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 images) 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 curves	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 2. These techniques 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). Using 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. 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. But nature determines in each application what is the best operator to use, and whether to stop after the adjoint, to go the whole way to the inverse, or to stop partway.

The operators and adjoints above transform vectors to other vectors. They also transform data planes to model planes, volumes, etc. A mathematical operator transforms an "abstract vector" which might be packed full of volumes of information like television signals (time series) can pack together a movie, a sequence of frames. We can always think of the operator as being a matrix but the matrix can be truly huge (and nearly empty). When the vectors transformed by the matrices are large like geophysical data set sizes then the matrix sizes are "large squared," far too big for computers. Thus although we can always think of an operator as a matrix, in practice, we handle an operator differently. Each practical application requires the practitioner to prepare two computer programs. One performs the matrix multiply $\mathbf{y} = \mathbf{A}\mathbf{x}$ and another multiplys by the transpose $\tilde{\mathbf{x}} = \mathbf{A}'\mathbf{y}$ (without ever having the matrix itself in memory). It is always easy to transpose a matrix. It is less easy to take a computer program that does $\mathbf{y} = \mathbf{A}\mathbf{x}$ and convert it to another to do $\tilde{\mathbf{x}} = \mathbf{A}'\mathbf{y}$. In this chapter are many examples of increasing complexity. At the end of the chapter we will see a test for any program pair to see whether the operators $\mathbf{A}$ and $\mathbf{A}'$ are mutually adjoint as they should be. Doing the job correctly (coding adjoints without making approximations) will reward us later when we tackle model and image estimation problems.

1.0.1 Programming linear 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 adjoint
        then erase x
if operator itself
        then erase y
do iy = 1, ny {
do ix = 1, nx {
        if adjoint
                x(ix) = x(ix) + b(iy,ix) $\times$ y(iy)
        if operator itself
                y(iy) = y(iy) + b(iy,ix) $\times$ x(ix)
        }}
```

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 adjoint and the operator itself.

Next we restate the matrix-multiply pseudo code in real code, in a language called **Loptran**[1], a language designed for exposition and research in model fitting and optimization in physical sciences. The module `matmult` for matrix multiply and its adjoint exhibits the style that we will use repeatedly. At last count there were 53 such routines (operator with adjoint) in this book alone.

```
module matmult {   # matrix multiply and its adjoint
real, dimension (:,:), pointer :: bb
#%  _init( bb)
#%  _lop(  x, y)
integer ix, iy
do ix= 1, size(x) {
do iy= 1, size(y) {
        if( adj)
                x(ix) = x(ix) + bb(iy,ix) * y(iy)
        else
                y(iy) = y(iy) + bb(iy,ix) * x(ix)
        }}
}
```

Notice that the module `matmult` does not explicitly erase its output before it begins, as does the psuedo code. That is because Loptran will always erase for you the space required for the operator's output. Loptran also defines a logical variable `adj` for you to to distinguish your computation of the adjoint `x=x+B'*y` from the forward operation `y=y+B*x`.

[1]The programming language, Loptran, is based on a dialect of Fortran called Ratfor. For more details, see Appendix A.

In computerese, the two lines beginning #% are macro expansions that take compact bits of information which expand into the verbose boilerplate that Fortran requires. Loptran is Fortran with these macro expansions. You can always see how they expand by looking at `http://sepwww.stanford.edu/sep/prof/gee/Lib/`.

What is new in Fortran 90, and will be a big help to us, is that instead of a subroutine with a single entry, we now have a module with two entries, one named `_init` for the physical scientist who defines the physical problem by defining the matrix, and another named `_lop` for the least-squares problem solver, the computer scientist who will not be interested in how we specify $\mathbf{B}$, but who will be iteratively computing $\mathbf{Bx}$ and $\mathbf{B}'\mathbf{y}$ to optimize the model fitting. The lines beginning with `#%` are expanded by Loptran into more verbose and distracting Fortran 90 code. The second line in the module `matmult`, however, is pure Fortran syntax saying that `bb` is a pointer to a real-valued matrix.

To use `matmult`, two calls must be made, the first one

```
            call matmult_init( bb)
```

is done by the physical scientist after he or she has prepared the matrix. Most later calls will be done by numerical analysts in solving code like in Chapter 2. These calls look like

```
            stat = matmult_lop( adj, add, x, y)
```

where `adj` is the logical variable saying whether we desire the adjoint or the operator itself, and where `add` is a logical variable saying whether we want to accumulate like $\mathbf{y} \leftarrow \mathbf{y} + \mathbf{Bx}$ or whether we want to erase first and thus do $\mathbf{y} \leftarrow \mathbf{Bx}$. The return value `stat` is an integer parameter, mostly useless (unless you want to use it for error codes).

Operator initialization often allocates memory. To release this memory, you can `call matmult_close()` although in this case nothing really happens.

1.1 FAMILIAR OPERATORS

The simplest and most fundamental linear operators arise when 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 $[a,b]$

$$y \quad = \quad [\, a \; b \,] \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad = \quad ax_1 + bx_2 \tag{1.1}$$

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:

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

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 module `igrad1`, which implements equation (1.3) and its adjoint. The last row in equation (1.3) is optional and depends not on the code shown, but the code that invokes it. It may seem unnatural to append a null row, but it can be a small convenience (when plotting) to have the input and output be the same size.

```
module igrad1 {                                    # gradient in one dimension
#%  _lop( xx,  yy)
integer i
do i= 1, size(xx)-1 {
        if( adj) {
                xx(i+1) = xx(i+1) + yy(i)          # resembles equation (1.2)
                xx(i  ) = xx(i  ) - yy(i)
                }
        else
                yy(i) = yy(i) + xx(i+1) - xx(i)    # resembles equation (1.1)
        }
}
```

The do loop over `i` assures that all values of `yy(i)` are used, whether computing all the *outputs* for the operator itself, or in the adjoint, using all the *inputs*. In switching from operator to adjoint, the outputs switch to inputs. The Loptran dialect allows us to write the inner code of the `igrad1` module more simply and symmetrically using the syntax of C, C++, and Java where expressions like `a=a+b` can be written more tersely as `a+=b`. With this, the heart of module `igrad1` becomes

```
if( adj) {   xx(i+1) += yy(i)
             xx(i)   -= yy(i)
           }
else {       yy(i)   += xx(i+1)
             yy(i)   -= xx(i)
           }
```

where we see that each component of the matrix is handled both by the operator and the adjoint. 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()`. Something odd happens at the ends of the adjoint only if we take the perspective that the adjoint should have been computed one component at a time instead of all together. By not taking that view, we avoid that confusion.

Figure 1.1 illustrates the use of module `igrad1` for each north-south line of a topographic map. We observe that the gradient gives an impression of illumination from a low sun angle. To apply `igrad1` along the 1-axis for each point on the 2-axis of a two-dimensional map, we use the loop

```
do iy=1,ny
      stat = igrad1_lop( adj, add, map(:,iy), ruf(:,iy))
```

On the other hand, to see the east-west gradient, we use the loop

