Chapter 1

Basic operators and adjoints

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 have 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

diffraction modeling	imaging by migration
hyperbola modeling	CDP stacking
ray tracing	**tomography**

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

When the adjoint operator is *not* an adequate approximation to the inverse, then you apply the techniques of fitting and optimization explained in chapter **??** which require iterative use of the modeling operator and its adjoint.

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). With complex-valued operators the transpose and complex conjugate go together and in Fourier analysis, taking the complex conjugate of $\exp(i\omega t)$ reverses the sense of time. Still assuming poetic license, I will 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, with linear interpolation 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. But nature determines in each application what is the best operator to use, whether to stop after the adjoint, to go the whole way to the inverse, or to stop part way.

We will see that computation of the **adjoint** is a straightforward adjunct to the computation itself, and the computed adjoint should be, and generally can be, exact (within machine precision). If the application's operator is computed in an approximate way, we will see that it is natural and best to compute the adjoint with adjoint approximations.

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 I postpone the 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 multiplication of a matrix $\mathbf{B}$ times a vector $\mathbf{x}$. The adjoint operation is $\tilde{x}_j = \sum_i b_{ij} y_i$. The operation adjoint to multiplying by a matrix is multiplying 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 matrix $\tilde{\mathbf{x}} = \mathbf{B}'\mathbf{y}$:

```
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()` (page 3), 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()` (page 3) for matrix multiply and its adjoint exhibits the style that we will use repeatedly. At last count there were about 25 such routines in this book alone.

```
# 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)
        else
                        x(ix) = x(ix) + bb(iy,ix) * y(iy)
```

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

```
        }}
return; end
```

1.1.1 Transient convolution

Matrix multiplication and transpose multiplication still fit easily in the same computational framework when the matrix has a special form, such as

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} = \begin{bmatrix} b_1 & 0 & 0 & 0 & 0 & 0 \\ b_2 & b_1 & 0 & 0 & 0 & 0 \\ b_3 & b_2 & b_1 & 0 & 0 & 0 \\ 0 & b_3 & b_2 & b_1 & 0 & 0 \\ 0 & 0 & b_3 & b_2 & b_1 & 0 \\ 0 & 0 & 0 & b_3 & b_2 & b_1 \\ 0 & 0 & 0 & 0 & b_3 & b_2 \\ 0 & 0 & 0 & 0 & 0 & b_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \mathbf{Bx} \tag{1.1}$$

Both are implemented by the pseudocode

```
do ix = 1, nx {
do ib = 1, nb {
        iy = ib + ix - 1
        if operator itself (convolution)
                y(iy) = y(iy) + b(ib) × x(ix)
        if adjoint (correlation)
                x(ix) = x(ix) + b(ib) × y(iy)
}}
```

Again, notice that the "bottom line" in the program is that x and y are simply interchanged.

Equation (1.1) could be rewritten as

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} = \begin{bmatrix} x_1 & 0 & 0 \\ x_2 & x_1 & 0 \\ x_3 & x_2 & x_1 \\ x_4 & x_3 & x_2 \\ x_5 & x_4 & x_3 \\ x_6 & x_5 & x_4 \\ 0 & x_6 & x_5 \\ 0 & 0 & x_6 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \mathbf{Xb} \tag{1.2}$$

In applications we can choose between $\mathbf{y} = \mathbf{Xb}$ and $\mathbf{y} = \mathbf{Bx}$. In one case the output $\mathbf{y}$ is dual to the filter $\mathbf{b}$, and in the other case the output $\mathbf{y}$ is dual to the input $\mathbf{x}$. Sometimes we must solve for $\mathbf{b}$ and sometimes for $\mathbf{x}$; so sometimes we use

equation (1.2) and sometimes (1.1). Such solutions begin from the adjoints. The adjoint of (1.1) is

$$\begin{bmatrix} \hat{x}_1 \\ \hat{x}_2 \\ \hat{x}_3 \\ \hat{x}_4 \\ \hat{x}_5 \\ \hat{x}_6 \end{bmatrix} = \begin{bmatrix} b_1 & b_2 & b_3 & 0 & 0 & 0 & 0 & 0 \\ 0 & b_1 & b_2 & b_3 & 0 & 0 & 0 & 0 \\ 0 & 0 & b_1 & b_2 & b_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & b_1 & b_2 & b_3 & 0 & 0 \\ 0 & 0 & 0 & 0 & b_1 & b_2 & b_3 & 0 \\ 0 & 0 & 0 & 0 & 0 & b_1 & b_2 & b_3 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} \tag{1.3}$$

The adjoint **crosscorrelate**s with the filter instead of convolving with it (because the filter is backwards). Another notable difference is that here the filter does not run off the ends of its input data and therefore does not assume data vanishes beyond the interval where it is given.

The adjoint of (1.2) crosscorrelates a fixed portion of filter input across a variable portion of filter output.

$$\begin{bmatrix} \hat{b}_1 \\ \hat{b}_2 \\ \hat{b}_3 \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & 0 & 0 \\ 0 & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & 0 \\ 0 & 0 & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} \tag{1.4}$$

Subroutine `tcai1()` (page 5) is used for $\mathbf{y} = \mathbf{B}\mathbf{x}$ and subroutine `tcaf1()` (page 5) is used for $\mathbf{y} = \mathbf{X}\mathbf{b}$.

