31} & f_{32} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{2.25}$$

which has the matrix abbreviation

$$\mathbf{F}'\mathbf{d} \quad = \quad (\mathbf{F}'\,\mathbf{F})\mathbf{x} \tag{2.26}$$

Equation (1.26) 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 (1.26) leads to an **analytic solution** for $\mathbf{x}$ using an inverse matrix. To solve formally for the unknown $\mathbf{x}$, we premultiply by the inverse matrix $(\mathbf{F}'\,\mathbf{F})^{-1}$:

$$\mathbf{x} \quad = \quad (\mathbf{F}'\,\mathbf{F})^{-1}\,\mathbf{F}'\mathbf{d} \tag{2.27}$$

Equation (1.27) is the central result of **least-squares** theory. We see it everywhere.

In our first manipulation of matrix algebra, we move around some parentheses in (1.26):

$$\mathbf{F}'\mathbf{d} \quad = \quad \mathbf{F}'\,(\mathbf{F}\mathbf{x}) \tag{2.28}$$

Moving the parentheses implies a regrouping of terms or a reordering of a computation. You can verify the validity of moving the parentheses if you write (1.28) in full as the set of two equations it represents. Equation (1.26) led to the "analytic" solution (1.27). In a later section on conjugate directions, we will see that equation (1.28) expresses better than (1.27) the philosophy of iterative methods.

Notice how equation (1.28) invites us to cancel the matrix $\mathbf{F}'$ from each side. We cannot do that of course, because $\mathbf{F}'$ is not a number, nor is it a square matrix with an inverse. If you really want to cancel the matrix $\mathbf{F}'$, you may, but the equation is then only an approximation that restates our original goal (1.11):

$$\mathbf{d} \quad \approx \quad \mathbf{F}\mathbf{x} \tag{2.29}$$

A speedy problem solver might ignore the mathematics covering the previous page, study his or her application until he or she is able to write the **statement of goals** (1.29) = (1.11), premultiply by $\mathbf{F}'$, replace $\approx$ by $=$, getting (1.26), and take (1.26) to a simultaneous equation-solving program to get $\mathbf{x}$.

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 $\mathbf{r} = \mathbf{F}\mathbf{x} - \mathbf{d}$. Formally this is often written as:

$$\min_{\mathbf{x}} \quad ||\mathbf{F}\mathbf{x} - \mathbf{d}|| \tag{2.30}$$

The notation above 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.

2.2.2 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 this, we'll show that those two vectors are orthogonal. Energy minimum is found by

$$0 \quad = \quad \frac{\partial}{\partial x_2} \, \mathbf{r} \cdot \mathbf{r} \quad = \quad 2 \, \mathbf{r} \cdot \frac{\partial \mathbf{r}}{\partial x_2} \quad = \quad 2 \, \mathbf{r} \cdot \mathbf{f}_2 \tag{2.31}$$

(To compute the derivative refer to equation (1.13).) Equation (1.31) 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 (1.28) to emphasize the perpendicularity. Bring both terms to the left, and recall the definition of the residual $\mathbf{r}$ from equation (1.13):

$$\mathbf{F}'(\mathbf{F}\mathbf{x} - \mathbf{d}) \quad = \quad \mathbf{0} \tag{2.32}$$

$$\mathbf{F}'\mathbf{r} \quad = \quad \mathbf{0} \tag{2.33}$$

Equation (1.33) says that the **residual** vector $\mathbf{r}$ is perpendicular to each row in the $\mathbf{F}'$ matrix. These rows are the **fitting functions**. Therefore, the residual, after it has been minimized, is perpendicular to *all* the fitting functions.

2.2.3 Differentiation by a complex vector

Complex numbers frequently arise in physical problems, particularly those with Fourier series. Let us extend the multivariable least-squares theory to the use of complex-valued unknowns $\mathbf{x}$. First recall how complex numbers were handled with single-variable least squares; i.e., as in the discussion leading up to equation (1.5). Use a prime, such as $\mathbf{x}'$, to denote the complex conjugate of the transposed vector $\mathbf{x}$. Now write the positive **quadratic form** as

$$Q(\mathbf{x}', \mathbf{x}) \quad = \quad (\mathbf{F}\mathbf{x} - \mathbf{d})'(\mathbf{F}\mathbf{x} - \mathbf{d}) \quad = \quad (\mathbf{x}'\mathbf{F}' - \mathbf{d}')(\mathbf{F}\mathbf{x} - \mathbf{d}) \tag{2.34}$$

After equation (1.4), 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 they are conjugates of each other. Now take the derivative of Q with respect to the (possibly complex, row) vector $\mathbf{x}'$. Notice that $\partial Q/\partial \mathbf{x}'$ is the complex conjugate transpose of $\partial Q/\partial \mathbf{x}$. Thus, setting one to zero sets the other also to zero. Setting $\partial Q/\partial \mathbf{x}' = \mathbf{0}$ gives the normal equations:

$$\mathbf{0} \quad = \quad \frac{\partial Q}{\partial \mathbf{x}'} \quad = \quad \mathbf{F}'(\mathbf{F}\mathbf{x} - \mathbf{d}) \tag{2.35}$$

The result is merely the complex form of our earlier result (1.32). 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.

2.2.4 From the frequency domain to the time domain

Equation (1.4) is a frequency-domain quadratic form that we minimized by varying a single parameter, a Fourier coefficient. Now we will look at the same problem in the time domain. We will see that the time domain offers flexibility with boundary conditions, constraints, and weighting functions. The notation will be that a filter f_t has input x_t and output y_t. In Fourier space this is $Y = XF$. There are two problems 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 (1.5). Thus the quadratic form to be minimized is a restatement of equation (1.34) with filter definitions:

$$Q(\mathbf{f}', \mathbf{f}) \quad = \quad (\mathbf{X}\mathbf{f} - \mathbf{y})'(\mathbf{X}\mathbf{f} - \mathbf{y}) \tag{2.36}$$

The solution $\mathbf{f}$ is found just as we found (1.35), and it is the set of simultaneous equations $\mathbf{0} = \mathbf{X}'(\mathbf{X}\mathbf{f} - \mathbf{y})$.

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 will never be able to find the input in that band and will need to prevent x_t from diverging there. We do this by the statement that we wish $\mathbf{0} \approx \epsilon\, \mathbf{x}$, where ϵ is a parameter that is small and whose exact size will be chosen by experimentation. Putting both wishes into a single, partitioned matrix equation gives

$$\begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix} \quad \approx \quad \begin{bmatrix} \mathbf{r}_1 \\ \mathbf{r}_2 \end{bmatrix} \quad = \quad \begin{bmatrix} \mathbf{F} \\ \epsilon\, \mathbf{I} \end{bmatrix} \mathbf{x} \quad - \quad \begin{bmatrix} \mathbf{y} \\ \mathbf{0} \end{bmatrix} \tag{2.37}$$

To minimize the residuals $\mathbf{r}_1$ and $\mathbf{r}_2$, we can minimize the scalar $\mathbf{r}'\mathbf{r} = \mathbf{r}'_1\mathbf{r}_1 + \mathbf{r}'_2\mathbf{r}_2$. This is

$$\begin{aligned} Q(\mathbf{x}',\mathbf{x}) &= (\mathbf{Fx}-\mathbf{y})'(\mathbf{Fx}-\mathbf{y}) + \epsilon^2\mathbf{x}'\mathbf{x} \\ &= (\mathbf{x}'\mathbf{F}'-\mathbf{y}')(\mathbf{Fx}-\mathbf{y}) + \epsilon^2\mathbf{x}'\mathbf{x} \end{aligned} \tag{2.38}$$

We solved this minimization in the frequency domain (beginning from equation (1.4)).

Formally the solution is found just as with equation (1.35), but this solution looks unappealing in practice because there are so many unknowns and because 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 to discover that constraints or time-variable weighting functions are required in some applications. This is an issue we must be continuously alert to, whether the cost of a method is justified by its need.

2.3 ITERATIVE METHODS

The **solution time** for simultaneous **linear equations** grows cubically with the number of unknowns. There are three regimes for solution; which one is applicable depends on the number of unknowns n. For n 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. For $n < 500$ we use exact numerical methods such as **Gauss reduction**. A 1988 vintage workstation solves a 100×100 system in a minute, but a 1000×1000 system requires a week. At around $n = 500$, exact numerical methods must be abandoned and **iterative method**s must be used.

An example of a geophysical problem with $n > 1000$ is a missing seismogram. Deciding how to handle a missing seismogram may at first seem like a question of missing *data*, not excess numbers of *model* points. In fitting wave-field data to a consistent model, however, the missing data is seen to be just more unknowns. In real life we generally have not one missing seismogram, but many. Theory in 2-D requires that seismograms be collected along an infinite line. Because any data-collection activity has a start and an end, however, practical analysis must choose between falsely asserting zero data-values at locations where data was not collected, or implicitly determining values for unrecorded data at the ends of a survey.

A numerical technique known as the "**conjugate-direction method**" works well for all values of n 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 is too large to allow n^3 computations, 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.

2.3.1 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 $(\mathbf{Fm} - \mathbf{d})$.

A strong experimentalist prefers to think of the error as experimental error, something for him or her 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.)

Regardless of the above, and opposite to common practice, I define the **sign convention** for the error (or residual) as $(\mathbf{Fm} - \mathbf{d})$. When we choose this sign convention, our hazard for analysis errors will be reduced because $\mathbf{F}$ is often complicated and formed by combining many 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 design a stronger theory.

