Chapter 1

Adjoint operators

A great many of the calculations we do in science and engineering are really matrix multiplication in disguise. The first goal of this chapter is to unmask the disguise by showing many examples. Second, we see how the **adjoint** operator (matrix transpose) back-projects information from data to the underlying model.

Geophysical modeling calculations generally use linear operators that predict data from models. Our usual task is to find the inverse of these calculations; i.e., to find models (or make maps) from the data. Logically, the adjoint is the first step and a part of all subsequent steps in this **inversion** process. Surprisingly, in practice the adjoint sometimes does a better job than the inverse! This is because the adjoint operator tolerates imperfections in the data and does not demand that the data provide full information.

Using the methods of this chapter, you will find that once you grasp the relationship between operators in general and their adjoints, you can obtain the adjoint just as soon as you have learned how to code the modeling operator.

If you will permit me a poet's license with words, I will offer you the following table of **operator**s and their **adjoint**s:

matrix multiply	conjugate-transpose matrix multiply
convolve	crosscorrelate
truncate	zero pad
replicate, scatter, spray	sum or stack
spray into neighborhood	sum in bins
derivative (slope)	negative derivative
causal integration	anticausal integration
add functions	do integrals
assignment statements	added terms
plane-wave superposition	slant stack / beam form
superpose on a curve	sum along a curve
stretch	squeeze
upward continue	downward continue

hyperbolic modeling	normal moveout and CDP stack
diffraction modeling	imaging by migration
ray tracing	**tomography**

The left column above is often called "**modeling**," and the adjoint operators on the right are often used in "data **processing**."

The adjoint operator is sometimes called the "**back projection**" operator because information propagated in one direction (earth to data) is projected backward (data to earth model). For complex-valued operators, the transpose goes together with a complex conjugate. In **Fourier analysis**, taking the complex conjugate of $\exp(i\omega t)$ reverses the sense of time. With more poetic license, I say that adjoint operators *undo* the time and phase shifts of modeling operators. The inverse operator does this too, but it also divides out the color. For example, when linear interpolation is done, then high frequencies are smoothed out, so inverse interpolation must restore them. You can imagine the possibilities for noise amplification. That is why adjoints are safer than inverses.

Later in this chapter we relate adjoint operators to inverse operators. Although inverse operators are more well known than adjoint operators, the inverse is built upon the adjoint so the adjoint is a logical place to start. Also, computing the inverse is a complicated process fraught with pitfalls whereas the computation of the adjoint is easy. It's a natural companion to the operator itself.

Much later in this chapter is a formal definition of adjoint operator. Throughout the chapter we handle an adjoint operator as a matrix transpose, but we hardly ever write down any matrices or their transposes. Instead, we always prepare two subroutines, one that performs $\mathbf{y} = \mathbf{A}\mathbf{x}$ and another that performs $\tilde{\mathbf{x}} = \mathbf{A}'\mathbf{y}$. So we need a test that the two subroutines really embody the essential aspects of matrix transposition. Although the test is an elegant and useful test and is itself a fundamental definition, curiously, that definition does not help construct adjoint operators, so we postpone a formal definition of adjoint until after we have seen many examples.

1.1 FAMILIAR OPERATORS

The operation $y_i = \sum_j b_{ij} x_j$ is the multiplication of a matrix $\mathbf{B}$ by a vector $\mathbf{x}$. The adjoint operation is $\tilde{x}_j = \sum_i b_{ij} y_i$. The operation adjoint to multiplication by a matrix is multiplication by the transposed matrix (unless the matrix has complex elements, in which case we need the complex-conjugated transpose). The following **pseudocode** does matrix multiplication $\mathbf{y} = \mathbf{B}\mathbf{x}$ and multiplication by the transpose $\tilde{\mathbf{x}} = \mathbf{B}'\mathbf{y}$:

```
if operator itself
        then erase y
if adjoint
        then erase x
do iy = 1, ny {
do ix = 1, nx {
        if operator itself
                y(iy) = y(iy) + b(iy,ix) × x(ix)
        if adjoint
                x(ix) = x(ix) + b(iy,ix) × y(iy)
}       }
```

Notice that the "bottom line" in the program is that x and y are simply interchanged. The above example is a prototype of many to follow, so observe carefully the similarities and differences between the operation and its adjoint.