```
# Transient Convolution Adjoint is the Input 1-D.           real yy(m1+n1)
#
subroutine tcai1( adj,add, bb,m,  xx,n,  yy )
integer           adj,add,  b,m,   x,n,       y
real                       bb(m), xx(n), yy( m+n)
call adjnull(     adj,add,        xx,n,  yy,(m+n))
do b=1,m {
do x=1,n {                 y = x + b - 1
        if( adj == 0 )
                       yy(y) = yy(y) + xx(x) * bb(b)
        else
                       xx(x) = xx(x) + yy(y) * bb(b)
        }}
return; end


# Transient Convolution, Adjoint is the Filter  1-dimensional
#
subroutine tcaf1( adj, add,  xx,nx,  bb,nb,  yy  )
integer           adj, add,   x,nx,   b,nb,   y
real                         xx(nx), bb(nb), yy( nx+nb)
call adjnull(     adj, add,          bb,nb,  yy, nx+nb)
```

```
do b= 1, nb {
do x= 1, nx {                          y = x + b - 1
        if( adj == 0 )
                        yy(y) = yy(y) + bb(b) * xx(x)
        else
                        bb(b) = bb(b) + yy(y) * xx(x)
        } }
return; end
```

The length of the output of transient convolution is one less than the length of the filter plus the length of the input. Notice, however that in these subroutines, the input length plus the filter length equals the output length because these routines erase the extra memory cell, so you must be sure you have allocated that memory cell. I erase the extra cell to reduce clutter in the many 2-D and 3-D subroutines that build upon transient convolution. It is important that the extra memory cell be erased and not simply ignored, since otherwise it may contain an impossible value which would break many subroutines such as plot routines and inversion routines.

1.1.2 Internal convolution

Convolution is the computational equivalent of differential equations. Its applications are vast and end effects are important. Another design for a convolution program is that zero data not be assumed beyond the interval that the data is given. This is important where crosscorrelations change with time and are handled as constant in short time windows. Care must be taken that zero signal values not be presumed off the ends of the short time windows otherwise the many ends of the many short segments can overwhelm the results.

In the sets (1.1) and (1.2), the top two equations explicitly assume the input data vanishes before the interval on which it is given, likewise at the bottom. Abandoning in (1.2) the top two and bottom two equations (which are making these assumptions) we get:

$$\begin{bmatrix} y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} = \begin{bmatrix} x_3 & x_2 & x_1 \\ x_4 & x_3 & x_2 \\ x_5 & x_4 & x_3 \\ x_6 & x_5 & x_4 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{1.5}$$

The adjoint is

$$\begin{bmatrix} \hat{b}_1 \\ \hat{b}_2 \\ \hat{b}_3 \end{bmatrix} = \begin{bmatrix} x_3 & x_4 & x_5 & x_6 \\ x_2 & x_3 & x_4 & x_5 \\ x_1 & x_2 & x_3 & x_4 \end{bmatrix} \begin{bmatrix} y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} \tag{1.6}$$

The difference between (1.6) and (1.4) is that here the adjoint crosscorrelates a fixed portion of output across a variable portion of *input* whereas with (1.4) the adjoint crosscorrelates a fixed portion of input across a variable portion of *output*.

In programs we typically allocate equal space for input and output. Since the output is shorter than the input, it could slide around in its allocated space, so its location is specified by an additional parameter called its `lag`. The statement `x=y-b+lag` in subroutine `icaf1()` (page 7) says that the output time `y` aligns with the input time `x` for the filter point `b=lag`.

```
# Internal Convolution, Adjoint is the Filter. 1-dimensional
#
subroutine icaf1( adj, add, lag, xx,n,  bb,nb,  yy  )
integer           adj, add, lag,  x,n,   b,nb,   y
real                               xx(n), bb(nb), yy(n)
call adjnull(     adj, add,              bb,nb,  yy,n )
do b= 1, nb {
        do y= 1+nb-lag, n-lag+1 {    x= y - b + lag
                if( adj == 0 )
                                  yy(y) = yy(y) + bb(b) * xx(x)
                else
                                  bb(b) = bb(b) + yy(y) * xx(x)
                }
        }
return; end
```

The ugly looking line `do y= 1+nb-lag, n-lag+1` lays the output down in its specified location. For `lag=1`, the subroutine implements not equation (1.5) but (1.7)

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ x_3 & x_2 & x_1 \\ x_4 & x_3 & x_2 \\ x_5 & x_4 & x_3 \\ x_6 & x_5 & x_4 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{1.7}$$

The **end effect**s of the convolution programs are summarized in Figure 1.1.

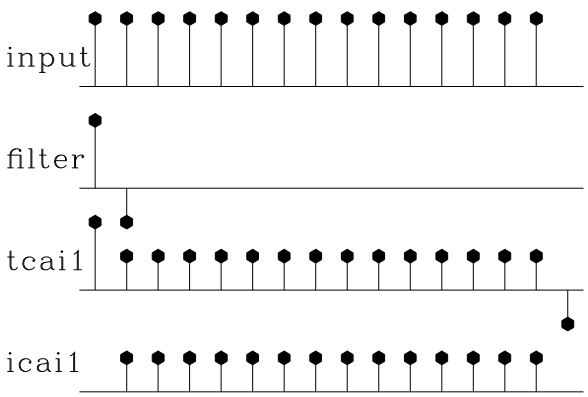

Figure 1.1: Example of convolution end effects. From top to bottom: (1) input; (2) filter; (3) output of `tcai1()`, (4) output of `icaf1()` also with (`lag=1`). ajt-conv [NR]

1.1.3 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 **x**,

and then make a longer vector $\mathbf{y}$ by adding some zeros at the end of $\mathbf{x}$. This zero padding can be regarded as the matrix multiplication

$$\mathbf{y} \quad = \quad \begin{bmatrix} \mathbf{I} \\ \mathbf{0} \end{bmatrix} \mathbf{x} \tag{1.8}$$

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:

$$\tilde{\mathbf{x}} \quad = \quad \begin{bmatrix} \mathbf{I} & \mathbf{0} \end{bmatrix} \mathbf{y} \tag{1.9}$$

So the transpose operation to zero padding data is simply *truncating* the data back to its original length. Subroutine `zpad1()` (page 8) 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
```