2.3.2 Method of random directions and steepest descent

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

$$\text{residual} \quad = \quad \text{transform} \quad \text{model space} \quad - \quad \text{data space} \tag{2.39}$$

$$\begin{bmatrix} \\ \mathbf{r} \\ \\ \end{bmatrix} = \begin{bmatrix} & & \\ & \mathbf{F} & \\ & & \end{bmatrix} \begin{bmatrix} \mathbf{x} \end{bmatrix} - \begin{bmatrix} \\ \mathbf{d} \\ \\ \end{bmatrix} \tag{2.40}$$

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 $0 \approx \mathbf{r} = \mathbf{F}\mathbf{x} - \mathbf{d}$. 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 $\mathbf{x}_{\rm new}$:

$$\mathbf{x}_{\rm new} \quad = \quad \mathbf{x} + \alpha\Delta\mathbf{x} \tag{2.41}$$

This gives a new residual:

$$\begin{aligned} \mathbf{r}_{\rm new} &= \mathbf{F}\ \mathbf{x}_{\rm new} - \mathbf{d} && (2.42) \\ \mathbf{r} + \alpha\mathbf{F}\Delta\mathbf{x} &= \mathbf{F}(\mathbf{x} + \alpha\Delta\mathbf{x}) - \mathbf{d} && (2.43) \\ \mathbf{r}_{\rm new} \quad = \quad \mathbf{r} + \alpha\Delta\mathbf{r} &= \mathbf{F}\mathbf{x} - \mathbf{d} + \alpha\mathbf{F}\Delta\mathbf{x} && (2.44) \end{aligned}$$

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

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

$$(\mathbf{r} + \alpha\Delta\mathbf{r}) \cdot (\mathbf{r} + \alpha\Delta\mathbf{r}) \quad = \quad \mathbf{r} \cdot \mathbf{r} + 2\alpha(\mathbf{r} \cdot \Delta\mathbf{r}) \ + \ \alpha^2 \Delta\mathbf{r} \cdot \Delta\mathbf{r} \tag{2.45}$$

Set to zero its derivative with respect to α using the chain rule

$$0 \quad = \quad (\mathbf{r} + \alpha\Delta\mathbf{r}) \cdot \Delta\mathbf{r} \ + \ \Delta\mathbf{r} \cdot (\mathbf{r} + \alpha\Delta\mathbf{r}) \quad = \quad 2(\mathbf{r} + \alpha\Delta\mathbf{r}) \cdot \Delta\mathbf{r} \tag{2.46}$$

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

$$\alpha \quad = \quad - \frac{(\mathbf{r} \cdot \Delta\mathbf{r})}{(\Delta\mathbf{r} \cdot \Delta\mathbf{r})} \tag{2.47}$$

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

$$\begin{array}{l} \mathbf{r} \quad \longleftarrow \quad \mathbf{F}\mathbf{x} - \mathbf{d} \\ \text{iterate } \{ \\ \qquad \begin{array}{lcl} \Delta\mathbf{x} & \longleftarrow & \text{random numbers} \\ \Delta\mathbf{r} & \longleftarrow & \mathbf{F}\ \Delta\mathbf{x} \\ \alpha & \longleftarrow & -(\mathbf{r} \cdot \Delta\mathbf{r})/(\Delta\mathbf{r} \cdot \Delta\mathbf{r}) \\ \mathbf{x} & \longleftarrow & \mathbf{x} + \alpha\Delta\mathbf{x} \\ \mathbf{r} & \longleftarrow & \mathbf{r}\ + \alpha\Delta\mathbf{r} \\ \} & & \end{array} \end{array}$$

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

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

$$\mathbf{g} \quad = \quad \mathbf{F}'\mathbf{r} \tag{2.48}$$

Notice also that this vector can be found by taking the gradient of the size of the residuals:

$$\frac{\partial}{\partial \mathbf{x}'} \mathbf{r}\cdot\mathbf{r} \quad = \quad \frac{\partial}{\partial \mathbf{x}'} (\mathbf{x}'\,\mathbf{F}' \;-\; \mathbf{d}')\,(\mathbf{F}\,\mathbf{x} \;-\; \mathbf{d}) \quad = \quad \mathbf{F}'\,\mathbf{r} \tag{2.49}$$

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

Starting from a model $\mathbf{x} = \mathbf{m}$ (which may be zero), below 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:

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

2.3.3 Null space and gradient 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 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 them. 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:

$$\mathbf{x}_{i+1} \;=\; \mathbf{x}_i + \alpha\Delta\mathbf{x} \tag{2.50}$$

$$\mathbf{x}_{i+1} \;=\; \mathbf{x}_i + \alpha\mathbf{F}'\mathbf{r} \tag{2.51}$$

$$\mathbf{x}_{i+1} \;=\; \mathbf{x}_i + \alpha\mathbf{F}'(\mathbf{F}\mathbf{x} - \mathbf{d}) \tag{2.52}$$

After we have iterated to convergence, the gradient $\Delta\mathbf{x}$ vanishes as does $\mathbf{F}'(\mathbf{F}\mathbf{x} - \mathbf{d})$. Thus, an iterative solver gets the same solution as the theory leading to equation (1.27).

Suppose that by adding a huge amount of $\mathbf{x}_{\text{null}}$, we now change $\mathbf{x}$ and continue iterating. Notice that $\Delta\mathbf{x}$ remains zero because $\mathbf{F}\mathbf{x}_{\text{null}}$ vanishes. Thus we conclude that any null space in the initial guess $\mathbf{x}_0$ will remain there unaffected by the gradient-descent process.

Although finding a basis for the null space would take us too far afield into linear algebra, a more practical goal is to see some members of the null space, if any. To try to see a member of the null space, we take two starting guesses and run our iterative solver for each of them. If the solutions are the same, there is no null space. If the two different solutions differ, say $\mathbf{x}_1$ and $\mathbf{x}_2$, there will be a null space. Since the residual squared is a convex quadratic function, the two solutions must have the same residual $\mathbf{r}$. Subtracting $\mathbf{r} = \mathbf{F}\mathbf{x}_1 - \mathbf{d}$ from $\mathbf{r} = \mathbf{F}\mathbf{x}_2 - \mathbf{d}$ we find $\mathbf{F}(\mathbf{x}_1 - \mathbf{x}_2) = 0$, so $\mathbf{x}_1 - \mathbf{x}_2$ is a model in the null space. Adding $\mathbf{x}_1 - \mathbf{x}_2$ to any to any model $\mathbf{x}$ will not change the theoretical data.

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

2.3.4 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). Visualize a contour map of a mountainous terrain. 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 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.

2.3.5 Conjugate direction

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 will be chosen to be the gradient vector, say $\mathbf{g}$. The other vector will be 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 equations will be specified here along with the program. (For *nonquadratic* functions a plane search is considered intractable, whereas a line search proceeds by bisection.)

For use in linear problems, the conjugate-direction method described in this book follows an identical path with the more well-known conjugate-gradient method. We use the conjugate-direction method for convenience in exposition and programming.

The simple form of the conjugate-direction algorithm covered here is a sequence of steps. In each step the minimum is found in the plane given by two vectors: the gradient vector and the vector of the previous step. Given the linear operator $\mathbf{F}$ and a generator of solution steps (random in the case of random directions or gradient in the case of steepest descent), we can construct an optimally convergent iteration process, which finds the solution in no more than n steps, where n is the size of the problem. This result should not be surprising. If $\mathbf{F}$ is represented by a full matrix, then the cost of direct inversion is proportional to n^3, and the cost of matrix multiplication is n^2. Each step of an iterative method boils down to a matrix multiplication. Therefore, we need at least n steps to arrive at the exact solution. Two circumstances make large-scale optimization practical. First, for sparse convolution matrices the cost of matrix multiplication is n instead of n^2. Second, we can often find a reasonably good solution after a limited number of iterations. If both these conditions are met, the cost of optimization grows linearly with n, which is a practical rate even for very large problems. Fourier-transformed variables are often capitalized. This convention will be 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 print. Thus we define the solution, the solution step (from one iteration to the next), and the gradient by

$$X \quad = \quad \mathbf{F}\, x \qquad \text{solution} \tag{2.53}$$
$$S_j \quad = \quad \mathbf{F}\, s_j \qquad \text{solution step} \tag{2.54}$$
$$G_j \quad = \quad \mathbf{F}\, g_j \qquad \text{solution gradient} \tag{2.55}$$

A linear combination in solution space, say $s+g$, corresponds to $S+G$ in the conjugate space, because $S+G = \mathbf{F}s+\mathbf{F}g = \mathbf{F}(s+g)$. According to equation (1.40), the residual is the theoretical data minus the observed data.

$$R \quad = \quad \mathbf{F}x \ - \ D \quad = \quad X \ - \ D \tag{2.56}$$

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

$$x \quad = \quad s_1 \ + \ s_2 \ + \ s_3 \ + \ \cdots \tag{2.57}$$

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

$$\text{solution update :} \qquad x \leftarrow x \ + s \tag{2.58}$$
$$\text{residual update :} \qquad R \leftarrow R \ + S \tag{2.59}$$

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

$$g = \mathbf{F}' R = \text{gradient} \tag{2.60}$$
$$G = \mathbf{F}\, g = \text{conjugate gradient} \tag{2.61}$$