A formal subroutine[1] for **matrix multiply** and its adjoint is found below. The first step is a subroutine, `adjnull()`, for optionally erasing the output. With the option `add=1`, results accumulate like `y=y+B*x`.

```
subroutine adjnull( adj, add, x, nx,  y, ny )
integer ix, iy,     adj, add,    nx,     ny
real                          x( nx), y( ny )
if( add == 0 )
        if( adj == 0 )
                do iy= 1, ny
                        y(iy) = 0.
        else
                do ix= 1, nx
                        x(ix) = 0.
return; end
```

The subroutine `matmult()` for matrix multiply and its adjoint exhibits the style that we will use repeatedly.

```
# matrix multiply and its adjoint
#
subroutine matmult( adj, add, bb,        x,nx,  y,ny)
integer ix, iy,     adj, add,              nx,    ny
real                          bb(ny,nx), x(nx), y(ny)
call adjnull(       adj, add,            x,nx,  y,ny)
do ix= 1, nx {
do iy= 1, ny {
        if( adj == 0 )
                        y(iy) = y(iy) + bb(iy,ix) * x(ix)
```

[1]The programming language used in this book is Ratfor, a dialect of Fortran. For more details, see Appendix A.

```
        else
                        x(ix) = x(ix) + bb(iy,ix) * y(iy)
        }}
return; end
```

Sometimes a matrix operator reduces to a simple row or a column.

A **row** is a summation operation.

A **column** is an impulse response.

If the inner loop of a matrix multiply ranges within a

row, the operator is called *sum* or *pull*.

column, the operator is called *spray* or *push*.

A basic aspect of adjointness is that the adjoint of a row matrix operator is a column matrix operator. For example, the row operator $[a,b]$

$$y \quad = \quad [\, a \; b \,] \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad = \quad a x_1 + b x_2 \tag{1.1}$$

has an adjoint that is two assignments:

$$\begin{bmatrix} \hat{x}_1 \\ \hat{x}_2 \end{bmatrix} \quad = \quad \begin{bmatrix} a \\ b \end{bmatrix} y \tag{1.2}$$

The adjoint of a sum of N terms is a collection of N assignments.

1.1.1 Adjoint derivative

Given a sampled signal, its time **derivative** can be estimated by convolution with the filter $(1,-1)/\Delta t$, expressed as the matrix-multiply below:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} \quad = \quad \begin{bmatrix} -1 & 1 & . & . & . & . \\ . & -1 & 1 & . & . & . \\ . & . & -1 & 1 & . & . \\ . & . & . & -1 & 1 & . \\ . & . & . & . & -1 & 1 \\ . & . & . & . & . & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} \tag{1.3}$$

Technically the output should be `n-1` points long, but I appended a zero row, a small loss of logical purity, so that the size of the output vector will match that of the input. This is a convenience for plotting and for simplifying the assembly of other operators building on this one.

The **filter impulse response** is seen in any column in the middle of the matrix, namely $(1,-1)$. In the transposed matrix, the filter-impulse response is time-reversed to $(-1,1)$. So, mathematically, we can say that the adjoint of the time derivative operation is the negative time derivative. This corresponds also to the fact that the complex conjugate of $-i\omega$ is $i\omega$. We can also speak of the adjoint of the boundary conditions: we might say that the adjoint of "no boundary condition" is a "specified value" boundary condition.

A complicated way to think about the adjoint of equation (1.3) is to note that it is the negative of the derivative and that something must be done about the ends. A simpler way to think about it is to apply the idea that the adjoint of a sum of N terms is a collection of N assignments. This is done in subroutine `igrad1()`, which implements equation (1.3) and its adjoint.

```
subroutine igrad1( adj, add,  xx,n,   yy   )
integer i,        adj, add,     n
real                          xx(n), yy(n)
call adjnull(     adj, add,  xx,n,   yy,n )
do i= 1, n-1 {
        if( adj == 0 )
                yy(i) = yy(i) + xx(i+1) - xx(i)
        else {
                xx(i+1) = xx(i+1) + yy(i)
                xx(i  ) = xx(i  ) - yy(i)
                }
        }
return; end
```

Notice that the do loop in the code covers all the outputs for the operator itself, and that in the adjoint operation it gathers all the inputs. This is natural because in switching from operator to adjoint, the outputs switch to inputs.