1.1.4 Product of operators

We will look into details of Fourier transformation elsewhere. Here we use it as an example of any operator containing complex numbers. For now, we can think of Fourier transform as 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 particularly 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

$$\text{Program} \quad = \quad \mathbf{F} \begin{bmatrix} \mathbf{I} \\ \mathbf{0} \end{bmatrix} \tag{1.10}$$

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

$$\text{Program}' \quad = \quad \begin{bmatrix} \mathbf{I} & \mathbf{0} \end{bmatrix} \mathbf{F}' \tag{1.11}$$

Thus the adjoint subroutine *truncates* the data *after* the inverse Fourier transform.

1.1.5 Adjoint derivative

Given a sampled signal, its time **derivative** can be estimated by convolution with the filter $(1, -1)/\Delta t$. This example arises so frequently that I display the matrix multiply below:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} = \begin{bmatrix} -1 & 1 & . & . & . & . \\ . & -1 & 1 & . & . & . \\ . & . & -1 & 1 & . & . \\ . & . & . & -1 & 1 & . \\ . & . & . & . & -1 & 1 \\ . & . & . & . & . & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} \tag{1.12}$$

Technically the output should be `n-1` points long, but I appended a zero row, a small loss of logical purity, so the size of the output vector will match that of the input, 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 the adjoint of "no boundary condition" is "specified value" boundary conditions.

Subroutine `ruffen1()` implements equation (1.12) and its adjoint.

```
subroutine ruffen1( 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
```

In subroutine `ruffen1()` (page 9) the adjoint code looks different from what we have previously seen. Where the forward operator has extra terms, the adjoint has extra assignments. To understand why this is so, look back to the matrix multiply subroutine `matmult()` (page 3). Suppose one of the `do` loops ranges only over *two* terms instead of being a loop to N terms. Then revise the code to write the two terms explicitly instead of using a `do` loop. Immediately you see that *the adjoint to extra terms is extra assignment statements.*

1.1.6 Nearest-neighbor interpolation and binning

In describing physical processes we often 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 mostly use positive counting variables. If you use counting variables that are both positive and negative, then to get the nearest integer you should do your rounding with the Fortran function `NINT()`.

Above is how we handle coordinates, next is the ordinates, the function values. Nearest-neighbor interpolation is an operator. There is both a forward operator and its adjoint. Normally the model is values given on a uniform mesh and the data is pairs of numbers, ordinates at coordinates, sprinkled around in the continuum (although sometimes the data is uniformly spaced and the model is not).

In the both the forward and the adjoint operation, we examine each data coordinate and find the nearest mesh point (the bin). For the forward operator, we add the value of the bin to that of the data. The adjoint is the reverse, we add the value of the data to that of the bin. This is done in subroutine `bin1()` (page 10).

```
# bin spray in 1-D, uniform model mesh to data ordinates and values.
#
subroutine bin1( adj, add, o1,d1,ordinate,     mm,m1,  dd, nd)
integer          adj, add,                id,  im,m1,      nd
real                        o1,d1,ordinate(nd), mm(m1), dd( nd)
call adjnull(    adj, add,                     mm,m1,  dd, nd)
do id= 1, nd {
        im = 1.5 + (ordinate(id)-o1)/d1
        if( 1<=im & im<=m1)
                if( adj == 0)
                        dd(id) = dd(id) + mm(im)
                else
```

```
                        mm(im) = mm(im) + dd(id)
        }
return; end
```

The most typical application requires an additional step, inversion. In the inversion application there is a different count of data values in each bin. In inversion, the adjoint operation is followed by dividing the bin value by the number of points in the bin. This is called binning. There is trouble if no data points happen to land in a bin, and we postpone that problem for the next chapter where we attack a family of such problems. To find the number of data points in a bin we can simply apply the adjoint of `bin1()` (page 10) to pseudo data of all ones. To capture this idea in an equation, let $\mathbf{B}$ denote the linear operator of spraying the bin value to the data values in the bin. The inverse operation, summing the data values in the bin and dividing by the number in the bin is represented by

$$\mathbf{m} \quad = \quad \mathbf{diag}(\mathbf{B}'\mathbf{1})^{-1}\mathbf{B}'\mathbf{d} \tag{1.13}$$

1.1.7 Linear interpolation

The **linear interpolation** operator is much like the binning operator but a little fancier. In the forward operation, we take each data coordinate and see which two model mesh points bracket it. Then we pick up the two model values and weight each of them in proportion to their nearness to the data coordinate before adding 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 to distribute 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 on a model point near the middle. Then for a model space 6 points long the operator might be:

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix} \quad \approx \quad \begin{bmatrix} .8 & .2 & . & . & . & . \\ . & . & 1 & . & . & . \\ . & . & . & . & .5 & .5 \end{bmatrix} \begin{bmatrix} m_0 \\ m_1 \\ m_2 \\ m_3 \\ m_4 \\ m_5 \end{bmatrix} \tag{1.14}$$

The two weights in each row sum to unity. 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 the matrix is crudely a diagonal matrix like equation (1.14). If the data coordinates covered the model space uniformly the adjoint would roughly be the inverse, but for a typical collection of data coordinates the adjoint could be far from the inverse since data values might pile up in some places while gaps remained elsewhere.