```
do ix=1,nx
      stat = igrad1_lop( adj, add, map(ix,:), ruf(ix,:))
```

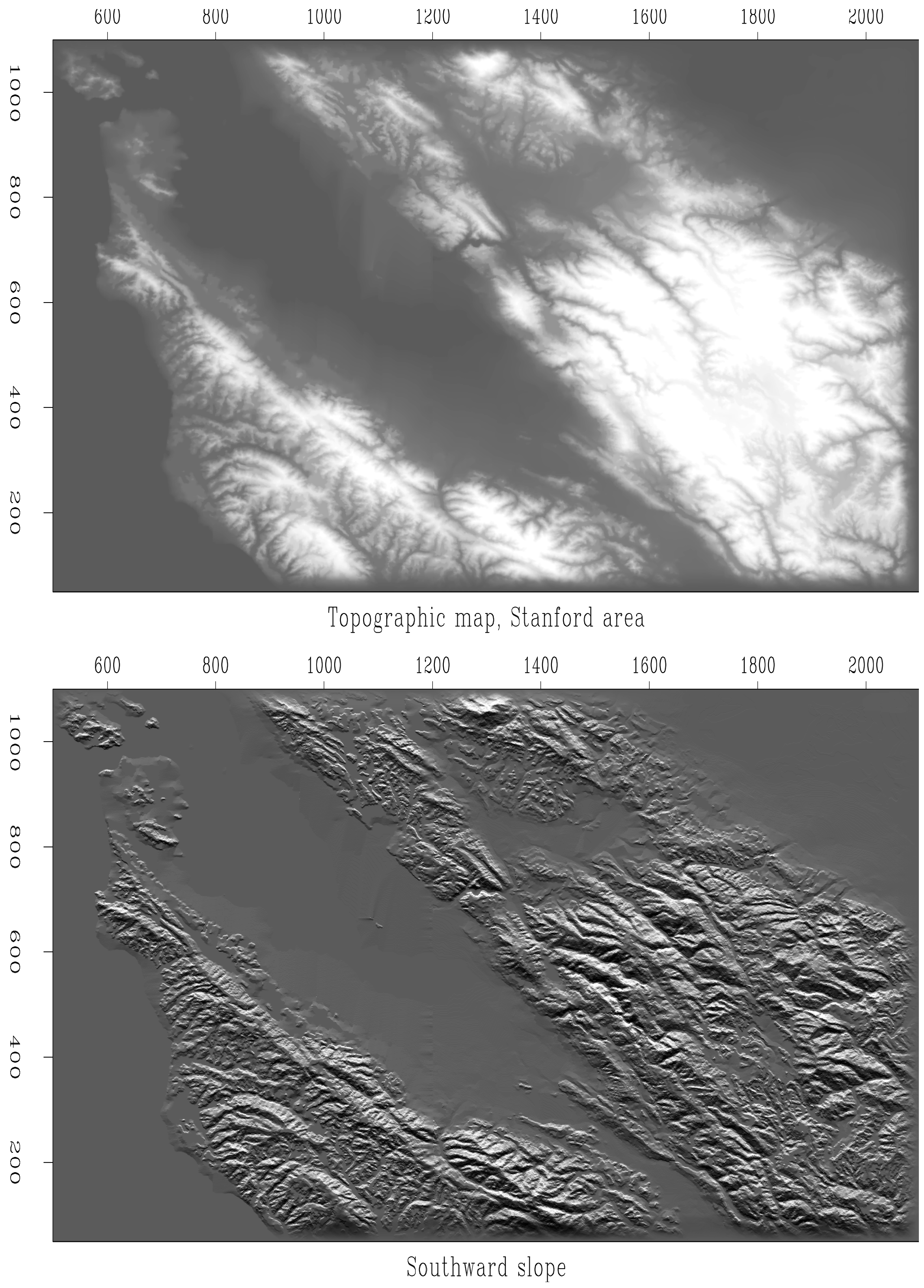

Figure 1.1: Topography near Stanford (top) southward slope (bottom). ajt-stangrad90 [ER,M]

1.1.2 Transient convolution

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

$$\mathbf{y} \quad = \quad \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} \quad = \quad \begin{bmatrix} b_1 & 0 & 0 & 0 & 0 & 0 \\ b_2 & b_1 & 0 & 0 & 0 & 0 \\ b_3 & b_2 & b_1 & 0 & 0 & 0 \\ 0 & b_3 & b_2 & b_1 & 0 & 0 \\ 0 & 0 & b_3 & b_2 & b_1 & 0 \\ 0 & 0 & 0 & b_3 & b_2 & b_1 \\ 0 & 0 & 0 & 0 & b_3 & b_2 \\ 0 & 0 & 0 & 0 & 0 & b_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} \quad = \quad \mathbf{Bx} \tag{1.4}$$

Notice that columns of equation (1.4) all contain the same signal, but with different shifts. This signal is called the filter's impulse response.

Equation (1.4) could be rewritten as

$$\mathbf{y} \quad = \quad \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} \quad = \quad \begin{bmatrix} x_1 & 0 & 0 \\ x_2 & x_1 & 0 \\ x_3 & x_2 & x_1 \\ x_4 & x_3 & x_2 \\ x_5 & x_4 & x_3 \\ x_6 & x_5 & x_4 \\ 0 & x_6 & x_5 \\ 0 & 0 & x_6 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \quad = \quad \mathbf{Xb} \tag{1.5}$$

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 (1.5) and sometimes (1.4). Such solutions begin from the adjoints. The adjoint of (1.4) is

$$\begin{bmatrix} \hat{x}_1 \\ \hat{x}_2 \\ \hat{x}_3 \\ \hat{x}_4 \\ \hat{x}_5 \\ \hat{x}_6 \end{bmatrix} \quad = \quad \begin{bmatrix} b_1 & b_2 & b_3 & 0 & 0 & 0 & 0 & 0 \\ 0 & b_1 & b_2 & b_3 & 0 & 0 & 0 & 0 \\ 0 & 0 & b_1 & b_2 & b_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & b_1 & b_2 & b_3 & 0 & 0 \\ 0 & 0 & 0 & 0 & b_1 & b_2 & b_3 & 0 \\ 0 & 0 & 0 & 0 & 0 & b_1 & b_2 & b_3 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} \tag{1.6}$$

The adjoint ***crosscorrelates*** with the filter instead of convolving with it (because the filter is backwards). Notice that each row in equation (1.6) contains all the filter coefficients and there are no rows where the filter somehow uses zero values off the ends of the data as we saw earlier. In some applications it is important not to assume zero values beyond the interval where inputs are given.

The adjoint of (1.5) crosscorrelates a fixed portion of filter input across a variable portion of filter output.

$$\begin{bmatrix} \hat{b}_1 \\ \hat{b}_2 \\ \hat{b}_3 \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & 0 & 0 \\ 0 & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & 0 \\ 0 & 0 & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} \tag{1.7}$$

Module `tcai1` is used for $\mathbf{y} = \mathbf{Bx}$ and module `tcaf1` is used for $\mathbf{y} = \mathbf{Xb}$.