As you look at the code, think about matrix elements being $+1$ or -1 and think about the forward operator "pulling" a sum into `yy(i)`, and think about the adjoint operator "pushing" or "spraying" the impulse `yy(i)` back into `xx()`.

You might notice that you can simplify the program by merging the "erase output" activity with the calculation itself. We will not do this optimization however because in many applications we do not want to include the "erase output" activity. This often happens when we build complicated operators from simpler ones.

1.1.2 Zero padding is the transpose of truncation

Surrounding a dataset by zeros (**zero pad**ding) is adjoint to throwing away the extended data (**truncation**). Let us see why this is so. Set a signal in a vector $\mathbf{x}$, and then to make a longer vector $\mathbf{y}$, add some zeros at the end of $\mathbf{x}$. This zero padding can be regarded as the matrix multiplication

$$\mathbf{y} \quad = \quad \begin{bmatrix} \mathbf{I} \\ \mathbf{0} \end{bmatrix} \mathbf{x} \tag{1.4}$$

The matrix is simply an identity matrix $\mathbf{I}$ above a zero matrix $\mathbf{0}$. To find the transpose to zero-padding, we now transpose the matrix and do another matrix multiply:

$$\tilde{\mathbf{x}} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \end{bmatrix} \mathbf{y} \tag{1.5}$$

So the transpose operation to zero padding data is simply *truncating* the data back to its original length. Subroutine `zpad1()` below pads zeros on both ends of its input. Subroutines for two- and three-dimensional padding are in the library named `zpad2()` and `zpad3()`.

```
# Zero pad.  Surround data by zeros. 1-D
#
subroutine zpad1( adj,add,  data,nd,  padd,np )
integer           adj,add,    d, nd,    p, np
real                        data(nd), padd(np)
call adjnull(     adj,add,  data,nd,  padd,np)
do d= 1, nd {                 p = d + (np-nd)/2
        if( adj == 0 )
                        padd(p) = padd(p) + data(d)
        else
                        data(d) = data(d) + padd(p)
        }
return; end
```

1.1.3 Adjoints of products are reverse-ordered products of adjoints

Here we examine an example of the general idea that adjoints of products are reverse-ordered products of adjoints. For this example we use the Fourier transformation. No details of **Fourier transformation** are given here and we merely use it as an example of a square matrix $\mathbf{F}$. We denote the complex-conjugate transpose (or **adjoint**) matrix with a prime, i.e., $\mathbf{F}'$. The adjoint arises naturally whenever we consider energy. The statement that Fourier transforms conserve energy is $\mathbf{y}'\mathbf{y} = \mathbf{x}'\mathbf{x}$ where $\mathbf{y} = \mathbf{F}\mathbf{x}$. Substituting gives $\mathbf{F}'\mathbf{F} = \mathbf{I}$, which shows that the inverse matrix to Fourier transform happens to be the complex conjugate of the transpose of $\mathbf{F}$.

With Fourier transforms, **zero pad**ding and **truncation** are especially prevalent. Most subroutines transform a dataset of length of 2^n, whereas dataset lengths are often of length $m \times 100$. The practical approach is therefore to pad given data with zeros. Padding followed by Fourier transformation $\mathbf{F}$ can be expressed in matrix algebra as

$$\text{Program} = \mathbf{F} \begin{bmatrix} \mathbf{I} \\ \mathbf{0} \end{bmatrix} \tag{1.6}$$

According to matrix algebra, the transpose of a product, say $\mathbf{AB} = \mathbf{C}$, is the product $\mathbf{C}' = \mathbf{B}'\mathbf{A}'$ in reverse order. So the adjoint subroutine is given by

$$\text{Program}' = \begin{bmatrix} \mathbf{I} & \mathbf{0} \end{bmatrix} \mathbf{F}' \tag{1.7}$$

Thus the adjoint subroutine *truncates* the data *after* the inverse Fourier transform. This concrete example illustrates that common sense often represents the mathematical abstraction that adjoints of products are reverse-ordered products of adjoints. It is also nice to see a formal mathematical notation for a practical necessity. Making an approximation need not lead to collapse of all precise analysis.