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

The minimization (1.45) is now generalized to scan not only the line with α, but simultaneously another line with β. The combination of the two lines is a plane:

$$Q(\alpha, \beta) = (R + \alpha G + \beta S) \cdot (R + \alpha G + \beta S) \tag{2.62}$$

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

$$0 = G \cdot (R + \alpha G + \beta S) \tag{2.63}$$

$$0 = S \cdot (R + \alpha G + \beta S) \tag{2.64}$$

The solution is

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \frac{-1}{(G\cdot G)(S\cdot S) - (G\cdot S)^2} \begin{bmatrix} (S\cdot S) & -(S\cdot G) \\ -(G\cdot S) & (G\cdot G) \end{bmatrix} \begin{bmatrix} (G\cdot R) \\ (S\cdot R) \end{bmatrix} \tag{2.65}$$

The many applications in this book all need to find α and β with (1.65) and then update the solution with (1.58) and update the residual with (1.59). Thus we package these activities in a subroutine named `cgstep()`. To use that subroutine we will have a computation **template** like we had for steepest descents, except that we will have the 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

$$
\begin{array}{l}
\mathbf{r} \quad \longleftarrow \quad \mathbf{F}\mathbf{x} - \mathbf{d} \\
\text{iterate } \{ \\
\qquad\qquad \Delta\mathbf{x} \quad \longleftarrow \quad \mathbf{F}'\,\mathbf{r} \\
\qquad\qquad \Delta\mathbf{r} \quad \longleftarrow \quad \mathbf{F}\,\Delta\mathbf{x} \\
\qquad\qquad (\mathbf{x}, \mathbf{r}) \quad \longleftarrow \quad \text{cgstep}(\mathbf{x}, \mathbf{r}, \Delta\mathbf{x}, \Delta\mathbf{r}) \\
\qquad\qquad \}
\end{array}
$$

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.

2.3.6 Routine for one step of conjugate-direction descent

Because **Fortran** does not recognize the difference between upper- and lower-case letters, 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()`.

```
module cgstep_mod  {
    real, dimension (:), allocatable, private   :: s, ss
contains
    function cgstep( forget, x, g, rr, gg) result (stat) {
        integer              :: stat
        real, dimension (:)  :: x, g, rr, gg
        logical              :: forget
        double precision     :: sds, gdg, gds, determ, gdr, sdr, alfa, beta
        if( .not. allocated (s)) {   forget = .true.
                     allocate ( s (size ( x)))
                     allocate (ss (size (rr)))
                     }
        if( forget){ s = 0.;  ss = 0.;  beta = 0.d0    # steepest descent
                     if( dot_product(gg, gg) == 0 )
                              call erexit('cgstep: grad vanishes identically')
                     alfa = - sum( dprod(gg,rr)) / sum( dprod(gg, gg))
                     }
        else{ gdg = sum( dprod(gg, gg))       # search plane by solving 2-by-2
              sds = sum( dprod(ss, ss))       #  G . (R - G*alfa - S*beta) = 0
              gds = sum( dprod(gg, ss))       #  S . (R - G*alfa - S*beta) = 0
              if( gdg==0. .or. sds==0.)  { stat = 1;  return }
              determ = gdg * sds * max (1.d0 - (gds/gdg)*(gds/sds), 1.d-12)
              gdr = - sum( dprod(gg,rr))
              sdr = - sum( dprod(ss,rr))
              alfa = ( sds * gdr - gds * sdr ) / determ
              beta = (-gds * gdr + gdg * sdr ) / determ
              }
        s  = alfa *  g + beta *  s            # update solution step
        ss = alfa * gg + beta * ss            # update residual step
        x  =  x +  s                          # update solution
        rr = rr + ss                          # update residual
        forget = .false.;   stat = 0
    }
    subroutine cgstep_close ( ) {
         if (allocated (s)) deallocate (s, ss)
    }
}
```

2.3.7 Setting up a generic solver program

There are many different methods for iterative least-square estimation some of which will be discussed later in this book. The conjugate-gradient (CG) family (including the first order conjugate-direction method described above) 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 non-linear 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 problem. The solver entrance is for the specialist in numerical algebra, who designs a new optimization method.

The Fortran-90 programming language allows us to achieve this design goal by means of generic function interfaces.

A generic solver subroutine `solver()` is shown in module `simple_solver`. It is simplified substantially from the library version, which has a much longer list of optional arguments

```
module simple_solver  {
  logical, parameter, private  :: T = .true., F = .false.
contains
  subroutine solver( oper, solv, x, dat, niter, x0, res)  {
    optional                                  :: x0, res
    interface {
       function oper( adj, add, x, dat)  result (stat) {
            integer               :: stat
            logical, intent (in) :: adj, add
            real, dimension (:)  :: x, dat
            }
       function solv( forget, x, g, rr, gg)  result (stat) {
            integer              :: stat
            logical              :: forget
            real, dimension (:) :: x, g, rr, gg
            }
       }
    real,    dimension (:),   intent (in)  :: dat, x0     # data, initial
    real,    dimension (:),   intent (out) :: x, res      # solution, residual
    integer,                  intent (in)  :: niter       # iterations
    real, dimension (size (x))             :: g           # gradient
    real, dimension (size (dat))           :: rr, gg      # residual, conj grad
    integer                                :: i, stat

                                        rr = - dat
    if( present( x0)) {  stat = oper( F, T, x0, rr)       # rr <- F x0 - dat
                                        x = x0            # start with x0
                           }
    else {                              x = 0.            # start with zero
         }
    do i = 1, niter  {
       stat = oper( T, F, g, rr)                          # g  <- F' rr
       stat = oper( F, F, g, gg)                          # gg <- F  g
       stat = solv( F, x, g, rr, gg)                      # step in x and rr
```

```
      }
    if( present( res)) res = rr
  }
}
```

(The `forget` parameter is not needed by the solvers we discuss first.)

The two most important arguments in `solver()` are the operator function `oper`, which is defined by the interface from Chapter 1, and the solver function `solv`, which implements one step of an iterative estimation. For example, a practitioner who choses to use our new `cgstep()` on page 15 for iterative solving the operator `matmult` on page 3 would write the call

```
call solver ( matmult_lop, cgstep, ...
```

The other required parameters to `solver()` are `dat` (the data we want to fit), `x` (the model we want to estimate), and `niter` (the maximum number of iterations). There is 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 problems in later chapters.

2.3.8 Test case: solving some simultaneous equations

Now we assemble a module `cgmeth` for solving simultaneous equations. Starting with the conjugate-direction module `cgstep_mod` on page 15 we insert the module `matmult` on page 3 as the linear operator.

```
module cgmeth {
  use matmult
  use cgstep_mod
  use simple_solver
contains
  # setup of conjugate gradient descent, minimize  SUM rr(i)**2
  #             nx
  # rr(i)  =   sum fff(i,j) * x(j)  -  yy(i)
  #            j=1
  subroutine cgtest( x, yy, rr, fff, niter) {
    real, dimension (:), intent (out) :: x, rr
    real, dimension (:), intent (in)  :: yy
    real, dimension (:,:), pointer    :: fff
    integer,             intent (in)  :: niter

    call matmult_init( fff)

    call solver( matmult_lop, cgstep, x, yy, niter, res = rr)

    call cgstep_close ()
```

```
  }
}
```

Below shows the solution to 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
```

EXERCISES:

1 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?

2 What linear operator subroutine from Chapter 1 can be used for finding the mean?

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

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

2.4 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 towards ***inversion*** in the application called **moveout and stack**.

The seismograms at the bottom of Figure 1.1 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

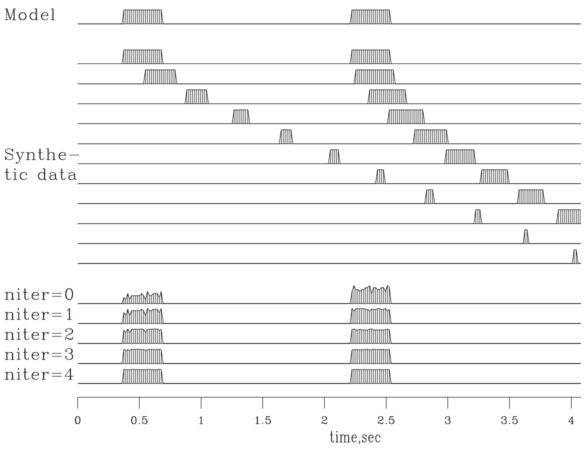

Figure 2.1: Top is a model trace **m**. Next are the synthetic data traces, $\mathbf{d} = \mathbf{M}\mathbf{m}$. Then, labeled `niter=0` is the **stack**, a result of processing by adjoint modeling. Increasing values of `niter` show **x** as a function of iteration count in the fitting goal $\mathbf{d} \approx \mathbf{M}\mathbf{m}$. (Carlos Cunha-Filho) $\boxed{\text{lsq-invstack90}}$ [ER]

events have unequal amplitudes, but after enough iterations they are equal, as they began. Mathematically, we can denote the top trace as the model **m**, the synthetic data signals as $\mathbf{d} = \mathbf{M}\mathbf{m}$, and the stack as $\mathbf{M}'\mathbf{d}$. The conjugate-gradient algorithm optimizes the fitting goal $\mathbf{d} \approx \mathbf{M}\mathbf{x}$ by variation of **x**, and the figure shows **x** converging to **m**. Because there are 256 unknowns in **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 `cgmeth` on page 18 except that the matrix-multiplication operator `matmult` on page 3 has been replaced by `imospray` on the facing page. Studying the program, you can deduce that, except for a scale factor, the output at `niter=0` is identical to the stack $\mathbf{M}'\mathbf{d}$. All the signals in Figure 1.1 are intrinsically the same scale.