```
module tcai1 {            # Transient Convolution Adjoint Input 1-D. yy(m1+n1)
real, dimension (:), pointer :: bb
#%  _init( bb)
#%  _lop ( xx, yy)
integer b, x, y
if( size(yy) < size (xx) + size(bb) - 1 ) call erexit('tcai')
do b= 1, size(bb) {
do x= 1, size(xx) {                  y = x + b - 1
        if( adj)        xx(x) +=  yy(y) * bb(b)
        else            yy(y) +=  xx(x) * bb(b)
        }}
}
```

```
module tcaf1 {              # Transient Convolution, Adjoint is the Filter, 1-D
real, dimension (:), pointer :: xx
#%  _init( xx)
#%  _lop ( bb,  yy)
integer        x,   b,   y
if( size(yy) < size(xx) + size(bb) - 1 )   call erexit('tcaf')
do b= 1, size(bb) {
do x= 1, size(xx) {                  y = x + b - 1
        if( adj)        bb(b) +=  yy(y) * xx(x)
        else            yy(y) +=  bb(b) * xx(x)
        } }
}
```

1.1.3 Internal convolution

Convolution is the computational equivalent of ordinary linear differential operators (with constant coefficients). Applications are vast, and end effects are important. Another choice of data handling at ends is that zero data not be assumed beyond the interval where the data is given. This is important in data where the crosscorrelation changes with time. Then it is sometimes handled as constant in short time windows. Care must be taken that zero signal values not be presumed off the ends of those short time windows; otherwise, the many ends of the many short segments can overwhelm the results.

In the sets (1.4) and (1.5), the top two equations explicitly assume that the input data vanishes before the interval on which it is given, and likewise at the bottom. Abandoning the top two and bottom two equations in (1.5) we get:

$$\begin{bmatrix} y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} = \begin{bmatrix} x_3 & x_2 & x_1 \\ x_4 & x_3 & x_2 \\ x_5 & x_4 & x_3 \\ x_6 & x_5 & x_4 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{1.8}$$

The adjoint is

$$\begin{bmatrix} \hat{b}_1 \\ \hat{b}_2 \\ \hat{b}_3 \end{bmatrix} = \begin{bmatrix} x_3 & x_4 & x_5 & x_6 \\ x_2 & x_3 & x_4 & x_5 \\ x_1 & x_2 & x_3 & x_4 \end{bmatrix} \begin{bmatrix} y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} \tag{1.9}$$

The difference between (1.9) and (1.7) is that here the adjoint crosscorrelates a fixed portion of *output* across a variable portion of *input*, whereas with (1.7) the adjoint crosscorrelates a fixed portion of *input* across a variable portion of *output*.

In practice we typically allocate equal space for input and output. Because 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 module `icaf1` says that the output time `y` aligns with the input time `x` for the filter point `b=lag`.

```
module icaf1 {                  # Internal Convolution, Adjoint is Filter. 1-D
integer :: lag
real, dimension (:), pointer :: xx
#%  _init ( xx, lag)
#%  _lop  ( bb,  yy)
integer   x,  b,   y
do b= 1, size(bb) {
        do y= 1+size(bb)-lag, size(yy)-lag+1 {    x= y - b + lag
                if( adj)        bb(b) +=  yy(y) * xx(x)
                else            yy(y) +=  bb(b) * xx(x)
                }
        }
}
```

The value of `lag` always used in this book is `lag=1`. For `lag=1` the module `icaf1` implements not equation (1.8) but (1.10):

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ x_3 & x_2 & x_1 \\ x_4 & x_3 & x_2 \\ x_5 & x_4 & x_3 \\ x_6 & x_5 & x_4 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{1.10}$$

It may seem a little odd to put the required zeros at the beginning of the output, but filters are generally designed so that their strongest coefficient is the first, namely `bb(1)` so the alignment of input and output in equation (1.10) is the most common one.

The **end effect**s of the convolution modules are summarized in Figure 1.2.

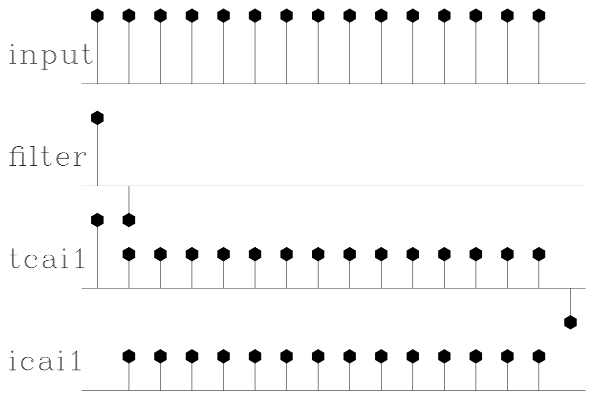

Figure 1.2: Example of convolution end-effects. From top to bottom: input; filter; output of `tcai1()`; output of `icaf1()` also with (`lag=1`). $\boxed{\text{ajt-conv90}}$ [ER]

1.1.4 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 to make a longer vector **y**, add some zeros at the end of **x**. This zero padding can be regarded as the matrix multiplication

$$\mathbf{y} \quad = \quad \begin{bmatrix} \mathbf{I} \\ \mathbf{0} \end{bmatrix} \mathbf{x} \tag{1.11}$$

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

$$\tilde{\mathbf{x}} \quad = \quad \begin{bmatrix} \mathbf{I} & \mathbf{0} \end{bmatrix} \mathbf{y} \tag{1.12}$$

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

```
module zpad1 {                                # Zero pad.  Surround data by zeros. 1-D
#%  _lop( data,  padd)
integer                                       p,  d
do d= 1, size(data) {                         p = d + (size(padd)-size(data))/2
        if( adj)
                        data(d) = data(d) + padd(p)
        else
                        padd(p) = padd(p) + data(d)
        }
}
```

1.1.5 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 modules 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

$$\text{Program} \quad = \quad \mathbf{F} \begin{bmatrix} \mathbf{I} \\ \mathbf{0} \end{bmatrix} \tag{1.13}$$

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 routine is given by

$$\text{Program}' \quad = \quad \begin{bmatrix} \mathbf{I} & \mathbf{0} \end{bmatrix} \mathbf{F}' \tag{1.14}$$

Thus the adjoint routine *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.

1.1.6 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()`.

1.1.7 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 module `bin2`.