1.1.4 Nearest-neighbor coordinates

In describing physical processes, we often either specify models as values given on a uniform mesh or we record data on a uniform mesh. Typically we have a function f of time t or depth z and we represent it by `f(iz)` corresponding to $f(z_i)$ for $i = 1,2,3,\ldots,n_z$ where $z_i = z_0 + (i-1)\Delta z$. We sometimes need to handle depth as an integer counting variable i and we sometimes need to handle it as a floating-point variable z. Conversion from the counting variable to the floating-point variable is exact and is often seen in a computer idiom such as either of

```
do iz= 1, nz {    z = z0 + (iz-1) * dz
do i3= 1, n3 {   x3 = o3 + (i3-1) * d3
```

The reverse conversion from the floating-point variable to the counting variable is inexact. The easiest thing is to place it at the nearest neighbor. This is done by solving for `iz`, then adding one half, and then rounding down to the nearest integer. The familiar computer idioms are:

```
iz =  .5 + 1 + ( z - z0) / dz
iz = 1.5 +     ( z - z0) / dz
i3 = 1.5 +     (x3 - o3) / d3
```

A small warning is in order: People generally use positive counting variables. If you also include negative ones, then to get the nearest integer, you should do your rounding with the Fortran function `NINT()`.

1.1.5 Data-push binning

Binning is putting data values in bins. Nearest-neighbor binning is an operator. There is both a forward operator and its adjoint. Normally the model consists of values given on a uniform mesh, and the data consists of pairs of numbers (ordinates at coordinates) sprinkled around in the continuum (although sometimes the data is uniformly spaced and the model is not).

In both the forward and the adjoint operation, each data coordinate is examined and the nearest mesh point (the bin) is found. For the forward operator, the value of the bin is added to that of the data. The adjoint is the reverse: we add the value of the data to that of the bin. Both are shown in two dimensions in subroutine `dpbin2()`.

```
# Data-push binning in 2-D.
#
```

```
subroutine dpbin2 ( adj, add, o1,d1,o2,d2, xy,       mm,m1,m2,  dd, nd)
integer      i1,i2, adj, add,                  id,     m1,m2,      nd
real                       o1,d1,o2,d2, xy(2,nd), mm(m1,m2), dd( nd)
call adjnull(       adj, add,                            mm,m1*m2,  dd, nd)
do id=1,nd {
        i1 = 1.5 + (xy(1,id)-o1)/d1
        i2 = 1.5 + (xy(2,id)-o2)/d2
        if( 1<=i1 && i1<=m1 &&
            1<=i2 && i2<=m2   )
                if( adj == 0)
                        dd( id)   = dd( id)   +  mm(i1,i2)
                else
                        mm(i1,i2) = mm(i1,i2) +  dd( id)
        }
return; end
```

The most typical application requires an additional step, inversion. In the inversion applications each bin contains a different number of data values. After the adjoint operation is performed, the inverse operator divides the bin value by the number of points in the bin. It is this inversion operator that is generally called binning. To find the number of data points in a bin, we can simply apply the adjoint of `dpbin2()` to pseudo data of all ones.

1.1.6 Linear interpolation

The **linear interpolation** operator is much like the binning operator but a little fancier. When we perform the forward operation, we take each data coordinate and see which two model mesh points bracket it. Then we pick up the two bracketing model values and weight each of them in proportion to their nearness to the data coordinate, and add them to get the data value (ordinate). The adjoint operation is adding a data value back into the model vector; using the same two weights, this operation distributes the ordinate value between the two nearest points in the model vector. For example, suppose we have a data point near each end of the model and a third data point exactly in the middle. Then for a model space 6 points long, as shown in Figure 1.1, we have the operator in (1.8).