```
module invstack {
        use imospray
        use cgstep_mod
        use simple_solver
contains
        # NMO stack by inverse of forward modeling
        subroutine stack( nt, model, nx, gather,     t0,x0,dt,dx,slow, niter) {
        integer          nt,        nx,                               niter
        real                 model (:), gather (:), t0,x0,dt,dx,slow

        call imospray_init( slow, x0,dx, t0,dt, nt, nx)

        call solver( imospray_lop, cgstep, model, gather, niter)

        call cgstep_close ();  call imospray_close ()   # garbage collection
        }
}
```

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

2.5 VESUVIUS PHASE UNWRAPPING

Figure 1.2 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). They 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 1.2. 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 1.2) but because these generally have spatially consistent phase I would not describe the data as containing noise bursts.

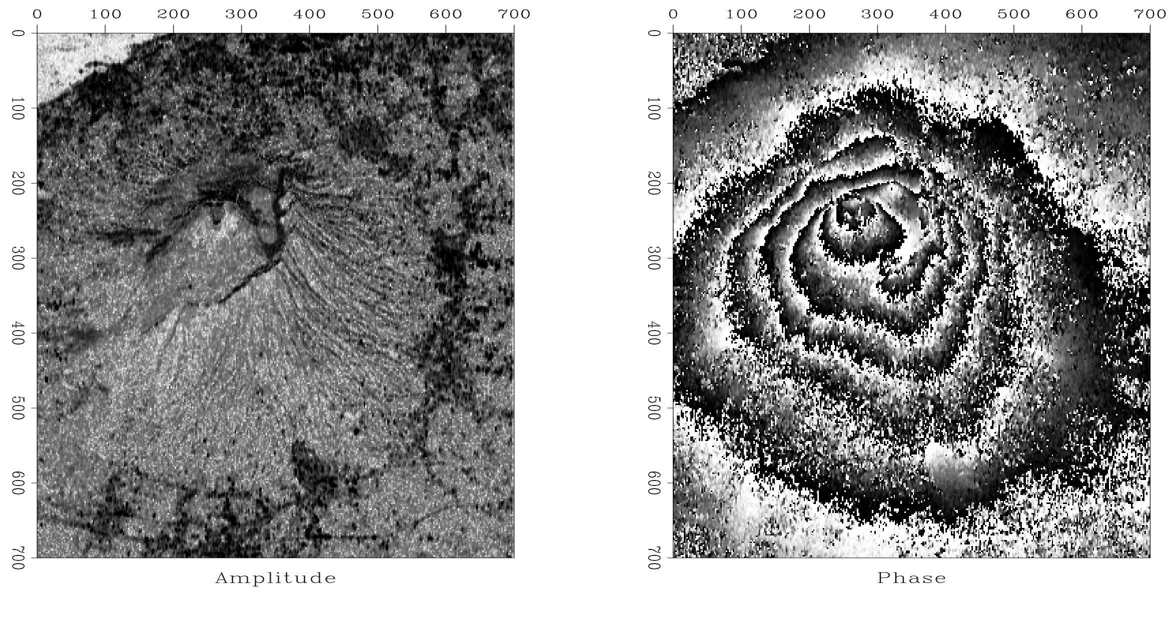

Figure 2.2: Radar image of Mt. Vesuvius. Left is the amplitude. Non-reflecting ocean in upper left corner. Right is the phase. (Umberto Spagnolini) lsq-vesuvio90 [ER]

To reduce the time needed for analysis and printing, I reduced the data size two different ways, by decimation and by local averaging, as shown in Figure 1.3. The

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

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 the plane of the real part and the plane of the imaginary part. Putting them back together again showed that the phase angle of the averaged data behaves much more consistently. This adds evidence that the data is not troubled by noise bursts.

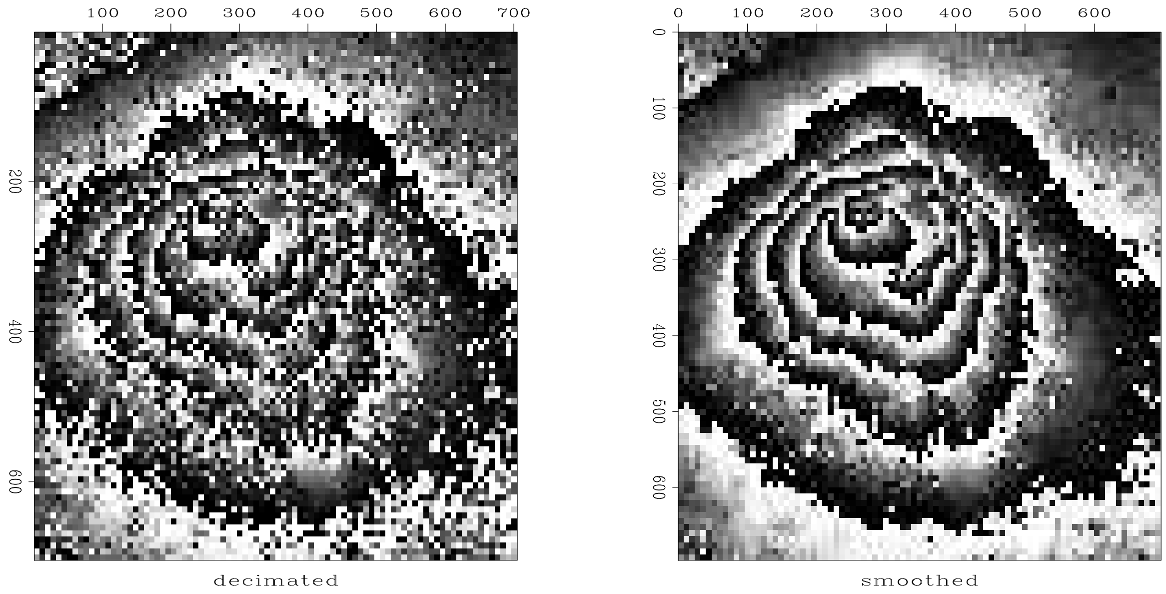

Figure 2.3: Phase based on decimated data (left) and smoothed data (right). $\boxed{\text{lsq-squeeze90}}$ [ER]

From Figures 1.2 and 1.3 we see that **contour**s 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 $Z = e^{ih}$ where h is altitude (in units unknown to us nonspecialists). 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 of its jumps of 2π. Begin with a complex number Z representing the complex-valued image at any location in the (x, y)-plane.

$$
\begin{aligned}
re^{i\phi} &= Z && (2.66)\\
\ln|r| + i\phi &= \ln Z && (2.67)
\end{aligned}
$$

$$\phi = \Im \ln Z \tag{2.68}$$

A computer will find the imaginary part of the logarithm with the arctan function of two arguments, `atan2(y,x)`, which will put the phase in the range $-\pi < \phi \leq \pi$ although any multiple of 2π could be added. We seem to escape the $2\pi N$ phase ambiguity by differentiating:

$$\frac{\partial \phi}{\partial x} = \Im \frac{1}{Z} \frac{\partial Z}{\partial x} \tag{2.69}$$

$$\frac{\partial \phi}{\partial x} = \frac{\Im \bar{Z} \frac{\partial Z}{\partial x}}{\bar{Z} Z} \tag{2.70}$$

For every point on the y-axis, equation (1.70) is a differential equation on the x-axis, and 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, so we get twice as many equations as unknowns. For ideal data, either of these sets of equations should be equivalent to the other, but for real data we expect to be fitting the fitting goal

$$\nabla \phi \approx \frac{\Im \bar{Z} \nabla Z}{\bar{Z} Z} \tag{2.71}$$

where $\nabla = (\frac{\partial}{\partial x}, \frac{\partial}{\partial y})$.

We will be handling the differential equation as a difference equation using an exact representation on the data mesh. By working with the phase difference of neighboring data values, we do not have to worry about phases greater than 2π (except where phase jumps that much between mesh points). Thus we solve (1.71) with finite differences instead of differentials. Module `igrad2` is a linear operator for the difference representation of the gradient operator.