```
module bin2 {                                            # Data-push binning in 2-D.
integer :: m1, m2
real    ::  o1,d1,o2,d2
real, dimension (:,:), pointer :: xy
#%  _init(     m1,m2, o1,d1,o2,d2,xy)
#%  _lop ( mm (m1,m2),  dd (:))
integer    i1,i2, id
do id=1,size(dd) {
        i1 = 1.5 + (xy(id,1)-o1)/d1
        i2 = 1.5 + (xy(id,2)-o2)/d2
        if( 1<=i1 && i1<=m1 &&
            1<=i2 && i2<=m2   )
                if( adj)
                        mm(i1,i2) = mm(i1,i2) +  dd( id)
                else
                        dd( id)   = dd( id)   +  mm(i1,i2)
        }
}
```

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 `bin2` to pseudo data of all ones. To capture this idea in an equation, let **B** denote the linear operator in which the bin value is sprayed to the data values. The inverse operation, in which the data values in the bin are summed and divided by the number in the bin, is represented by

$$\mathbf{m} \quad = \quad \mathbf{diag}(\mathbf{B}'\mathbf{1})^{-1}\mathbf{B}'\mathbf{d} \tag{1.15}$$

Empty bins, of course, leave us a problem. That we'll address in chapter 3. In Figure 1.3, the empty bins contain zero values.

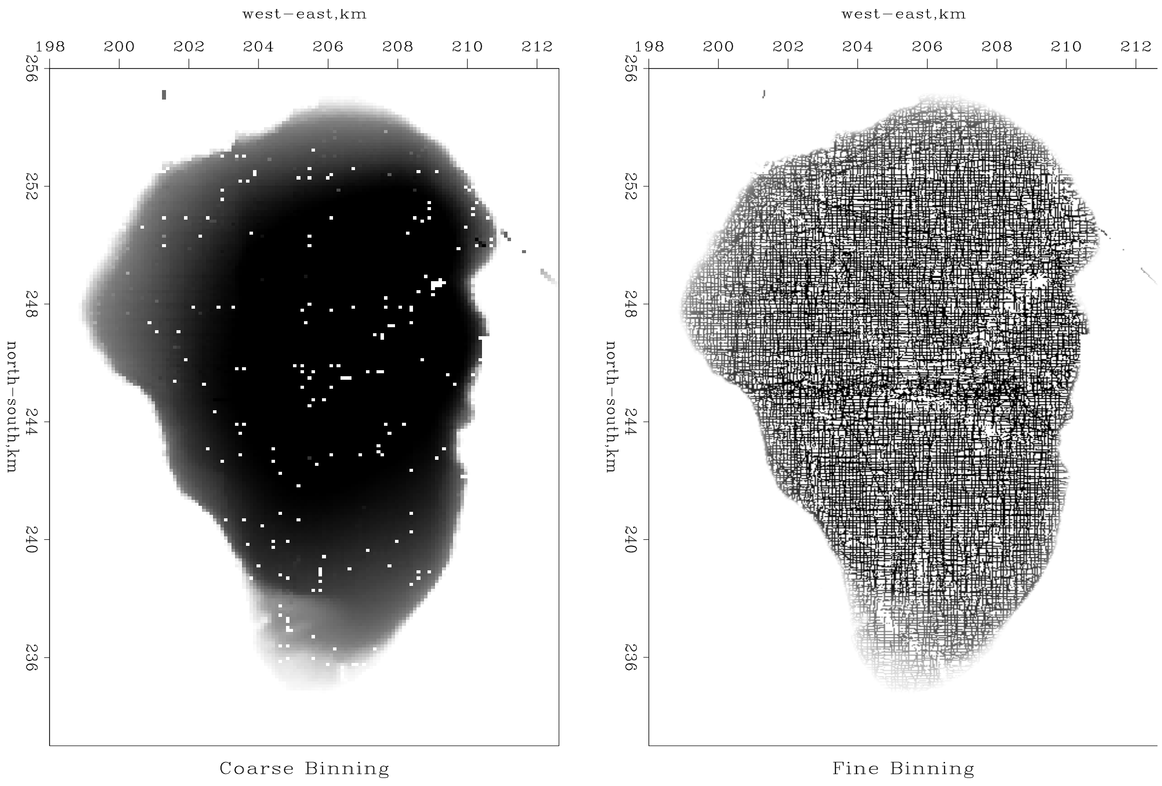

Figure 1.3: Binned depths of the Sea of Galilee. ajt-galbin90 [ER]

1.1.8 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 bin centers 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, the adjoint distributes the data ordinate value between the two nearest bins 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 1.4, we have the operator in (1.16).

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix} \approx \begin{bmatrix} .7 & .3 & . & . & . & . \\ . & . & 1 & . & . & . \\ . & . & . & . & .5 & .5 \end{bmatrix} \begin{bmatrix} m_0 \\ m_1 \\ m_2 \\ m_3 \\ m_4 \\ m_5 \end{bmatrix} \tag{1.16}$$

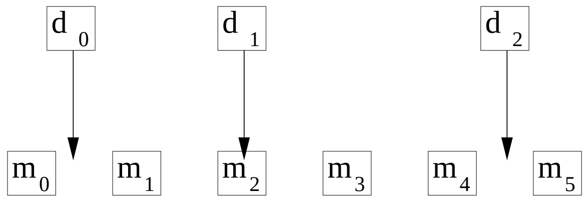

Figure 1.4: Uniformly sampled model space and irregularly sampled data space corresponding to (1.16). ajt-helgerud [NR]

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 (1.16). 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.

Module `lint1` does linear interpolation and its adjoint. In chapters 3 and 6 we build inverse operators.

```
# Nearest-neighbor interpolation would do this:  data = model( 1.5 + (t-t0)/dt)
#                             This is likewise but with _linear_ interpolation.
module lint1 {
real :: o1,d1
real, dimension (:), pointer :: coordinate
#%  _init ( o1,d1, coordinate)
#%  _lop  ( mm, dd)
integer i, im,  id
real    f, fx,gx
do id= 1, size(dd) {
        f = (coordinate(id)-o1)/d1;     i=f  ;   im= 1+i
        if( 1<=im && im< size(mm)) {   fx=f-i;   gx= 1.-fx
                if( adj) {
                        mm(im  ) +=  gx * dd(id)
                        mm(im+1) +=  fx * dd(id)
                        }
                else
                        dd(id)   +=  gx * mm(im)  +  fx * mm(im+1)
                }
        }
}
```

1.1.9 Spray and sum : scatter and gather

Perhaps the most common operation is the summing of 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 a row vector of ones. Its adjoint is a column vector of ones. In one dimension this operator is almost too easy for us to bother showing a routine. 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 module `spraysum`, both input and output are taken 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)`. When module `spraysum` is given the input dimensions and output dimensions stated below, the operations stated alongside are implied.

`(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, according to the specified axis lengths.

```
module spraysum {                    # Spray or sum over 1, 2, and/or 3-axis.
integer ::   n1,n2,n3,     m1,m2,m3
#%  _init(   n1,n2,n3,     m1,m2,m3)
#%  _lop( xx(n1,n2,n3), yy(m1,m2,m3))
integer i1,i2,i3,  x1,x2,x3, y1,y2,y3
        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)  xx(x1,x2,x3)  +=  yy(y1,y2,y3)
        else      yy(y1,y2,y3)  +=  xx(x1,x2,x3)
        }}}
}
```

1.1.10 Causal and leaky integration

Causal integration is defined as

$$y(t) \quad = \quad \int_{-\infty}^{t} x(\tau)\, d\tau \tag{1.17}$$

Leaky integration is defined as

$$y(t) \quad = \quad \int_{0}^{\infty} x(t-\tau)\, e^{-\alpha\tau}\, d\tau \tag{1.18}$$

As $\alpha \to 0$, 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 that we should call causal summation, but we often call it causal integration. Equation (1.19) represents causal integration for $\rho = 1$ and leaky integration for $0 < \rho < 1$.

$$\mathbf{y} \quad = \quad \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} \quad = \quad \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \rho & 1 & 0 & 0 & 0 & 0 & 0 \\ \rho^2 & \rho & 1 & 0 & 0 & 0 & 0 \\ \rho^3 & \rho^2 & \rho & 1 & 0 & 0 & 0 \\ \rho^4 & \rho^3 & \rho^2 & \rho & 1 & 0 & 0 \\ \rho^5 & \rho^4 & \rho^3 & \rho^2 & \rho & 1 & 0 \\ \rho^6 & \rho^5 & \rho^4 & \rho^3 & \rho^2 & \rho & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} \quad = \quad \mathbf{Cx} \qquad (1.19)$$

(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 that for the 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 (1.19) is more efficiently thought of as the recursion

$$y_t \quad = \quad \rho\, y_{t-1} + x_t \qquad\qquad t \text{ increasing} \qquad (1.20)$$

(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 (1.19) than of (1.20). Let the matrix of equation (1.19) 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, as in

$$\tilde{x}_{t-1} \quad = \quad \rho \tilde{x}_t + y_{t-1} \qquad\qquad t \text{ decreasing} \qquad (1.21)$$

Thus the adjoint of causal integration is **anticausal integration**.

A module to do these jobs is `leakint`. 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 1.5 illustrates the program for $\rho = 1$.