Subroutine `lint1()` (page 11) does linear interpolation and its adjoint. In chapters ?? and ?? we build inverse operators.

```
# Linear interpolation 1-D, uniform model mesh to data ordinates and values.
#
subroutine lint1( adj, add, o1,d1,ordinate,     mm,m1,  dd, nd)
integer i, im,    adj, add,                 id,     m1,      nd
real    f, fx,gx,           o1,d1,ordinate(nd), mm(m1), dd( nd)
call adjnull(     adj, add,                     mm,m1,  dd, nd)
do id= 1, nd {
        f = (ordinate(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
```

1.1.8 Causal and leaky integration

Causal integration is defined as

$$y(t) \quad = \quad \int_{-\infty}^{t} x(t)\, dt \tag{1.15}$$

Leaky integration is defined as

$$y(t) \quad = \quad \int_{0}^{\infty} x(t-\tau)\, e^{-\alpha\tau}\, d\tau \tag{1.16}$$

As $\alpha \rightarrow 0$, leaky integration becomes causal integration. The word "leaky" comes from electrical circuit theory where the voltage on a capacitor would be the integral of the current if the capacitor did not leak electrons.

Sampling the time axis gives a matrix equation which we should call causal summation, but we often call it causal integration. Equation (1.17) represents causal integration for $\rho = 1$ and leaky integration for $0 < \rho < 1$.

$$\mathbf{y} \quad = \quad \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} \quad = \quad \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \rho & 1 & 0 & 0 & 0 & 0 & 0 \\ \rho^2 & \rho & 1 & 0 & 0 & 0 & 0 \\ \rho^3 & \rho^2 & \rho & 1 & 0 & 0 & 0 \\ \rho^4 & \rho^3 & \rho^2 & \rho & 1 & 0 & 0 \\ \rho^5 & \rho^4 & \rho^3 & \rho^2 & \rho & 1 & 0 \\ \rho^6 & \rho^5 & \rho^4 & \rho^3 & \rho^2 & \rho & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} \quad = \quad \mathbf{Cx} \tag{1.17}$$

(The discrete world is related to the continuous by $\rho = e^{-\alpha\Delta\tau}$ and in some applications, the diagonal is 1/2 instead of 1.) 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 $y_5 = \rho y_4 + x_5$. Thus equation (1.17) is more efficiently thought of as the recursion

$$y_t \quad = \quad \rho\, y_{t-1} + x_t \qquad t \text{ increasing} \tag{1.18}$$

(which for $\rho = 1$ may also be regarded as a numerical representation of the **differential equation** $dy/dt + \alpha y = x(t)$.)

When it comes time to think about the adjoint, however, it is easier to think of equation (1.17) than of (1.18). Let the matrix of equation (1.17) 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

$$\tilde{x}_{t-1} \quad = \quad \rho\tilde{x}_t + y_{t-1} \qquad t \text{ decreasing} \tag{1.19}$$

Thus the adjoint of causal integration is anticausal integration.

A subroutine to do these jobs is `leakint()` (page 13). 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. Figure 1.2 illustrates the program for $\rho = 1$.

```
# leaky integration
#
subroutine leakint( adj, add,      rho, xx,n,  yy   )
integer i,       n, adj, add; real rho, 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)   = rho*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) = rho*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.

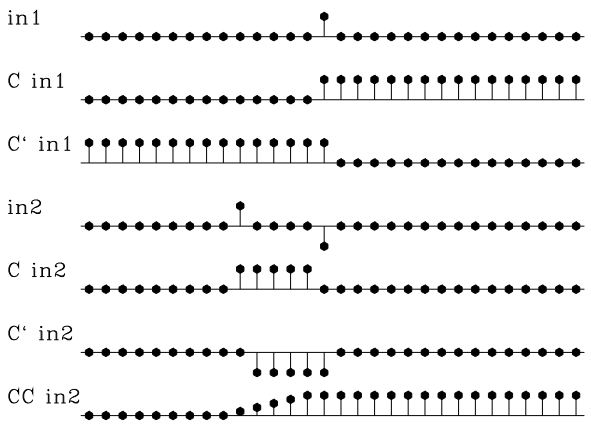

Figure 1.2: `in1` is an input pulse. `C in1` is its causal integral. `C' in1` is the anticausal integral of the pulse. `in2` is a separated doublet. Its causal integration is a box and its anticausal integration is the negative. `CC in2` is the double causal integral of `in2`. How can an equilateral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

EXERCISES:

1 Consider the matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & \rho & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.20}$$

and others like it with ρ in other locations. Show what combination of these matrices will represent the leaky integration matrix in equation (1.17). What is the adjoint?

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

3 Notice that the triangle waveform is not time aligned with the input `in2`. Force time alignment with the operator $\mathbf{C}'\mathbf{C}$ or $\mathbf{C}\mathbf{C}'$.

4 Modify `leakint()` (page 13) by changing the diagonal to contain 1/2 instead of 1. Notice how time alignment changes in Figure 1.2.

1.1.9 Spray and sum : scatter and gather

Perhaps the most common operation is summing many values to get one value. Its adjoint operation takes a single input value and throws it out to a space of many values. The summation operator is is a row vector of ones. Its adjoint is a column vector of ones. This operator is almost too easy to bother showing a subroutine but it is more interesting in three dimensions where we could be summing or spraying on any of three subscripts, or even summing on some and spraying on others. In subroutine

spraysum() (page 15) we take both input and output to be three-dimensional arrays. Externally, however, either could be a scalar, vector, plane, or cube. For example the internal array xx(n1,1,n3) could be externally the matrix map(n1,n3). Giving to subroutine spraysum() (page 15) the input dimensions and output dimensions stated below implies the operations stated alongside.

(n1,n2,n3)	(1,1,1)	Sum a cube into a value.
(1,1,1)	(n1,n2,n3)	Spray a value into a cube.
(n1,1,1)	(n1,n2,1)	Spray a column into a matrix.
(1,n2,1)	(n1,n2,1)	Spray a row into a matrix.
(n1,n2,1)	(n1,n2,n3)	Spray a plane into a cube.
(n1,n2,1)	(n1,1,1)	Sum rows of a matrix into a column.
(n1,n2,1)	(1,n2,1)	Sum columns of a matrix into a row.
(n1,n2,n3)	(n1,n2,n3)	Copy and add the whole cube.

If an axis is not of unit length on either input or output, then both lengths must be the same, otherwise there is an error. Normally, after (possibly) erasing the output, we simply loop over all points on each axis, adding the input to the output. This implements either a copy or an add, depending on the add parameter. It is either a spray, a sum, or a copy depending on the specified axis lengths.