```
module igrad2 {                              # 2-D gradient with adjoint,  r= grad( p)
integer :: n1, n2
#%_init   (n1, n2)
#%_lop ( p(n1, n2),  r(n1,n2,2))
integer i,j
do i= 1, n1-1 {
do j= 1, n2-1 {
        if( adj) {
                p(i+1,j  )  +=  r(i,j,1)
                p(i  ,j  )  -=  r(i,j,1)
                p(i  ,j+1)  +=  r(i,j,2)
                p(i  ,j  )  -=  r(i,j,2)
                }
        else {  r(i,j,1)  +=  ( p(i+1,j) - p(i,j))
                r(i,j,2)  +=  ( p(i,j+1) - p(i,j))
                }
        }}
}
```

The difference equation representation of the fitting goal (1.71) is:

$$\begin{aligned} \phi_{i+1,j} - \phi_{i,j} &\approx \Delta\phi_{ac} \\ \phi_{i,j+1} - \phi_{i,j} &\approx \Delta\phi_{ab} \end{aligned} \tag{2.72}$$

where we still need to define the right-hand side. Define the parameters a, b, c, and d as follows:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad = \quad \begin{bmatrix} Z_{i,j} & Z_{i,j+1} \\ Z_{i+1,j} & Z_{i+1,j+1} \end{bmatrix} \tag{2.73}$$

Arbitrary 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 $\ln|r_a r_b| + i\Delta\phi_{ab}$.

$$\begin{array}{rcccl} \phi_b - \phi_a &=& \Delta\phi_{ab} &=& \Im \ln \bar{a}b \\ \phi_d - \phi_c &=& \Delta\phi_{cd} &=& \Im \ln \bar{c}d \\ \phi_c - \phi_a &=& \Delta\phi_{ac} &=& \Im \ln \bar{a}c \\ \phi_d - \phi_b &=& \Delta\phi_{bd} &=& \Im \ln \bar{b}d \end{array} \tag{2.74}$$

which gives the information needed to fill in the right-hand side of (1.72), as done by subroutine `igrad2init()` from module `unwrap` on the current page.

2.5.1 Digression: curl grad as a measure of bad data

The relation (1.74) between the phases and the phase differences is

$$\begin{bmatrix} -1 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} \phi_a \\ \phi_b \\ \phi_c \\ \phi_d \end{bmatrix} \quad = \quad \begin{bmatrix} \Delta\phi_{ab} \\ \Delta\phi_{cd} \\ \Delta\phi_{ac} \\ \Delta\phi_{bd} \end{bmatrix} \tag{2.75}$$

Starting from the phase differences, equation (1.75) hope to find all the phases themselves because an additive constant cannot be found. In other words, the column vector $[1, 1, 1, 1]'$ is in the null space. Likewise, if we add phase increments while we move around a loop, the sum should be zero. Let the loop be $a \to c \to d \to b \to a$. The phase increments that sum to zero are:

$$\Delta\phi_{ac} + \Delta\phi_{cd} - \Delta\phi_{bd} - \Delta\phi_{ab} \quad = \quad 0 \tag{2.76}$$

Rearranging to agree with the order in equation (1.75) yields

$$-\Delta\phi_{ab} + \Delta\phi_{cd} + \Delta\phi_{ac} - \Delta\phi_{bd} \quad = \quad 0 \tag{2.77}$$

which says that the row vector $[-1, +1, +1, -1]$ premultiplies (1.75), yielding zero. Rearrange again

$$-\Delta\phi_{bd} + \Delta\phi_{ac} \quad = \quad \Delta\phi_{ab} - \Delta\phi_{cd} \tag{2.78}$$

and finally interchange signs and directions (i.e., $\Delta\phi_{db} = -\Delta\phi_{bd}$)

$$(\Delta\phi_{db} - \Delta\phi_{ca}) \ - \ (\Delta\phi_{dc} - \Delta\phi_{ba}) \quad = \quad 0 \tag{2.79}$$

This is the finite-difference equivalent of

$$\frac{\partial^2 \phi}{\partial x \partial y} \ - \ \frac{\partial^2 \phi}{\partial y \partial x} \quad = \quad 0 \tag{2.80}$$

and is also the z-component of the theorem that the curl of a gradient $\nabla \times \nabla\phi$ vanishes for any ϕ.

The four $\Delta\phi$ summed around the 2×2 mesh should add to zero. I wondered what would happen if random complex numbers were used for a, b, c, and d, so I computed the four $\Delta\phi$s with equation (1.74), and then computed the sum with (1.76). They did sum to zero for 2/3 of my random numbers. Otherwise, with probability 1/6 each, they summed to $\pm 2\pi$. The nonvanishing curl represents a phase that is changing too rapidly between the mesh points. Figure 1.4 shows the locations at Vesuvius where bad data occurs. This is shown at two different resolutions. The figure shows a tendency for bad points with curl 2π to have a neighbor with -2π. If Vesuvius were random noise instead of good data, the planes in Figure 1.4 would be one-third covered with dots but as expected, we see considerably fewer.

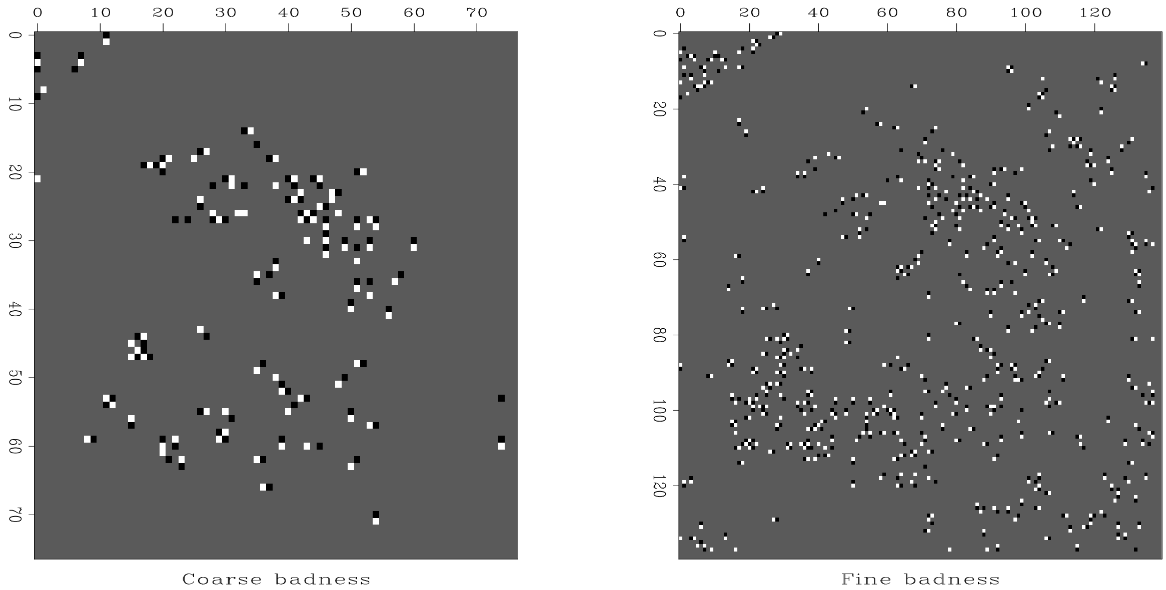

Figure 2.4: Values of curl at Vesuvius. The bad data locations at both coarse and fine resolution tend to occur in pairs of opposite polarity. $\boxed{\text{lsq-screw90}}$ [ER]

2.5.2 Estimating the inverse gradient

To optimize the fitting goal (1.72), module `unwrap()` uses the conjugate-direction method like the modules `cgmeth()` on page 18 and `invstack()` on page 20.

```
module unwrap {
        use cgstep_mod
        use igrad2
        use simple_solver
contains
        subroutine grad2init( z,n1,n2, rt ) {
        integer  i, j,          n1,n2
```