```
module leakint {                                                  # leaky integration
real ::    rho
#%  _init( rho)
#%  _lop ( xx,  yy)
integer i, n
real tt
n = size (xx); tt = 0.
if( adj)
        do i= n, 1, -1 {   tt = rho*tt + yy(i)
                xx(i)  +=  tt
                }
else
        do i= 1, n {       tt  = rho*tt + xx(i)
                yy(i)  +=  tt
                }
}
```

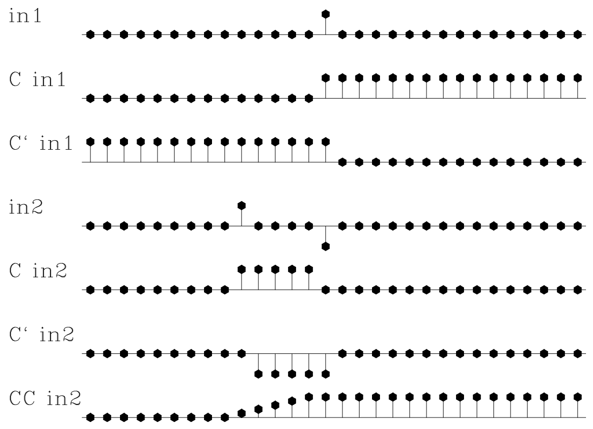

Figure 1.5: `in1` is an input pulse. `C in1` is its causal integral. `C′ in1` is the anticausal integral of the pulse. `in2` is a separated doublet. Its causal integration is a box and its anticausal integration is a negative box. `CC in2` is the double causal integral of `in2`. How can an equilateral triangle be built? $\boxed{\text{ajt-causint90}}$ [ER]

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 Consider the matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & \rho & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.22}$$

and others like it with ρ in other locations. Show what combination of these matrices will represent the leaky integration matrix in equation (1.19). What is the adjoint?

2 Modify the calculation in Figure 1.5 so that there is 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` on the preceding page by changing the diagonal to contain 1/2 instead of 1. Notice how time alignment changes in Figure 1.5.

1.1.11 Backsolving, polynomial division and deconvolution

Ordinary differential equations often lead us to the backsolving operator. For example, the damped harmonic oscillator leads to a special case of equation (1.23) where $(a_3, a_4, \cdots) = 0$.

There is a huge literature on finite-difference solutions of ordinary differential equations that lead to equations of this type. Rather than derive such an equation on the basis of many possible physical arrangements, we can begin from the filter transformation **B** in (1.4) but put the matrix on the other side of the equation so our transformation can be called one of inversion or backsubstitution. Let us also force the matrix **B** to be a square matrix by truncating it with $\mathbf{T} = [\mathbf{I} \quad \mathbf{0}]$, say $\mathbf{A} = [\mathbf{I} \quad \mathbf{0}]\mathbf{B} = \mathbf{TB}$. To link up with applications in later chapters, I specialize to 1's on the main diagonal and insert some bands of zeros.

$$\mathbf{Ay} \quad = \quad \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ a_1 & 1 & 0 & 0 & 0 & 0 & 0 \\ a_2 & a_1 & 1 & 0 & 0 & 0 & 0 \\ 0 & a_2 & a_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & a_2 & a_1 & 1 & 0 & 0 \\ a_5 & 0 & 0 & a_2 & a_1 & 1 & 0 \\ 0 & a_5 & 0 & 0 & a_2 & a_1 & 1 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} \quad = \quad \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} \quad = \quad \mathbf{x} \qquad (1.23)$$

Algebraically, this operator goes under the various names, "backsolving", "polynomial division", and "deconvolution". The leaky integration transformation (1.19) is a simple example of backsolving when $a_1 = -\rho$ and $a_2 = a_5 = 0$. To confirm this, you need to verify that the matrices in (1.23) and (1.19) are mutually inverse.

A typical row in equation (1.23) says

$$x_t \quad = \quad y_t + \sum_{\tau>0} a_\tau \, y_{t-\tau} \qquad (1.24)$$

Change the signs of all terms in equation (1.24) and move some terms to the opposite side

$$y_t \quad = \quad x_t - \sum_{\tau>0} a_\tau \, y_{t-\tau} \qquad (1.25)$$

Equation (1.25) is a recursion to find y_t from the values of y at earlier times.

The most rudimentary form of mathematics in which this recursion occurs is in connection with multiplying and dividing polynomials. The transformation (1.23) can be derived by multiplying two polynomials, $Y(Z) = y_0 + y_1 Z + y_2 Z^2 + y_3 Z^3 + y_4 Z^4 + y_5 Z^5$ times $A(Z) = 1 + a_1 Z + a_2 Z^2 + a_5 Z^5$. Identifying the k-th power of Z in the product $X(Z) = A(Z)Y(Z)$ gives the k-th row of the transformation (1.23). Thus the operator in (1.23) is $Y(Z) = X(Z)/A(Z)$. The polynomials in Z are called Z transforms. They may be recognized as Fourier transforms where $Z = e^{i\omega\Delta t}$. Convolution corresponds to multiplying polynomials (convolving the coefficients) and deconvolution (or backsolving) corresponds to polynomial division.

A causal operator is one that uses its present and past inputs to make its current output. Anticausal operators use the future but not the past. Causal operators are generally associated with lower triangular matrices and positive powers of Z, whereas anticausal operators are associated with upper triangular matrices and negative powers of Z. A transformation like equation (1.23) but with the transposed matrix would require us to run the recursive solution the opposite direction in time, as we did with leaky integration.

A module to backsolve (1.23) is `polydiv1`.