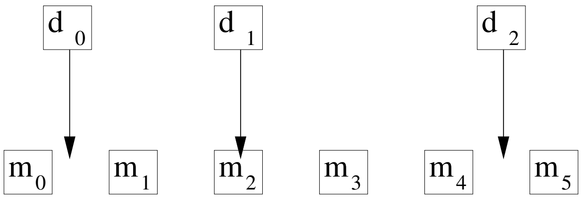

Figure 1.1: Uniformly sampled model space and irregularly sampled data space corresponding to (1.8). conj-helgerud [NR]

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix} \approx \begin{bmatrix} .8 & .2 & . & . & . & . \\ . & . & 1 & . & . & . \\ . & . & . & . & .5 & .5 \end{bmatrix} \begin{bmatrix} m_0 \\ m_1 \\ m_2 \\ m_3 \\ m_4 \\ m_5 \end{bmatrix} \tag{1.8}$$

The two weights in each row sum to unity. If a binning operator were used for the same data and model, the binning operator would contain a "1." in each row. In one dimension (as here), data coordinates are often sorted into sequence, so that the matrix is crudely a diagonal matrix like equation (1.8). If the data coordinates covered the model space uniformly, the adjoint would roughly be the inverse. Otherwise, when data values pile up in some places and gaps remain elsewhere, the adjoint would be far from the inverse.

Subroutine `lint1()` does linear interpolation and its adjoint.

```
# Linear interpolation 1-D, uniform model mesh to data coordinates and values.
#
subroutine lint1( adj, add, o1,d1,coordinate,     mm,m1,  dd, nd)
integer i, im,    adj, add,                 id,     m1,      nd
real    f, fx,gx,            o1,d1,coordinate(nd), mm(m1), dd( nd)
call adjnull(     adj, add,                     mm,m1,  dd, nd)
do id= 1, nd {
        f = (coordinate(id)-o1)/d1;     i=f;     im= 1+i
        if( 1<=im && im<m1) {          fx=f-i;   gx= 1.-fx
                if( adj == 0)
                        dd(id) = dd(id) + gx * mm(im) + fx * mm(im+1)
                else {
                        mm(im  ) = mm(im  ) + gx * dd(id)
                        mm(im+1) = mm(im+1) + fx * dd(id)
                        }
                }
        }
return; end
```

1.1.7 Causal integration

Causal integration is defined as

$$y(t) \quad = \quad \int_{-\infty}^{t} x(t)\, dt \tag{1.9}$$

Sampling the time axis gives a matrix equation which we should call causal summation, but we often call it causal integration.

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \end{bmatrix} \quad = \quad \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{bmatrix} \tag{1.10}$$

(In some applications the 1 on the diagonal is replaced by 1/2.) Causal integration is the simplest prototype of a recursive operator. The coding is trickier than operators we considered earlier. Notice when you compute y_5 that it is the sum of 6 terms, but that this sum is more quickly computed as $y_5 = y_4 + x_5$. Thus equation (1.10) is more efficiently thought of as the recursion

$$y_t \quad = \quad y_{t-1} + x_t \qquad \text{for increasing } t \tag{1.11}$$

(which may also be regarded as a numerical representation of the **differential equation** $dy/dt = x$.)

When it comes time to think about the adjoint, however, it is easier to think of equation (1.10) than of (1.11). Let the matrix of equation (1.10) be called $\mathbf{C}$. Transposing to get $\mathbf{C}'$ and applying it to $\mathbf{y}$ gives us something back in the space of $\mathbf{x}$, namely $\tilde{\mathbf{x}} = \mathbf{C}'\mathbf{y}$. From it we see that the adjoint calculation, if done recursively, needs to be done backwards like

$$\tilde{x}_{t-1} \quad = \quad \tilde{x}_t + y_{t-1} \qquad \text{for decreasing } t \tag{1.12}$$

We can sum up by saying that the adjoint of causal integration is anticausal integration.