```
# Spray or sum over 1, 2, and/or 3-axis.
#
subroutine spraysum( adj, add, xx,n1,n2,n3,  yy,m1,m2,m3 )
integer i1,i2,i3,    adj, add,    n1,n2,n3,     m1,m2,m3,  x1,x2,x3, y1,y2,y3
real                            xx(n1,n2,n3), yy(m1,m2,m3)
call adjnull(        adj, add, xx,n1*n2*n3,  yy,m1*m2*m3)
        if( n1 != 1  &  m1 != 1  &  n1 != m1)   call erexit('spraysum: n1,m1')
        if( n2 != 1  &  m2 != 1  &  n2 != m2)   call erexit('spraysum: n2,m2')
        if( n3 != 1  &  m3 != 1  &  n3 != m3)   call erexit('spraysum: n3,m3')
do i3= 1, max0(n3,m3) {   x3= min0(i3,n3);   y3= min0(i3,m3)
do i2= 1, max0(n2,m2) {   x2= min0(i2,n2);   y2= min0(i2,m2)
do i1= 1, max0(n1,m1) {   x1= min0(i1,n1);   y1= min0(i1,m1)
        if( adj == 0 )
                yy(y1,y2,y3) = yy(y1,y2,y3) + xx(x1,x2,x3)
        else
                xx(x1,x2,x3) = xx(x1,x2,x3) + yy(y1,y2,y3)
        }}}
return; end
```

1.1.10 Nearest-neighbor normal moveout (NMO)

Normal-moveout correction (**NMO**) is a geometrical correction of reflection data that stretches the time axis so that data recorded at nonzero separation x_0 of shot and receiver, after stretching, appears to be at $x_0 = 0$. NMO correction is roughly like time-to-depth conversion with the equation $v^2t^2 = z^2 + x_0^2$. After the data at x_0 is stretched from t to z, it should look like stretched data from any other x (assuming plane horizontal reflectors, etc.). In practice, z is not used; rather, **traveltime depth** τ is used, where $\tau = z/v$; so $t^2 = \tau^2 + x_0^2/v^2$. (Because of the limited alphabet of programming languages, I often use the keystroke z to denote τ.)

Typically, many receivers record every shot. Each seismogram can be transformed by NMO and the results all added. This is called "**stack**ing" or "**NMO stack**ing." The adjoint to this operation is to begin from a model that is identical to the near-offset trace and spray this trace to all offsets. From a matrix viewpoint, stacking is like a *row* vector of normal moveout operators and modeling is like a *column*.

A subroutine that does reverse moveout is `hypotenuse()` (page 16). Given a zero-offset trace, it makes one at non-zero offset. The adjoint does the usual normal moveout correction.

```
subroutine hypotenuse( adj, add, xs, t0, dt, nt, zz,     tt )
integer                adj, add,             nt,                          it, iz
real                                    t0, dt,     zz(nt), tt(nt), t, xs, zsquared
call adjnull(          adj, add,                    zz,nt,  tt,nt)
do it= 1, nt {  t = t0 + dt*(it-1)
        zsquared =  t * t - xs * xs
        if ( zsquared >= 0.) {  iz = 1.5 + (sqrt( zsquared) - t0) /dt
                if ( iz > 0 ) { if( adj == 0 )
                                        tt(it) = tt(it) + zz(iz)
                                else
                                        zz(iz) = zz(iz) + tt(it)
                                }
                }
        }
return; end
```

A companion routine `imospray()` (page 16) loops over offsets making a trace for each. The adjoint of `imospray()` is the industrial process of moveout and stack. My 1992 textbook (PVI) illustrates many additional features of Normal Moveout.