```
module polydiv1 {                    # Polynomial division (recursive filtering)
real, dimension (:), pointer :: aa
#%  _init ( aa)
#%  _lop  ( xx, yy)
integer  ia, ix, iy
real     tt
if( adj)
        do ix= size(xx), 1, -1 {
                tt = yy( ix)
                do ia = 1, min( size(aa), size (xx) - ix) {
                        iy = ix + ia
                        tt -=  aa( ia) * xx( iy)
                        }
                xx( ix) = xx( ix) + tt
                }
else
        do iy= 1, size(xx) {
                tt = xx( iy)
                do ia = 1, min( size(aa), iy-1) {
                        ix = iy - ia
                        tt -=  aa( ia) * yy( ix)
                        }
                yy( iy) =  yy( iy) + tt
                }
}
```

The more complicated an operator, the more complicated is its adjoint. Given a transformation from $\mathbf{x}$ to $\mathbf{y}$ that is $\mathbf{TBy} = \mathbf{x}$, we may wonder if the adjoint transform really is $(\mathbf{TB})'\hat{\mathbf{x}} = \mathbf{y}$. It amounts to asking if the adjoint of $\mathbf{y} = (\mathbf{TB})^{-1}\mathbf{x}$ is $\hat{\mathbf{x}} = ((\mathbf{TB})')^{-1}\mathbf{y}$. Mathematically we are asking if the inverse of a transpose is the transpose of the inverse. This is so because in $\mathbf{AA}^{-1} = \mathbf{I} = \mathbf{I}' = (\mathbf{A}^{-1})'\mathbf{A}'$ the parenthesized object must be the inverse of its neighbor $\mathbf{A}'$.

The adjoint has a meaning which is nonphysical. It is like the forward operator except that we must begin at the final time and revert towards the first. The adjoint pendulum damps as we compute it backward in time, but that, of course, means that the adjoint pendulum diverges as it is viewed moving forward in time.

1.1.12 The basic low-cut filter

Many geophysical measurements contain very low-frequency noise called "drift." For example, it might take some months to survey the depth of a lake. Meanwhile, rainfall or evaporation could change the lake level so that new survey lines become inconsistent with old ones. Likewise, gravimeters are sensitive to atmospheric pressure, which changes with the weather. A magnetic survey of an archeological site would need to contend with the fact that the earth's main magnetic field is changing randomly through time while the survey is being done. Such noises are sometimes called "secular noise."

The simplest way to eliminate low frequency noise is to take a time derivative. A disadvantage is that the derivative changes the waveform from a pulse to a doublet (finite difference).

Here we examine the most basic low-cut filter. It preserves the waveform at high frequencies; it has an adjustable parameter for choosing the bandwidth of the low cut; and it is causal (uses the past but not the future).

We make our causal lowcut filter (highpass filter) by two stages which can be done in either order.

1. Apply a time derivative, actually a finite difference, convolving the data with $(1,-1)$.
2. Integrate, actually to do a leaky integration, to deconvolve with $(1,-\rho)$ where numerically, ρ is slightly less than unity.

The convolution ensures that the zero frequency is removed. The leaky integration almost undoes the differentiation (but does not restore the zero frequency). Adjusting the numerical value of ρ adjusts the cutoff frequency of the filter. To learn the impulse response of the combined processes, we need to convolve the finite difference $(1,-1)$ with the leaky integration $(1,\rho,\rho^2,\rho^3,\rho^4,\cdots)$. The result is $(1,\rho,\rho^2,\rho^3,\rho^4,\cdots)$ minus $(0,1,\rho,\rho^2,\rho^3,\cdots)$. We can think of this as $(1,0,0,0,0,\cdots)$ minus $(1-\rho)(1,\rho,\rho^2,\rho^3,\cdots)$. In other words the impulse response is an impulse followed by the negative of a weak $(1-\rho)$ decaying exponential ρ^t. Roughly speaking, the cutoff frequency of the filter corresponds to matching one wavelength to the exponential decay time.

Some exercise with Fourier transforms or Z-transforms[2], shows the Fourier transform of this highpass filter filter to be

$$H(Z) \quad = \quad \frac{1-Z}{1-\rho Z} \quad = \quad 1-(1-\rho)[Z^1+\rho Z^2+\rho^2 Z^3+\rho^3 Z^4\cdots] \tag{1.26}$$

where the unit-delay operator is $Z=e^{i\omega\Delta t}$ and where ω is the frequency. A symmetical (noncausal) lowcut filter would filter once forward with $H(Z)$ and once backwards (adjoint) with $H(1/Z)$.

Seismological data is more complex. A single "measurement" consists of an explosion and echo signals recorded at many locations. As before, a complete survey is a track (or tracks) of explosion locations. Thus, in seismology, data space is higher dimensional. Its most troublesome noise is not simply low frequency; it is low velocity. We will do more with multidimensional data in later chapters.

EXERCISES:

1 Give an analytic expression for the waveform of equation (1.26).

2 Define a low-pass filter as $1-H(Z)$. What is the low-pass impulse response?

3 Put Galilee data on a coarse mesh. Consider north-south lines as one-dimensional signals. Find the value of ρ for which H is the most pleasing filter.

[2]An introduction to Z-transforms is found in my earlier books, FGDP and ESA-PVI.

4 Find the value of ρ for which $\bar{H}H$ is the most pleasing filter.

5 Find the value of ρ for which H applied to Galilee has minimum energy. (Experiment with a range of about ten values around your favorite value.)

6 Find the value of ρ for which $\bar{H}H$ applied to Galilee has minimum energy.

7 Repeat above for east-west lines.

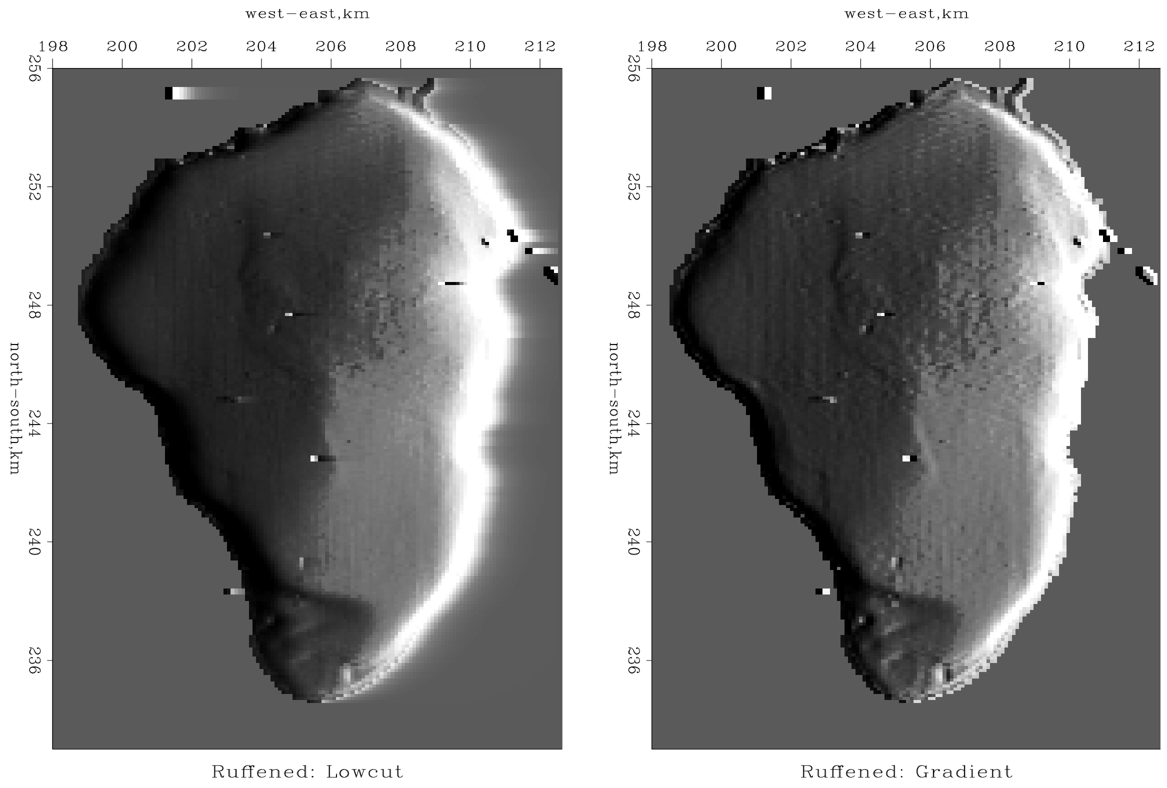

Figure 1.6: The depth of the Sea of Galilee after roughening. $\boxed{\text{ajt-galocut90}}$ [ER,M]

1.1.13 Nearest-neighbor normal moveout (NMO)

Normal-moveout correction (**NMO**) is a geometrical correction of reflection seismic 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 these are 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 each 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 module that does reverse moveout is `hypotenusei`. Given a zero-offset trace, it makes another at non-zero offset. The adjoint does the usual normal moveout correction.