```
        real                          rt( n1,n2,2)
        complex              z(n1,n2),             a,b,c
        rt = 0.
        do i= 1, n1-1 {
        do j= 1, n2-1 {
                a =  z(i  ,j  )
                c =  z(i+1,j  );     rt(i,j,1) =  aimag( clog( c * conjg( a) ) )
                b =  z(i,  j+1);     rt(i,j,2) =  aimag( clog( b * conjg( a) ) )
                }}
        }
        # Phase unwraper.   Starting from phase h, improve it.
        subroutine unwraper( zz, hh, niter) {
        integer  n1,n2,             niter
        complex           zz(:,:)
        real              hh(:)
        real, allocatable ::  rt(:)
        n1 = size (zz, 1)
        n2 = size (zz, 2)
        allocate( rt (n1*n2*2))
        call grad2init( zz,n1,n2,rt)
        call igrad2_init(  n1,n2)
        call solver( igrad2_lop, cgstep, hh, rt, niter, x0 = hh)
        call cgstep_close ()
        deallocate( rt)
        }
}
```

An open question is whether the required number of iterations is reasonable or whether we would need to uncover a preconditioner or more rapid solution method. I adjusted the frame size (by the amount of smoothing in Figure 1.3) so that I would get the solution in about ten seconds with 400 iterations. Results are shown in Figure 1.5.

2.5.3 Discontinuity in the solution

The viewing angle (23 degrees off vertical) in Figure 1.2 might be such that the mountain blocks some of the landscape behind it. This leads to the interesting possibility that the phase function must have a discontinuity where our viewing angle jumps over the hidden terrain. It will be interesting to discover whether we can estimate functions with such discontinuities. I am not certain that the Vesuvius data really has such a shadow zone, so I prepared the synthetic data in Figure 1.6, which is noise free and definitely has one.

We notice the polarity of the synthetic data in 1.6 is opposite that of the Vesuvius data. This means that the radar altitude of Vesuvius is not measured from sea level but from the satellite level.

A reason I particularly like this Vesuvius exercise is that slight variations on the theme occur in various other fields. For example, in 3-D seismology we can take

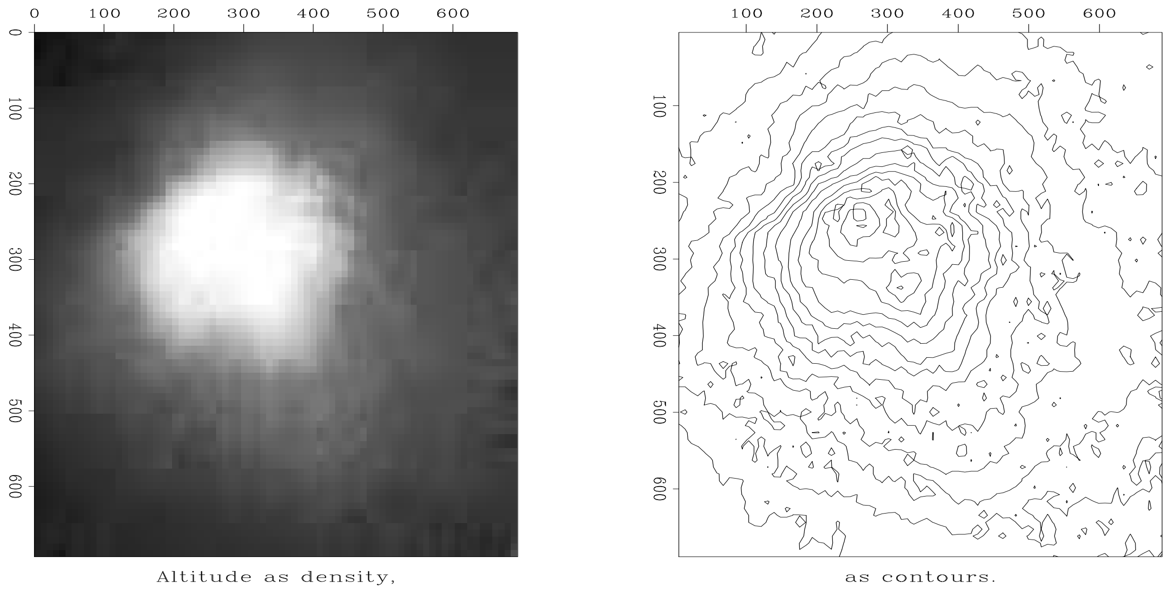

Figure 2.5: Estimated altitude. lsq-veshigh90 [ER]

the cross-correlation of each seismogram with its neighbor and pick the time lag of the maximum correlation. Such time shifts from trace to trace can be organized as we have organized the $\Delta\phi$ values of Vesuvius. The discontinuity in phase along the skyline of our Vesuvius view is like the faults we find in the earth.

EXERCISES:

1 In differential equations, boundary conditions are often (1) a specified function value or (2) a specified derivative. These are associated with (1) transient convolution or (2) internal convolution. Gradient operator `igrad2` on page 23 is based on internal convolution with the filter $(1, -1)$. Revise `igrad2` to make a module called `tgrad2` which has transient boundaries. Hint: Compare with `icai2` on page 183 and `tcai2` on page ??.

2.6 THE WORLD OF CONJUGATE GRADIENTS

Nonlinearity arises in two ways: First, theoretical 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.

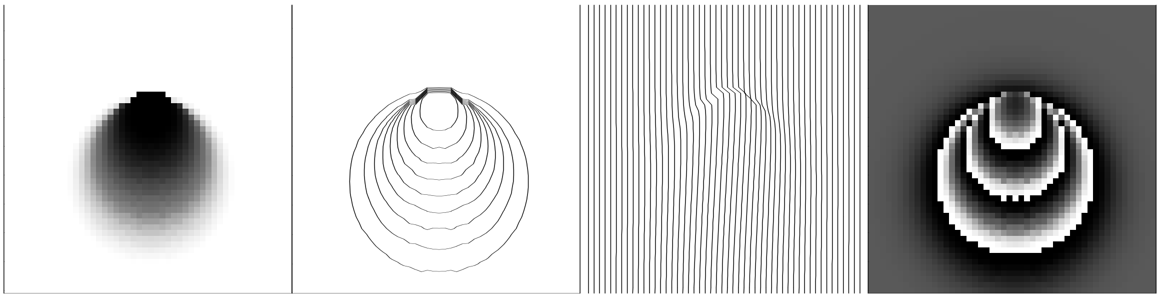

Figure 2.6: Synthetic mountain with hidden backside. For your estimation enjoyment. lsq-synmod90 [ER]

2.6.1 Physical nonlinearity

When standard methods of physics relate theoretical data $\mathbf{d}_{\text{theor}}$ to model parameters $\mathbf{m}$, they often use a nonlinear relation, say $\mathbf{d}_{\text{theor}} = \mathbf{f}(\mathbf{m})$. The power-series approach then leads to representing theoretical data as

$$\mathbf{d}_{\text{theor}} \quad = \quad \mathbf{f}(\mathbf{m}_0 + \Delta\mathbf{m}) \quad \approx \quad \mathbf{f}(\mathbf{m}_0) + \mathbf{F}\Delta\mathbf{m} \tag{2.81}$$

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 theoretical data $\mathbf{d}_{\text{theor}}$ minus the observed data $\mathbf{d}_{\text{obs}}$ is the residual we minimize.

$$\mathbf{0} \quad \approx \quad \mathbf{d}_{\text{theor}} - \mathbf{d}_{\text{obs}} \quad = \quad \mathbf{F}\boldsymbol{\Delta}\mathbf{m} + [\mathbf{f}(\mathbf{m}_0) - \mathbf{d}_{\text{obs}}] \tag{2.82}$$

$$\mathbf{r}_{\text{new}} \quad = \quad \mathbf{F}\boldsymbol{\Delta}\mathbf{m} + \mathbf{r}_{\text{old}} \tag{2.83}$$

It is worth noticing that the residual updating (1.83) in a nonlinear problem is the same as that in a linear problem (1.44). If you make a large step $\Delta\mathbf{m}$, however, the new residual will be different from that expected by (1.83). 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.

2.6.2 Statistical nonlinearity

The data itself often has **noise bursts** or **gaps**, and we will see later in Chapter **??** that this leads us to readjusting the **weighting function**. In principle, we should fix the weighting function and solve the problem. Then we should revise the weighting function and solve the problem again. In practice we find it convenient to change the weighting function during the optimization descent. Failure is possible when the weighting function is changed too rapidly or drastically. (The proper way to solve this problem is with robust estimators. Unfortunately, I do not yet have an all-purpose robust solver. Thus we are (temporarily, I hope) reduced to using crude reweighted least-squares methods. Sometimes they work and sometimes they don't.)

2.6.3 Coding nonlinear fitting problems

We can solve nonlinear least-squares problems in about the same way as we do iteratively reweighted ones. A simple adaptation of a linear method gives us a **nonlinear solver** if the residual is recomputed at each iteration. Omitting the weighting function (for simplicity) the **template** is:

$$
\begin{array}{lll}
\text{iterate } \{ & & \\
& \mathbf{r} \quad \longleftarrow \quad \mathbf{f}(\mathbf{m}) - \mathbf{d} & \\
& \text{Define } \mathbf{F} = \partial \mathbf{d} / \partial \mathbf{m}. & \\
& \Delta\mathbf{m} \quad \longleftarrow \quad \mathbf{F}'\ \mathbf{r} & \\
& \Delta\mathbf{r} \quad \longleftarrow \quad \mathbf{F}\ \Delta\mathbf{m} & \\
& (\mathbf{m}, \mathbf{r}) \quad \longleftarrow \quad \text{step}(\mathbf{m}, \mathbf{r}, \Delta\mathbf{m}, \Delta\mathbf{r}) & \\
& \} &
\end{array}
$$

A formal theory for the optimization exists, but we are not using it here. The assumption we make is that the step size will be small, so that familiar line-search and plane-search approximations will 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 steepest descent with a smaller step size. Perhaps we should test an incremental variation on the status quo: where inside `solver` on page 17, 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). I am planning to work with some mathematicians to gain experience with other solvers.

Experience shows that nonlinear problems have many pitfalls. Start with a linear problem, add a minor physical improvement or unnormal 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, and then move in a stable and controlled manner. A simple solution is to begin with several steps of steepest descent and then switch over to do some more steps of CD. Avoiding CD in earlier iterations can avoid instability. Strong linear "regularization" discussed later can also reduce the effect of nonlinearity.

2.6.4 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 on page 15 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

$$\begin{aligned} 0 &\approx \mathbf{W}(\mathbf{F}\mathbf{m} - \mathbf{d}) && (2.84)\\ 0 &\approx \mathbf{A}\mathbf{m}, && (2.85) \end{aligned}$$

they are equivalent to minimizing the quadratic form:

$$\mathbf{m}: \qquad \min_{\mathbf{m}} Q(\mathbf{m}) \quad = \quad (\mathbf{m}'\mathbf{F}' - \mathbf{d}')\mathbf{W}'\mathbf{W}(\mathbf{F}\mathbf{m} - \mathbf{d}) \; + \; \mathbf{m}'\mathbf{A}'\mathbf{A}\mathbf{m} \qquad (2.86)$$

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

$$\mathbf{x}: \qquad \min_{\mathbf{x}} Q(\mathbf{x}) \quad = \quad \mathbf{x}'\mathbf{H}\mathbf{x} - \mathbf{b}'\mathbf{x} \qquad (2.87)$$

To relate equation (1.86) to (1.87) we expand the parentheses in (1.86) and abandon the constant term $\mathbf{d}'\mathbf{d}$. 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:

$$\begin{aligned} \mathbf{H} &= \mathbf{F}'\mathbf{W}'\mathbf{W}\mathbf{F} + \mathbf{A}'\mathbf{A} && (2.88)\\ \mathbf{b}' &= 2(\mathbf{F}'\mathbf{W}'\mathbf{W}\mathbf{d})' && (2.89) \end{aligned}$$

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

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

When people have trouble with conjugate gradients or conjugate directions, I always refer them to the **Paige and Saunders algorithm** `LSQR`. Methods that form **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. I include module `lsqr` on page 31 without explaining why it works. The interface is similar to `solver` on page 17. Note that the residual vector does not appear explicitly in the program and that we cannot start from a nonzero initial model.