A subroutine to do these jobs is `causint()` on this page. The code for anticausal integration is not obvious from the code for integration and the adjoint coding tricks we learned earlier. To understand the adjoint, you need to inspect the detailed form of the expression $\tilde{\mathbf{x}} = \mathbf{C}'\mathbf{y}$ and take care to get the ends correct.

```
# causal integration  (1's on diagonal)
#
subroutine causint( adj, add,        n,xx,   yy   )
integer i,       n, adj, add;   real xx(n), yy(n )
temporary real tt( n)
call adjnull(       adj, add,        xx,n,   yy,n )
if( adj == 0){                                tt(1) = xx(1)
                                        do i= 2, n
                                                tt(i) = tt(i-1) + xx(i)
        do i= 1, n
                yy(i) = yy(i) + tt(i)
        }
else {                                          tt(n) = yy(n)
                                        do i= n, 2, -1
                                                tt(i-1) = tt(i) + yy(i-1)
        do i= 1, n
                xx(i) = xx(i) + tt(i)
        }
return; end
```

Later we will consider equations to march wavefields up towards the earth's surface, a layer at a time, an operator for each layer. Then the adjoint will start from the earth's surface and march down, a layer at a time, into the earth.

EXERCISES:

1 Modify the calculation in Figure 1.2 to make a triangle waveform on the bottom row.

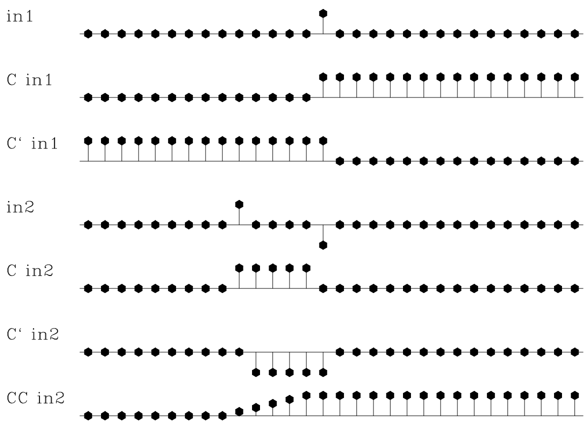

Figure 1.2: `in1` is an input pulse. `C in1` is its causal integral. `C′ in1` is the anticausal integral of the pulse. `in2` is a separated doublet. Its causal integration is a box and its anticausal integration is the negative. `CC in2` is the double causal integral of `in2`. How can an equilateral triangle be built? conj-causint [ER]

1.2 ADJOINTS AND INVERSES

Consider a model $\mathbf{m}$ and an operator $\mathbf{F}$ which creates some theoretical data $\mathbf{d}_{\rm theor}$.

$$\mathbf{d}_{\rm theor} \quad = \quad \mathbf{Fm} \tag{1.13}$$

The general task of geophysicists is to begin from observed data $\mathbf{d}_{\rm obs}$ and find an estimated model $\mathbf{m}_{\rm est}$ that satisfies the simultaneous equations

$$\mathbf{d}_{\rm obs} \quad = \quad \mathbf{Fm}_{\rm est} \tag{1.14}$$

This is the topic of a large discipline variously called "inversion" or "estimation". Basically, it defines a residual $\mathbf{r} = \mathbf{d}_{\rm obs} - \mathbf{d}_{\rm theor}$ and then minimizes its length $\mathbf{r}\cdot\mathbf{r}$. Finding $\mathbf{m}_{\rm est}$ this way is called the **least squares** method. The basic result (not proven here) is that

$$\mathbf{m}_{\rm est} = (\mathbf{F}'\mathbf{F})^{-1}\mathbf{F}'\mathbf{d}_{\rm obs} \tag{1.15}$$

In many cases including all seismic imaging cases, the matrix $\mathbf{F}'\mathbf{F}$ is far too large to be invertible. People generally proceed by a rough guess at an approximation for $(\mathbf{F}'\mathbf{F})^{-1}$. The usual first approximation is the optimistic one that $(\mathbf{F}'\mathbf{F})^{-1} = \mathbf{I}$. To this happy approximation, the inverse $\mathbf{F}^{-1}$ is the adjoint $\mathbf{F}'$.