```
# inverse moveout and spray into a gather.
#
subroutine imospray( adj, add, slow, x0,dx, t0,dt, nx,nt, stack, gather)
integer ix,          adj, add,                       nx,nt
real xs,                          slow, x0,dx, t0,dt,    stack(nt), gather( nt,nx)
call adjnull(        adj, add,                           stack,nt,  gather, nt*nx)
do ix= 1, nx {
        xs = (x0 + dx * (ix-1)) * slow
        call hypotenuse( adj, 1, xs, t0, dt, nt,      stack,     gather(1,ix))
        }
return; end
```

1.2 ADJOINT DEFINED: DOT-PRODUCT TEST

Having seen many examples of **space**s, operators, and adjoints, it is time for more formal definitions because abstraction helps us push these concepts to their limits.

1.2.1 Definition of a vector space

An operator transforms a space to another space. Examples of spaces are model space $\mathbf{m}$ and data space $\mathbf{d}$. We think of these spaces as vectors whose components are packed with numbers, either real or complex numbers. The important practical concept is that this packing includes not only one-dimensional spaces like signals,

two-dimensional spaces like images, 3-D movie cubes, zero-dimensional spaces like a data mean, etc, but spaces can be sets of all the above. A space that is a set of three cubes is the earth's magnetic field which has three components and each component is a function of a three-dimensional space. (The 3-D *physical space* we live in is not the abstract *vector space* of models and data so abundant in this book. Here the word "space" without an adjective means the vector space.)

A more heterogeneous example of a vector space is data tracks. A depth sounding survey of a lake can make a vector space that is a collection of tracks, a vector of vectors (each vector having a different number of components, since lakes are not square). This vector space of depths along tracks in a lake contains the depth values only. The (x, y)-coordinate information locating each measured depth value is (normally) something outside the vector space. A data space could also be a collection of echo soundings, waveforms recorded along tracks.

I briefly recapitulate information about vector spaces found in elementary books: Let α be any scalar. Then if $\mathbf{d}_1$ is a vector space and $\mathbf{d}_2$ is conformable with it, then other spaces are $\alpha\mathbf{d}_1$ and $\mathbf{d}_1 + \mathbf{d}_2$. Spaces have a size measure which is a positive value called a norm. The norm is usually defined to be the dot product (also called the L_2 norm), say $\mathbf{d} \cdot \mathbf{d}$. For complex data it is $\bar{\mathbf{d}} \cdot \mathbf{d}$ where $\bar{\mathbf{d}}$ is the complex conjugate of $\mathbf{d}$. In theoretical work the "length of a vector" means the vector's norm. In computational work the "length of a vector" means the number of components in the vector.

Norms generally include a weighting function. In physics the norm generally measures a conserved quantity like energy or momentum so, for example, a weighting function for magnetic flux is permittivity. In data analysis the proper choice of the weighting function is a practical statistical issue, discussed repeatedly throughout this book. The algebraic view of a weighting function is that it is a diagonal matrix with positive values $w(i) \geq 0$ spread along the diagonal, and it is denoted $\mathbf{W} = \mathbf{diag}[w(i)]$. With this weighting function the L_2 norm of a data space is denoted $\mathbf{d}'\mathbf{W}\mathbf{d}$. Standard notation for norms uses a double absolute value where $||\mathbf{d}|| = \mathbf{d}'\mathbf{W}\mathbf{d}$. A central concept with norms is the triangle inequality, $||\mathbf{d}_1 + \mathbf{d}_2|| \leq ||\mathbf{d}_1|| + ||\mathbf{d}_2||$ whose proof you might recall (or reproduce using dot products).

1.2.2 Dot-product test

There is a huge gap between the conception of an idea and putting it into practice. During development, things fail far more often than not. Often, when something fails, many tests are needed to track down the cause of failure. Maybe the cause cannot even be found. More insidiously, failure may be below the threshold of detection and poor performance suffered for years. I find the **dot-product test** to be an extremely valuable checkpoint.

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 trans-

forms.) What we find in practice is that an application and its adjoint amounts to two subroutines. The first subroutine amounts to the matrix multiplication $\mathbf{Ax}$. The adjoint subroutine computes $\mathbf{A}'\mathbf{y}$, where $\mathbf{A}'$ is the **conjugate-transpose** matrix. In later chapters we will be solving huge sets of simultaneous equations. Then both subroutines are required. We are doomed from the start if the practitioner provides an inconsistent pair of subroutines. The dot product test is a simple test for verifying that the two subroutines are adjoint to each other.

The associative property of linear algebra says that we do not need parentheses in a vector-matrix-vector product like $\mathbf{y}'\mathbf{Ax}$ because we get the same result no matter where we put the parentheses. They serve only to determine the sequence of computation. Thus,

$$\mathbf{y}'(\mathbf{Ax}) = (\mathbf{y}'\mathbf{A})\mathbf{x} \tag{1.21}$$

$$\mathbf{y}'(\mathbf{Ax}) = (\mathbf{A}'\mathbf{y})'\mathbf{x} \tag{1.22}$$

(In general, the matrix is not square.) For the dot-product test, load the vectors $\mathbf{x}$ and $\mathbf{y}$ with random numbers. Compute the vector $\tilde{\mathbf{y}} = \mathbf{Ax}$ using your program for $\mathbf{A}$, and compute $\tilde{\mathbf{x}} = \mathbf{A}'\mathbf{y}$ using your program for $\mathbf{A}'$. Inserting these into equation (1.22) gives you two scalars that should be equal.

$$\mathbf{y}'(\mathbf{Ax}) = \mathbf{y}'\tilde{\mathbf{y}} = \tilde{\mathbf{x}}'\mathbf{x} = (\mathbf{A}'\mathbf{y})'\mathbf{x} \tag{1.23}$$