```
module lsqr {
  logical, parameter, private :: T = .true., F = .false.
  private                     :: normalize
contains
    subroutine solver( oper, x, dat, niter)  {
       interface {
           function oper( adj, add, x, dat) result (stat) {
                integer              :: stat
                logical, intent( in) :: adj, add
                real, dimension (:)  :: x, dat
                }
           }
    real, dimension (:), intent (in)    :: dat
    real, dimension (:), intent (out)   :: x
    integer, intent (in)                :: niter
    real, dimension ( size( x  ))       :: w, v
    real, dimension ( size( dat))       :: u
    integer                             :: iter, stat
    double precision                    :: alfa, beta, rhobar, phibar
    double precision                    :: c, s, teta, rho, phi, t1, t2
    u = dat;   x = 0.    ;   call normalize( u, beta)
    stat = oper( T,F,v,u);   call normalize( v, alfa)
    w = v
    rhobar = alfa
    phibar = beta
    do iter = 1, niter {
        u = - alfa * u;  stat = oper( F, T, v, u);  call normalize( u, beta)
        v = - beta * v;  stat = oper( T, T, v, u);  call normalize( v, alfa)
        rho = sqrt( rhobar*rhobar + beta*beta)
        c = rhobar/rho;   s = beta/rho;   teta = s * alfa
        rhobar = - c * alfa;   phi = c * phibar;   phibar = s * phibar
        t1 = phi/rho;  t2 = -teta/rho
        x = x + t1 * w
        w = v + t2 * w
        }
    }
  subroutine normalize( vector, size) {
    real, dimension (:), intent (inout) :: vector
    double precision,    intent (out)   :: size
    size = sqrt( sum( dprod( vector, vector)))
    vector = vector / size
    }
}
```

2.6.5 Understanding CG magic and advanced methods

This section includes Sergey Fomel's explanation on the "magic" convergence properties of the conjugate-direction methods. It also presents a classic version of conjugate gradients, which can be found in numerous books on least-square optimization. The key idea for constructing an optimal iteration is to update the solution at each step in the direction, composed by a linear combination of the current direction and all previous solution steps. To see why this is a helpful idea, let us consider first the method of random directions. Substituting expression (1.47) into formula (1.45), we see that the residual power decreases at each step by

$$\mathbf{r}\cdot\mathbf{r} - \mathbf{r}_{\rm new}\cdot\mathbf{r}_{\rm new} \quad = \quad \frac{(\mathbf{r}\cdot\Delta\mathbf{r})^2}{(\Delta\mathbf{r}\cdot\Delta\mathbf{r})} \; . \tag{2.90}$$

To achieve a better convergence, we need to maximize the right hand side of (1.90). Let us define a new solution step $\mathbf{s}_{\rm new}$ as a combination of the current direction $\Delta\mathbf{x}$ and the previous step $\mathbf{s}$, as follows:

$$\mathbf{s}_{\rm new} \quad = \quad \Delta\mathbf{x} + \beta\mathbf{s} \; . \tag{2.91}$$

The solution update is then defined as

$$\mathbf{x}_{\rm new} \quad = \quad \mathbf{x} + \alpha\mathbf{s}_{\rm new} \; . \tag{2.92}$$

The formula for α (1.47) still holds, because we have preserved in (1.92) the form of equation (1.41) and just replaced $\Delta\mathbf{x}$ with $\mathbf{s}_{\rm new}$. In fact, formula (1.47) can be simplified a little bit. From (1.46), we know that $\mathbf{r}_{\rm new}$ is orthogonal to $\Delta\mathbf{r} = \mathbf{F}\mathbf{s}_{\rm new}$. Likewise, $\mathbf{r}$ should be orthogonal to $\mathbf{F}\mathbf{s}$ (recall that $\mathbf{r}$ was $\mathbf{r}_{\rm new}$ and $\mathbf{s}$ was $\mathbf{s}_{\rm new}$ at the previous iteration). We can conclude that

$$(\mathbf{r}\cdot\Delta\mathbf{r}) \quad = \quad (\mathbf{r}\cdot\mathbf{F}\mathbf{s}_{\rm new}) \quad = \quad (\mathbf{r}\cdot\mathbf{F}\Delta\mathbf{x}) + \beta(\mathbf{r}\cdot\mathbf{F}\mathbf{s}) \quad = \quad (\mathbf{r}\cdot\mathbf{F}\Delta\mathbf{x}) \; . \tag{2.93}$$

Comparing (1.93) with (1.90), we can see that adding a portion of the previous step to the current direction does not change the value of the numerator in expression (1.90). However, the value of the denominator can be changed. Minimizing the denominator maximizes the residual increase at each step and leads to a faster convergence. This is the denominator minimization that constrains the value of the adjustable coefficient β in (1.91).

The procedure for finding β is completely analogous to the derivation of formula (1.47). We start with expanding the dot product $(\Delta\mathbf{r}\cdot\Delta\mathbf{r})$:

$$(\mathbf{F}\mathbf{s}_{\rm new}\cdot\mathbf{F}\mathbf{s}_{\rm new}) \quad = \quad \mathbf{F}\Delta\mathbf{x}\cdot\mathbf{F}\Delta\mathbf{x} + 2\beta(\mathbf{F}\Delta\mathbf{x}\cdot\mathbf{F}\mathbf{s}) + \beta^2\,\mathbf{F}\mathbf{s}\cdot\mathbf{F}\mathbf{s} \; . \tag{2.94}$$

Differentiating with respect to β and setting the derivative to zero, we find that

$$0 \quad = \quad 2(\mathbf{F}\Delta\mathbf{x} + \beta\mathbf{F}\mathbf{s})\cdot\mathbf{F}\mathbf{s} \; . \tag{2.95}$$

Equation (1.95) states that the *conjugate direction* $\mathbf{Fs}_{\text{new}}$ is orthogonal (perpendicular) to the previous conjugate direction $\mathbf{Fs}$. It also defines the value of β as

$$\beta = -\frac{(\mathbf{F}\Delta\mathbf{x} \cdot \mathbf{Fs})}{(\mathbf{Fs} \cdot \mathbf{Fs})} \ . \tag{2.96}$$

Can we do even better? The positive quantity that we minimized in (1.94) decreased by

$$\mathbf{F}\Delta\mathbf{x} \cdot \mathbf{F}\Delta\mathbf{x} - \mathbf{Fs}_{\text{new}} \cdot \mathbf{Fs}_{\text{new}} = \frac{(\mathbf{F}\Delta\mathbf{x} \cdot \mathbf{Fs})^2}{(\mathbf{Fs} \cdot \mathbf{Fs})} \tag{2.97}$$

Can we decrease it further by adding another previous step? In general, the answer is positive, and it defines the method of conjugate directions. I will state this result without a formal proof (which uses the method of mathematical induction).

- If the new step is composed of the current direction and a combination of all the previous steps:

$$\mathbf{s}_n = \Delta\mathbf{x}_n + \sum_{i<n} \beta_i \mathbf{s}_i \ , \tag{2.98}$$

  then the optimal convergence is achieved when

$$\beta_i = -\frac{(\mathbf{F}\Delta\mathbf{x}_n \cdot \mathbf{Fs}_i)}{(\mathbf{Fs}_i \cdot \mathbf{Fs}_i)} \ . \tag{2.99}$$

- The new conjugate direction is orthogonal to the previous ones:

$$(\mathbf{Fs}_n \cdot \mathbf{Fs}_i) = 0 \quad \text{for all} \quad i < n \tag{2.100}$$

To see why this is an optimally convergent method, it is sufficient to notice that vectors $\mathbf{Fs}_i$ form an orthogonal basis in the data space. The vector from the current residual to the smallest residual also belongs to that space. If the data size is n, then n basis components (at most) are required to represent this vector, hence no more then n conjugate-direction steps are required to find the solution.

The computation template for the method of conjugate directions is

$$
\begin{array}{l}
\mathbf{r} \quad \longleftarrow \quad \mathbf{Fx} - \mathbf{d} \\
\text{iterate } \{ \\
\qquad \Delta\mathbf{x} \quad \longleftarrow \quad \text{random numbers} \\
\qquad \mathbf{s} \quad \longleftarrow \quad \Delta\mathbf{x} + \sum_{i<n} \beta_i \mathbf{s}_i \quad \text{where} \quad \beta_i = -\frac{(\mathbf{F}\Delta\mathbf{x} \cdot \mathbf{Fs}_i)}{(\mathbf{Fs}_i \cdot \mathbf{Fs}_i)} \\
\qquad \Delta\mathbf{r} \quad \longleftarrow \quad \mathbf{Fs} \\
\qquad \alpha \quad \longleftarrow \quad -(\mathbf{r} \cdot \Delta\mathbf{r})/(\Delta\mathbf{r} \cdot \Delta\mathbf{r}) \\
\qquad \mathbf{x} \quad \longleftarrow \quad \mathbf{x} + \alpha\mathbf{s} \\
\qquad \mathbf{r} \quad \longleftarrow \quad \mathbf{r} + \alpha\Delta\mathbf{r} \\
\qquad \}
\end{array}
$$

What happens if we "feed" the method with gradient directions instead of just random directions? It turns out that in this case we need to remember from all the previous steps $\mathbf{s}_i$ only the one that immediately precedes the current iteration. Let us derive a formal proof of that fact as well as some other useful formulas related to the method of *conjugate gradients*.

According to formula (1.46), the new residual $\mathbf{r}_{\rm new}$ is orthogonal to the conjugate direction $\Delta \mathbf{r} = \mathbf{F} \mathbf{s}_{\rm new}$. According to the orthogonality condition (1.100), it is also orthogonal to all the previous conjugate directions. Defining $\Delta \mathbf{x}$ equal to the gradient $\mathbf{F}'\mathbf{r}$ and applying the definition of the adjoint operator, it is convenient to rewrite the orthogonality condition in the form

$$0 \quad = \quad (\mathbf{r}_n \cdot \mathbf{F} \mathbf{s}_i) \quad = \quad (\mathbf{F}' \mathbf{r}_n \cdot \mathbf{s}_i) \quad = \quad (\Delta \mathbf{x}_{n+1} \cdot \mathbf{s}_i) \quad \text{for all} \quad i \leq n \tag{2.101}$$

According to formula (1.98), each solution step $\mathbf{s}_i$ is just a linear combination of the gradient $\Delta \mathbf{x}_i$ and the previous solution steps. We deduce from formula (1.101) that

$$0 \quad = \quad (\Delta \mathbf{x}_n \cdot \mathbf{s}_i) \quad = \quad (\Delta \mathbf{x}_n \cdot \Delta \mathbf{x}_i) \quad \text{for all} \quad i < n \tag{2.102}$$

In other words, in the method of conjugate gradients, the current gradient direction is always orthogonal to all the previous directions. The iteration process constructs not only an orthogonal basis in the data space but also an orthogonal basis in the model space, composed of the gradient directions.

Now let us take a closer look at formula (1.99). Note that $\mathbf{F} \mathbf{s}_i$ is simply related to the residual step at i-th iteration:

$$\mathbf{F} \mathbf{s}_i = \frac{\mathbf{r}_i - \mathbf{r}_{i-1}}{\alpha_i} \; . \tag{2.103}$$

Substituting relationship (1.103) into formula (1.99) and applying again the definition of the adjoint operator, we obtain

$$\beta_i = - \frac{\mathbf{F} \Delta \mathbf{x}_n \cdot (\mathbf{r}_i - \mathbf{r}_{i-1})}{\alpha_i (\mathbf{F} \mathbf{s}_i \cdot \mathbf{F} \mathbf{s}_i)} = - \frac{\Delta \mathbf{x}_n \cdot \mathbf{F}' (\mathbf{r}_i - \mathbf{r}_{i-1})}{\alpha_i (\mathbf{F} \mathbf{s}_i \cdot \mathbf{F} \mathbf{s}_i)} = - \frac{\Delta \mathbf{x}_n \cdot (\Delta \mathbf{x}_{i+1} - \Delta \mathbf{x}_i)}{\alpha_i (\mathbf{F} \mathbf{s}_i \cdot \mathbf{F} \mathbf{s}_i)} \tag{2.104}$$

Since the gradients $\Delta \mathbf{x}_i$ are orthogonal to each other, the dot product in the numerator is equal to zero unless $i = n-1$. It means that only the immediately preceding step $\mathbf{s}_{n-1}$ contributes to the definition of the new solution direction $\mathbf{s}_n$ in (1.98). This is precisely the property of the conjugate gradient method we wanted to prove.

To simplify formula (1.104), rewrite formula (1.47) as

$$\alpha_i \quad = \quad - \frac{(\mathbf{r}_{i-1} \cdot \mathbf{F} \Delta \mathbf{x}_i)}{(\mathbf{F} \mathbf{s}_i \cdot \mathbf{F} \mathbf{s}_i)} \quad = \quad - \frac{(\mathbf{F}' \mathbf{r}_{i-1} \cdot \Delta \mathbf{x}_i)}{(\mathbf{F} \mathbf{s}_i \cdot \mathbf{F} \mathbf{s}_i)} \quad = \quad - \frac{(\Delta \mathbf{x}_i \cdot \Delta \mathbf{x}_i)}{(\mathbf{F} \mathbf{s}_i \cdot \mathbf{F} \mathbf{s}_i)} \tag{2.105}$$

Substituting (1.105) into (1.104), we obtain

$$\beta = - \frac{(\Delta \mathbf{x}_n \cdot \Delta \mathbf{x}_n)}{\alpha_{n-1} (\mathbf{F} \mathbf{s}_{n-1} \cdot \mathbf{F} \mathbf{s}_{n-1})} = - \frac{(\Delta \mathbf{x}_n \cdot \Delta \mathbf{x}_n)}{(\Delta \mathbf{x}_{n-1} \cdot \Delta \mathbf{x}_{n-1})} \; . \tag{2.106}$$

The computation template for the method of conjugate gradients is then

$$
\begin{aligned}
&\mathbf{r} \quad \longleftarrow \quad \mathbf{F}\mathbf{x} - \mathbf{d} \\
&\beta \quad \longleftarrow \quad 0 \\
&\text{iterate } \{ \\
&\qquad \Delta\mathbf{x} \quad \longleftarrow \quad \mathbf{F}'\mathbf{r} \\
&\qquad \text{if not the first iteration } \beta \quad \longleftarrow \quad -\frac{(\Delta\mathbf{x}\cdot\Delta\mathbf{x})}{\gamma} \\
&\qquad \gamma \quad \longleftarrow \quad (\Delta\mathbf{x}\cdot\Delta\mathbf{x}) \\
&\qquad \mathbf{s} \quad \longleftarrow \quad \Delta\mathbf{x} + \beta\mathbf{s} \\
&\qquad \Delta\mathbf{r} \quad \longleftarrow \quad \mathbf{F}\mathbf{s} \\
&\qquad \alpha \quad \longleftarrow \quad -\gamma/(\Delta\mathbf{r}\cdot\Delta\mathbf{r}) \\
&\qquad \mathbf{x} \quad \longleftarrow \quad \mathbf{x} + \alpha\mathbf{s} \\
&\qquad \mathbf{r} \quad \longleftarrow \quad \mathbf{r} + \alpha\Delta\mathbf{r} \\
&\qquad \}
\end{aligned}
$$

Module `conjgrad` on the current page provides an implementation of this method. The interface is exactly similar to that of `cgstep` on page 15, therefore you can use `conjgrad` as an argument to `solver` on page 17. When the orthogonality of the gradients, (implied by the classical conjugate-gradient method) is not numerically assured, the `conjgrad` algorithm may loose its convergence properties. This problem does not exist in the algebraic derivations, but appears in practice because of numerical errors. A proper remedy is to orthogonalize each new gradient against previous ones. Naturally, this increases the cost and memory requirements of the method.