```
module hypotenusei {                                # Inverse normal moveout
integer :: nt
integer, dimension (nt), allocatable :: iz
#%  _init( nt, t0, dt, xs)
integer it
real t0, dt, xs, t, zsquared
do it= 1, nt {  t = t0 + dt*(it-1)
        zsquared =  t * t - xs * xs
        if ( zsquared >= 0.)
                iz (it) = 1.5 + (sqrt( zsquared) - t0) /dt
        else
                iz (it) = 0
        }
#%  _lop(   zz, tt)
integer   it
do it= 1, nt {
             if ( iz (it) > 0 ) {
                            if( adj)   zz( iz(it))  +=  tt(    it )
                            else       tt(it)       +=  zz( iz(it))
                }
        }
}
```

A companion routine `imospray` loops over offsets and makes 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.

```
module imospray {                   # inverse moveout and spray into a gather.
use hypotenusei
real :: x0,dx, t0,dt
integer :: nx,nt
#%  _init (slow, x0,dx, t0,dt, nt,nx)
                real slow
                x0 = x0*slow
                dx = dx*slow
#% _lop(  stack(nt),    gather(nt,nx))
integer ix, stat
do ix= 1, nx {
        call hypotenusei_init ( nt, t0, dt, x0 + dx*(ix-1))
        stat = hypotenusei_lop ( adj, .true., stack, gather(:,ix))
        }
call hypotenusei_close ()
}
```

1.2 ADJOINT DEFINED: DOT-PRODUCT TEST

Having seen many examples of **space**s, operators, and adjoints, we should now see 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 not only does this packing include one-dimensional spaces like signals, two-dimensional spaces like images, 3-D movie cubes, and zero-dimensional spaces like a data mean, etc, but spaces can be sets of all the above. One 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, because 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.

We briefly recall information about vector spaces found in elementary books: Let α be any scalar. Then if $\mathbf{d}_1$ is a vector and $\mathbf{d}_2$ is conformable with it, then other vectors are $\alpha\mathbf{d}_1$ and $\mathbf{d}_1 + \mathbf{d}_2$. The size measure of a vector 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 with the use of dot products).

1.2.2 Dot-product test for validity of an adjoint

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. The **dot-product test** enables us to ascertain whether the program for the adjoint of an operator is precisely consistent with the operator itself. It can be, and it should be.

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 routines. The first routine amounts to the matrix multiplication $\mathbf{F}\mathbf{x}$. The adjoint routine computes $\mathbf{F}'\mathbf{y}$, where $\mathbf{F}'$ is the **conjugate-transpose** matrix. In later chapters we will be solving huge sets of simultaneous equations, in which both routines are required. If the pair of routines are inconsistent, we are doomed from the start. The dot-product test is a simple test for verifying that the two routines 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 $\mathbf{y}'\mathbf{F}\mathbf{x}$ because we get the same result no matter where we put the parentheses. They serve only to determine the sequence of computation. Thus,

$$\mathbf{y}'(\mathbf{F}\mathbf{x}) = (\mathbf{y}'\mathbf{F})\mathbf{x} \tag{1.27}$$
$$\mathbf{y}'(\mathbf{F}\mathbf{x}) = (\mathbf{F}'\mathbf{y})'\mathbf{x} \tag{1.28}$$

(In general, the matrix is not square.) To perform the dot-product test, load the vectors $\mathbf{x}$ and $\mathbf{y}$ with random numbers. Using your program for $\mathbf{F}$, compute the vector $\tilde{\mathbf{y}} = \mathbf{F}\mathbf{x}$, and using your program for $\mathbf{F}'$, compute $\tilde{\mathbf{x}} = \mathbf{F}'\mathbf{y}$. Inserting these into equation (1.28) gives you two scalars that should be equal.

$$\mathbf{y}'(\mathbf{F}\mathbf{x}) \quad = \quad \mathbf{y}'\tilde{\mathbf{y}} \quad = \quad \tilde{\mathbf{x}}'\mathbf{x} \quad = \quad (\mathbf{F}'\mathbf{y})'\mathbf{x} \tag{1.29}$$

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). The program for applying the dot product test is `dot_test` on the current page. The Fortran way of passing a linear operator as an argument is to specify the function interface. Fortunately, we have already defined the interface for a generic linear operator. To use the `dot_test` program, you need to initialize an operator with specific arguments (the `_init` subroutine) and then pass the operator itself (the `_lop` function) to the test program. You also need to specify the sizes of the model and data vectors so that temporary arrays can be constructed. The program runs the dot product test twice, second time with `add = .true.` to test if the operator can be used properly for accumulating the result like $\mathbf{y} \leftarrow \mathbf{y} + \mathbf{B}\mathbf{x}$.