The left and right sides of this equation will be computationally equal only if the program doing $\mathbf{A}'$ is indeed adjoint to the program doing $\mathbf{A}$ (unless the random numbers do something miraculous).

I tested (1.23) on many operators and was surprised and delighted to find that it is often satisfied to an accuracy near the computing precision. I do not doubt that larger rounding errors could occur, but so far, every time I encountered a relative discrepancy of 10^{-5} or more, I was later able to uncover a conceptual or programming error. Naturally, when I do dot-product tests, I scale the implied matrix to a small dimension in order to speed things along, and to be sure that boundaries are not overwhelmed by the much larger interior.

Do not be alarmed if the operator you have defined has **truncation** errors. Such errors in the definition of the original operator should be identically matched by truncation errors in the adjoint operator. If your code passes the **dot-product test**, then you really have coded the adjoint operator. In that case, you can take advantage of the standard methods of mathematics to obtain inverse operators.

We can speak of a continuous function $f(t)$ or a discrete one f_t. For continuous functions we use integration, and for discrete ones we use summation. In formal mathematics the dot-product test *defines* the adjoint operator, except that the summation in the dot product may need to be changed to an integral. The input or the output or both can be given either on a continuum or in a discrete domain. So the dot-product test $\mathbf{y}'\tilde{\mathbf{y}} = \tilde{\mathbf{x}}'\mathbf{x}$ could have an integration on one side of the equal sign and a summation on the other. Linear-operator theory is rich with concepts, but I will not develop it here.

1.2.3 The word "adjoint"

In mathematics the word "**adjoint**" has three meanings. One of them, the so-called **Hilbert adjoint**, is the one generally found in Physics and Engineering and it is the one used in this book. In Linear Algebra is a different matrix, called the **adjugate** matrix. It is a matrix whose elements are signed cofactors (minor determinants). For invertible matrices, this matrix is the determinant times the inverse matrix. It is computable without ever using division, so potentially the adjugate can be useful in applications where an inverse matrix cannot. Unfortunately, the adjugate matrix is sometimes called the adjoint matrix particularly in the older literature. Because of the confusion of multiple meanings of the word adjoint, in the first printing of PVI I avoided the use of the word, substituting the definition, "conjugate transpose". Unfortunately this was often abbreviated to "conjugate" which caused even more confusion.

1.2.4 Matrix versus operator

Here is a short summary of where we have been and where we are going: Start from the class of linear operators, add subscripts and you get matrices. Examples of operators without subscripts are subroutines that solve differential equations and subroutines that do fast Fourier transform. What people call "sparse matrices" are often not really matrices but operators because they are not defined by data structures but by subroutines that apply them to a vector. With sparse matrices you can do $\mathbf{A}(\mathbf{B}(\mathbf{C}\mathbf{x}))$ but not $(\mathbf{A}\mathbf{B}\mathbf{C})\mathbf{x}$.

Although a linear operator does not have defined subscripts, you can determine what would be the operator value at any subscript by applying the operator to an impulse function getting a matrix column. The adjoint operator is one from which we can extract the transpose matrix. For large spaces this is unwieldy so we test the validity of adjoints by probing them with random vectors, say $\mathbf{x}$ and $\mathbf{y}$, to see whether $\mathbf{y}'(\mathbf{A}\mathbf{x}) = (\mathbf{A}'\mathbf{y})'\mathbf{x}$. Mathematicians define adjoints by this test except that instead of using random vectors, they say "for all functions" which includes the continuum. This defining test makes adjoints look mysterious. Careful inspection of operator adjoints, however, generally reveals that they are built up from simple matrices. Given adjoints $\mathbf{A}'$, $\mathbf{B}'$, and $\mathbf{C}'$, the adjoint of $\mathbf{A}\mathbf{B}\mathbf{C}$ is $\mathbf{C}'\mathbf{B}'\mathbf{A}'$. Fourier transforms and linear differential equation solvers are chains of matrices so their adjoints can be assembled by applying the adjoint components in reverse order. The other way we often see complicated operators being built from simple ones and from matrices is by putting operators into matrices as components, typically a 1×2 or 2×1 matrix containing two operators. An example of the adjoint of a two component column operator is

$$\begin{bmatrix} \mathbf{A} \\ \mathbf{B} \end{bmatrix}' \quad = \quad \begin{bmatrix} \mathbf{A}' & \mathbf{B}' \end{bmatrix} \tag{1.24}$$

Although in practice an operator might be built from matrices, fundamentally,

a matrix is a data structure whereas an operator is a procedure. A matrix is an operator if its subscripts are hidden but you can apply it to a space, getting another space.

As matrices have inverses, so do linear operators. You don't need subscripts to find an inverse. The conjugate-gradient method is an attractive method of finding them. It merely applies $\mathbf{A}$ and $\mathbf{A}'$ and uses inner products to find coefficients of a polynomial in $\mathbf{A}\mathbf{A}'$ that represents the inverse operator.

Whenever we encounter a **positive-definite** matrix we should recognize its likely origin in a nonsymmetric matrix $\mathbf{A}$ times its adjoint. Natural scientists often work on solving simultaneous equations but fail to realize they should return to the origin of the equations which is usually a regression, i.e., data approximately is an operator onto a model, i.e., $\mathbf{d} \approx \mathbf{d}_0 + \mathbf{A}(\mathbf{m} - \mathbf{m}_0)$ where the operator $\mathbf{A}$ is a partial derivative matrix (and there are potential underlying nonlinearities). This begins another story with a new ingredient, weighting functions and "spectral leveling," a story which soon balloons into inverse covariance matrices whose representation and construction is a fascinating unfinished chapter in natural science.