```
module conjgrad_mod {
  real, dimension (:), allocatable, private     :: s, ss
contains
  subroutine conjgrad_close ()  {
        if( allocated( s))  deallocate( s, ss)
        }
  function conjgrad( forget, x, g, rr, gg)  result( stat) {
    integer              :: stat
    real, dimension (:)  :: x, g, rr, gg
    logical              :: forget
    real, save           :: rnp
    double precision     :: rn, alpha, beta
    rn = sum( dprod( g, g))
    if( .not. allocated( s)) {   forget = .true.
          allocate( s  (size (x )));  s  = 0.
          allocate( ss (size (rr)));  ss = 0.
          }
    if( forget .or. rnp < epsilon (rnp))
          alpha = 0.d0
    else
          alpha = - rn / rnp
    s  =  g + alpha *  s
    ss = gg + alpha * ss
    beta = sum( dprod( ss, ss))
    if( beta > epsilon( beta)) {
          alpha = rn / beta
          x =  x  + alpha *  s
          rr = rr + alpha * ss
```

```
          }
    rnp = rn;    forget = .false.;   stat = 0
    }
}
```

2.7 REFERENCES

Gill, P.E., Murray, W., and Wright, M.H., 1981, Practical optimization: Academic Press.

Hestenes, M.R., and Stiefel, E., 1952, Methods of conjugate gradients for solving linear systems: J. Res. Natl. Bur. Stand., **49**, 409-436.

Luenberger, D.G., 1973, Introduction to linear and nonlinear programming: Addison-Wesley.

Nolet, G., 1985, Solving or resolving inadequate and noisy tomographic systems: J. Comp. Phys., **61**, 463-482.

Paige, C.C., and Saunders, M.A., 1982a, LSQR: an algorithm for sparse linear equations and sparse least squares: Assn. Comp. Mach. Trans. Mathematical Software, **8,** 43-71.

Paige, C.C., and Saunders, M.A., 1982b, Algorithm 583, LSQR: sparse linear equations and least squares problems: Assn. Comp. Mach. Trans. Mathematical Software, **8,** 195-209.

Press, W.H. et al., 1989, Numerical recipes: the art of scientific computing: Cambridge University Press.

Saad, Y., 1996, Iterative methods for sparse linear systems: PWS Publishing Company

Strang, G., 1986, Introduction to applied mathematics: Wellesley-Cambridge Press.

Index