In this book we'll see examples where $\mathbf{F}'\mathbf{F} \approx \mathbf{I}$ is a good approximation and other examples where it isn't. We can tell how good the approximation is. We take some hypothetical data and convert it to a model, and use that model to make some reconstructed data $\mathbf{d}_{\rm recon} = \mathbf{FF}'\mathbf{d}_{\rm hypo}$. Likewise we could go from a hypothetical model to some data and then to a reconstructed model $\mathbf{m}_{\rm recon} = \mathbf{F}'\mathbf{Fm}_{\rm hypo}$. Luckily, it often happens that the reconstructed differs from the hypothetical in some trivial way, like by a scaling factor, or by a scaling factor that is a function of physical location or time, or a scaling factor that is a function of frequency. It isn't always simply a matter of a scaling factor, but it often is, and when it is, we often simply redefine the operator to include the scaling factor. Observe that there are two places for scaling functions (or filters), one in model space, the other in data space.

We could do better than the adjoint by iterative modeling methods (conjugate gradients) that are also described elsewhere. These methods generally demand that the adjoint be computed correctly. As a result, we'll be a little careful about adjoints in this book to compute them correctly even though this book does not require them to be exactly correct.

1.2.1 Dot product test

We define an adjoint when we write a program that computes one. In an abstract logical mathematical sense, however, every adjoint is defined by a **dot product test**. This abstract definition gives us no clues how to code our program. After we have finished coding, however, this abstract definition (which is actually a test) has considerable value to us.

Conceptually, the idea of matrix transposition is simply $a'_{ij} = a_{ji}$. In practice, however, we often encounter matrices far too large to fit in the memory of any computer. Sometimes it is also not obvious how to formulate the process at hand as a matrix multiplication. (Examples are differential equations and fast Fourier transforms.) What we find in practice is that an application and its adjoint amounts to two subroutines. The first subroutine amounts to the matrix multiplication $\mathbf{Fx}$. The adjoint subroutine computes $\mathbf{F}'\mathbf{y}$, where $\mathbf{F}'$ is the conjugate-transpose matrix. Most methods of solving inverse problems will fail if the programmer provides an inconsistent pair of subroutines for $\mathbf{F}$ and $\mathbf{F}'$. The dot product test described next is a simple test for verifying that the two subroutines really are adjoint to each other.

The matrix expression $\mathbf{y}'\mathbf{Fx}$ may be written with parentheses as either $(\mathbf{y}'\mathbf{F})\mathbf{x}$ or $\mathbf{y}'(\mathbf{Fx})$. Mathematicians call this the "associative" property. If you write matrix multiplication using summation symbols, you will notice that putting parentheses around matrices simply amounts to reordering the sequence of computations. But we soon get a very useful result. Programs for some linear operators are far from obvious, for example `causint()` on page 10. Now we build a useful test for it.

$$\mathbf{y}'(\mathbf{Fx}) \quad = \quad (\mathbf{y}'\mathbf{F})\mathbf{x} \tag{1.16}$$

$$\mathbf{y}'(\mathbf{Fx}) \quad = \quad (\mathbf{F}'\mathbf{y})'\mathbf{x} \tag{1.17}$$

For the dot-product test, load the vectors $\mathbf{x}$ and $\mathbf{y}$ with random numbers. Compute the vector $\tilde{\mathbf{y}} = \mathbf{Fx}$ using your program for $\mathbf{F}$, and compute $\tilde{\mathbf{x}} = \mathbf{F}'\mathbf{y}$ using your program for $\mathbf{F}'$. Inserting these into equation (1.17) gives you two scalars that should be equal.

$$\mathbf{y}'(\mathbf{Fx}) \quad = \quad \mathbf{y}'\tilde{\mathbf{y}} \quad = \quad \tilde{\mathbf{x}}'\mathbf{x} \quad = \quad (\mathbf{F}'\mathbf{y})'\mathbf{x} \tag{1.18}$$

The left and right sides of this equation will be computationally equal only if the program doing $\mathbf{F}'$ is indeed adjoint to the program doing $\mathbf{F}$ (unless the random numbers do something miraculous). Note that the vectors $\mathbf{x}$ and $\mathbf{y}$ are generally of different lengths.

Of course passing the dot product test does not prove that a computer code is correct, but if the test fails we know the code is incorrect. More information about adjoint operators, and much more information about inverse operators is found in my other books, Earth Soundings Analysis: Processing versus inversion (PVI) and Geophysical Estimation by Example (GEE).

Index