```
module dottest {
  logical, parameter, private ::  T = .true.,  F = .false.
contains
  subroutine dot_test( oper, n_mod, n_dat, dot1, dot2) {
    integer,             intent (in)  :: n_mod, n_dat
    real, dimension (2), intent (out) :: dot1, dot2
    interface {
       function oper( adj, add, mod, dat) result(stat) {
            integer              :: stat
            logical, intent (in) :: adj, add
            real, dimension (:)  :: mod, dat
            }
       }
    real, dimension( n_mod)           :: mod1, mod2
    real, dimension( n_dat)           :: dat1, dat2
    integer                           :: stat
    call random_number( mod1);  call random_number( dat2)
    stat = oper( F, F, mod1, dat1);  dot1( 1) = dot_product( dat1, dat2)
    stat = oper( T, F, mod2, dat2);  dot1( 2) = dot_product( mod1, mod2)
    write (0,*) dot1
    stat = oper( F, T, mod1, dat1);  dot2( 1) = dot_product( dat1, dat2)
    stat = oper( T, T, mod2, dat2);  dot2( 2) = dot_product( mod1, mod2)
    write (0,*) dot2
  }
}
```

I tested (1.29) 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, to obtain inverse operators, you can take advantage of the standard methods of mathematics.

We can speak of a **continuous function** $f(t)$ or a **discrete function** 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 not developed 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 can be computed without ever using division, so potentially the adjugate can be useful in applications where an inverse matrix does not exist. 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 and substituted the definition, "**conjugate transpose**". Unfortunately this was often abbreviated to "conjugate," which caused even more confusion. Thus I decided to use the word adjoint and have it always mean the Hilbert adjoint found in physics and engineering.

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 routines that solve differential equations and routines 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 routines that apply them to a vector. With sparse matrices you easily can do $\mathbf{A}(\mathbf{B}(\mathbf{C}\mathbf{x}))$ but not $(\mathbf{ABC})\mathbf{x}$.

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, you would get a matrix column. The adjoint operator is one from which we can extract the transpose matrix. For large spaces this extraction is unwieldy, so to test the validity of adjoints, we probe 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 $\mathbf{ABC}$ 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 the application of adjoint components in reverse order. The other way we often see complicated operators being built from simple ones is when operators are put into components of matrices, typically a 1×2 or 2×1 matrix containing two operators. An example of the adjoint of a two-component column operator is

$$\begin{bmatrix} \mathbf{A} \\ \mathbf{B} \end{bmatrix}' \quad = \quad \begin{bmatrix} \mathbf{A}' & \mathbf{B}' \end{bmatrix} \tag{1.30}$$

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 it can be applied to a space, producing another space.

As matrices have inverses, so do linear operators. You don't need subscripts to find an inverse. The conjugate-gradient method and conjugate-direction method explained in the next chapter are attractive methods of finding them. They merely apply $\mathbf{A}$ and $\mathbf{A}'$ and use inner products to find coefficients of a polynomial in $\mathbf{AA}'$ that represents the inverse operator.

Whenever we encounter a **positive-definite** matrix we should recognize its likely origin in a nonsymmetric matrix $\mathbf{F}$ times its adjoint. Those in natural sciences often work on solving simultaneous equations but fail to realize that they should return to the origin of the equations which is often a fitting goal; i.e., applying an operator to a model should yield data, i.e., $\mathbf{d} \approx \mathbf{d}_0 + \mathbf{F}(\mathbf{m} - \mathbf{m}_0)$ where the operator $\mathbf{F}$ is a partial derivative matrix (and there are potential underlying nonlinearities). This begins another story with new ingredients, weighting functions and statistics.

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 the adjustment of components in a vector of model parameters $\mathbf{m}$.

$$\mathbf{d}_{\text{obs}} \quad \approx \quad \mathbf{d}_{\text{theor}} \quad = \quad \mathbf{Fm} \tag{1.31}$$

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

$$\begin{aligned} \hat{\mathbf{m}}_1 &= (\mathbf{F}'\mathbf{F})^{-1}\mathbf{F}'\mathbf{d} && (1.32) \\ \hat{\mathbf{m}}_2 &= \mathbf{F}'(\mathbf{FF}')^{-1}\mathbf{d} && (1.33) \end{aligned}$$

Some reasons for the literature being huge are the many questions about the existence, quality, and cost of the inverse operators. Before summarizing that, let us quickly see why these two solutions are reasonable. Inserting equation (1.31) into equation (1.32), and inserting equation (1.33) into equation (1.31), we get the reasonable statements:

$$\begin{aligned} \hat{\mathbf{m}}_1 &= (\mathbf{F}'\mathbf{F})^{-1}(\mathbf{F}'\mathbf{F})\mathbf{m} &&= \mathbf{m} && (1.34) \\ \hat{\mathbf{d}}_{\text{theor}} &= (\mathbf{FF}')(\mathbf{FF}')^{-1}\mathbf{d} &&= \mathbf{d} && (1.35) \end{aligned}$$

Equation (1.34) says that the estimate $\hat{\mathbf{m}}_1$ gives the correct model $\mathbf{m}$ if you start from the theoretical data. Equation (1.35) says that the model estimate $\hat{\mathbf{m}}_2$ gives the theoretical data if we derive $\hat{\mathbf{m}}_2$ 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 $\mathbf{F}$ is a tall matrix (more data values than model values) then the matrix for finding $\hat{\mathbf{m}}_1$ is invertible while that for finding $\hat{\mathbf{m}}_2$ is not, and when the matrix is wide instead of tall (the number of data values is less than the number of model values) it is the other way around. In many applications neither $\mathbf{F}'\mathbf{F}$ nor $\mathbf{FF}'$ 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 $\mathbf{FF}'$ or $\mathbf{F}'\mathbf{F}$ equals $\mathbf{I}$ (unitary operator), that the adjoint operator $\mathbf{F}'$ is the inverse $\mathbf{F}^{-1}$ by either equation (1.32) or (1.33).

Theoreticians like to study inverse problems where **m** is drawn from the field of continuous functions. This is like the vector **m** having infinitely many components. Such problems are hopelessly intractable unless we find, or assume, that the operator $\mathbf{F}\mathbf{F}'$ is an identity or diagonal matrix.

In practice, theoretical considerations may have little bearing on how we proceed. Current computational power limits matrix inversion jobs to about 10^4 variables. This book specializes in big problems, those with more than about 10^4 variables, but the methods we learn are also excellent for smaller problems.

1.2.6 Automatic adjoints

Computers are not only able to perform computations; they can do mathematics. Well known software is Mathematica and Maple. Adjoints can also be done by symbol manipulation. For example Ralf Giering[3] offers a program for converting linear operator programs into their adjoints.

EXERCISES:

1 Suppose a linear operator **F** has its input in the discrete domain and its output in the continuum. How does the operator resemble a matrix? Describe the operator $\mathbf{F}'$ that 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?

[3]http://klima47.dkrz.de/giering/tamc/tamc.html

Index

adjoint, 1, 12, 26
adjoint
    defined, 24
    operator, 27
    truncation errors, 26
adjugate, 27
anticausal integration, 17

back projection, 2
bin2 operator module, 13

causal integration, 16
conjugate transpose, 27
conjugate-transpose, 25
continuous function, 26
convolution, 8
convolution
    internal, 9
    transient, 8
crosscorrelate, 8

damping, 28
data tracks, 24
data-push binning, 13
derivative, 5
determinant, 27
differential equation, 17
discrete function, 26
dot product, 24
dot-product test, 25, 26
dottest module, 25

end effect, 11

filter impulse response, 5
Fourier analysis, 2
Fourier transformation, 12

gather, 15

Hilbert adjoint, 26
hypotenusei operator module, 23

icaf1 operator module, 10
igrad1 operator module, 5
imospray operator module, 23
index, 31
integration
    causal, 16
    leaky, 16
inverse matrix, 27
inversion, 1

leakint operator module, 17
leaky integration, 16
linear interpolation, 14
lint1 operator module, 15
Loptran, 3

matmult operator module, 3
matrix multiply, 1, 2
modeling, 2
module
    `dottest`, dot-product test, 25

nearest neighbor binning, 13
nearest neighbor coordinates, 12
NMO, 22
NMO stack, 22
norm, 24
normal moveout, 22

operator, 1, 27
    `bin2`, push data into bin, 13
    `hypotenusei`, inverse moveout, 23
    `icaf1`, convolve internal, 10
    `igrad1`, first difference, 5
    `imospray`, inverse NMO spray, 23
    `leakint`, leaky integral, 17

`lint1`, linear interp, 15
`matmult`, matrix multiply, 3
`polydiv1`, deconvolve, 19
`spraysum`, sum and spray, 16
`tcaf1`, transient convolution, 9
`tcai1`, transient convolution, 9
`zpad1`, zero pad 1-D, 11
operator
adjoint, 27

polydiv1 operator module, 19
positive-definite, 28
processing, 2
pseudocode, 3

recursion
integration, 17

scatter, 15
space, 24
spray, 15
spraysum operator module, 16
stack, 22
summation operator, 15

tcaf1 operator module, 9
tcai1 operator module, 9
tomography, 2
traveltime depth, 22
truncation, 11, 12, 26

vector space, 24

weighting function, 24

zero pad, 11, 12
zpad1 operator module, 11