1.2.5 Inverse operator

A common practical task is to fit a vector of observed data $\mathbf{d}_{\text{obs}}$ to some theoretical data $\mathbf{d}_{\text{theor}}$ by adjusting components in a vector of model parameters $\mathbf{m}$.

$$\mathbf{d}_{\text{obs}} \quad \approx \quad \mathbf{d}_{\text{theor}} \quad = \quad \mathbf{A}\mathbf{m} \tag{1.25}$$

A huge volume of theoretical literature establishes two estimates for the model, $\hat{\mathbf{m}}_1$ and $\hat{\mathbf{m}}_2$ where

$$\hat{\mathbf{m}}_1 \quad = \quad (\mathbf{A}'\mathbf{A})^{-1}\mathbf{A}'\mathbf{d} \tag{1.26}$$

$$\hat{\mathbf{m}}_2 \quad = \quad \mathbf{A}'(\mathbf{A}\mathbf{A}')^{-1}\mathbf{d} \tag{1.27}$$

Some reasons the literature is huge are the many questions about the existance, quality, and cost of the inverse operators. Before I summarize that, let us quickly see why these two solutions are reasonable. Inserting equation (1.25) into equation (1.26) and equation (1.27) into equation (1.25) we get the reasonable statements:

$$\hat{\mathbf{m}}_1 \quad = \quad (\mathbf{A}'\mathbf{A})^{-1}(\mathbf{A}'\mathbf{A})\mathbf{m} \quad = \quad \mathbf{m} \tag{1.28}$$

$$\hat{\mathbf{d}}_{\text{theor}} \quad = \quad (\mathbf{A}\mathbf{A}')(\mathbf{A}\mathbf{A}')^{-1}\mathbf{d} \quad = \quad \mathbf{d} \tag{1.29}$$

Equation (1.28) says that the estimate $\hat{\mathbf{m}}_1$ gives the correct model $\mathbf{m}$ if you start from the theoretical data. Equation (1.29) says that the model estimate $\hat{\mathbf{m}}_2$ gives the theoretical data if we derive $\mathbf{m}$ from the theoretical data. Both of these statements are delightful. Now let us return to the problem of the inverse matrices.

Strictly speaking, a rectangular matrix does not have an inverse. Surprising things often happen, but commonly, when $\mathbf{A}$ is a tall matrix (more data values than model

values) then the matrix for finding $\hat{\mathbf{m}}_1$ is invertable while that for finding $\hat{\mathbf{m}}_2$ is not, and when the number of data values is less than the number of model values it is the other way around. In many applications neither $\mathbf{A}'\mathbf{A}$ nor $\mathbf{A}\mathbf{A}'$ is invertible. This difficulty is solved by "damping" as we will see in later chapters. The point to notice in this chapter on adjoints is that in any application where $\mathbf{A}\mathbf{A}'$ or $\mathbf{A}'\mathbf{A}$ equals $\mathbf{I}$ (unitary operator) that the adjoint operator $\mathbf{A}'$ is the inverse $\mathbf{A}^{-1}$ by either equation (1.26) or (1.27). Theoreticians like to study inverse problems where $\mathbf{m}$ is drawn from the field of continuous functions. This is like the vector $\mathbf{m}$ having infinitely many components. Such problems are hopelessly intractable unless we find, or assume, that the operator $\mathbf{A}\mathbf{A}'$ is an identity or diagonal matrix.

In practice, theoretical considerations may have little bearing on how we proceed. Current computation power limits matrix inversion programs to about 10^3 variables. This book specializes in big problems, those with more than about 10^3 variables, but the methods we learn are also excellant for smaller problems.

EXERCISES:

1 Suppose a linear operator $\mathbf{A}$ has its input in the discrete domain and its output in the continuum. How does the operator resemble a matrix? Describe the operator $\mathbf{A}'$ which has its output in the discrete domain and its input in the continuum. To which do you apply the words "scales and adds some functions," and to which do you apply the words "does a bunch of integrals"? What are the integrands?

2 Generally in a computer we represent a one-dimensional function by a series of values (ordinates at discrete, uniformly spaced coordinates). Suppose instead we wish to interlace function values with function derivatives, say $y_0, y_1', y_2, y_3', y_4, y_5', \cdots$ where prime denotes derivative. I have found that a simple transformation with this flavor going from values to interlaced values and derivatives is:

$$\mathbf{L} \quad = \quad \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 & . & . & . & . & . & . \\ . & 1 & -1 & . & . & . & . & . \\ . & . & 1 & 1 & . & . & . & . \\ . & . & . & 1 & -1 & . & . & . \\ . & . & . & . & 1 & 1 & . & . \\ . & . & . & . & . & 1 & -1 & . \\ . & . & . & . & . & . & 1 & 1 \end{bmatrix} \tag{1.30}$$

Although I know an exact inverse operator to $\mathbf{L}$ (based on $\mathbf{L}\mathbf{L}'$ being tridiagonal), I have chosen the scale factor so that the tridiagonal matrix $\mathbf{L}'\mathbf{L}$ is an identity matrix approximation for low frequencies since $\mathbf{L}'\mathbf{L}\mathbf{1} = \mathbf{1}$ (except for end effects) where $\mathbf{1}$ is a vector of 1's, i.e., a low frequency signal.

1. Write a subroutine for $\mathbf{L}$ and $\mathbf{L}'$. Do the dot-product test.
2. Given a signal represented in "signal-derivative-interlace" form, write a routine for filtering this signal where the adjoint is the filter, and another where the adjoint is the input.

3. Evaluate $\mathbf{L}'\mathbf{L}$ and verify that $\mathbf{L}'\mathbf{L}\mathbf{1} = \mathbf{1}$ (except for end effects).
4. If you don't like the inverse approximation notice that we could represent the derivative by the bilinear approximation (see PVI) which has an exact inverse. Explain.
5. Suggest an application where it might be helpful to have interlaced signal and derivative.
6. A question arises in the obvious extension to two dimensions, $L_x L_y$ or $L_y L_x$ or $(L_x L_y + L_y L_x)/2$. What belongs in location q in the data space

$$\begin{bmatrix} a & b & a \\ b & q & b \\ a & b & a \end{bmatrix} \tag{1.31}$$

where a denotes function values and b denotes derivatives?