Basic operators and adjoints

Jon Claerbout

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! Better because the adjoint operator tolerates imperfections in the data and does not demand the data provide full information.

Using the methods of this chapter, you 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 neighborhoods	sum within 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
scalar field gradient	negative of vector field divergence
upward continue	downward continue
diffraction modeling	imaging by migration
hyperbola modeling	stacking for image or velocity
chop image into overlapping patches	merge the patches
ray tracing	**tomography**

The left column is often called "**modeling**," and the adjoint operators in the right column 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 **??**. 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 therefore, phase shifts of modeling operators. The inverse operator does also, but it also divides out the color. For example, when linear interpolation is done, then high frequencies are smoothed out; inverse interpolation must restore them. You can imagine the possibilities for noise amplification which 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 previously shown transform vectors to other vectors. They also transform data planes to model planes, volumes, etc. A mathematical operator transforms an "abstract vector" that 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 $\mathbf{y} = \mathbf{B}\mathbf{x}$, while the other multiplies by the transpose $\tilde{\mathbf{x}} = \mathbf{B}^{\mathrm{T}}\mathbf{y}$ (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 $\mathbf{y} = \mathbf{B}\mathbf{x}$ and convert it to another to do $\tilde{\mathbf{x}} = \mathbf{B}^{\mathrm{T}}\mathbf{y}$, which is what we'll be doing here. In this chapter are many examples of increasing complexity. At the end of the chapter, we see a test for any program pair to see whether the operators $\mathbf{B}$ and $\mathbf{B}^{\mathrm{T}}$ are mutually adjoint as they should be. Doing the job correctly (coding adjoints without making approximations) rewards us later when we tackle model and image-estimation applications.

Mathematicians often denote the transpose of a matrix $\mathbf{B}$ by $\mathbf{B}^{\mathrm{T}}$. In physics and engineering, we often encounter complex numbers. There, the adjoint is the complex-conjugate transposed matrix denoted $\mathbf{B}^{*}$. What this book calls the adjoint is more properly called the Hilbert adjoint.

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 $\tilde{\mathbf{x}} = \mathbf{B}^{\mathrm{T}}\mathbf{y}$:

```
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 preceding example is a prototype of many to follow; therefore observe carefully the similarities and differences between the operator and its adjoint.

Next, we restate the matrix-multiply pseudo code in real code.

The module `matmult` for matrix multiply along with its adjoint exhibits the style we use repeatedly. At last count there were 53 such routines (operator with adjoint) in this book alone.

user/pwd/matmult.c

```c
void matmult_lop (bool adj, bool add,
                  int nx, int ny, float* x, float*y)
/*< linear operator >*/
{
    int ix, iy;
    sf_adjnull (adj, add, nx, ny, x, y);
    for (ix = 0; ix < nx; ix++) {
            for (iy = 0; iy < ny; iy++) {
                    if (adj) x[ix] += B[iy][ix] * y[iy];
                    else     y[iy] += B[iy][ix] * x[ix];
            }
    }
}
```

We now have a module with two entries/ One is named `_init` for the physical scientist who defines the physical problem by defining the operator. The other is named `_lop` for the least-squares problem solvers, computer scientists not interested in how we specify $\mathbf{B}$. They will be iteratively computing $\mathbf{Bx}$ and $\mathbf{B}^{\mathrm{T}}\mathbf{y}$ to optimize the model fitting.

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

```
matmult_init( bb);
```

is done by physical scientists to set up the operation. Most later calls are done by numerical analysts in solving code like in Chapter **??**. These calls look like

```
            matmult_lop( adj, add, nx, ny, 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 $\mathbf{y} \leftarrow \mathbf{y} + \mathbf{B}\mathbf{x}$ or whether we want to erase first and thus do $\mathbf{y} \leftarrow \mathbf{B}\mathbf{x}$.

We split operators into two independent processes; the first is used for geophysical set up, while the second is invoked by mathematical library code (introduced in the next chapter) to find the model that best fits the data. Here is why we do so. It is important that the math code contain nothing about the geophysical particulars. This independence enables us to use the same math code on many different geophysical applications. This concept of "information hiding" arrived late in human understanding of what is desirable in a computer language. Subroutines and functions are the way new programs use old ones. Object modules are the way old programs (math solvers) are able to use new ones (geophysical operators).

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*.

Generally, inputs and outputs are high dimensional, such as signals or images. Push gives ugly outputs. Some output locations may be empty, each having an erratic number of contributions. Consequently, most data processing (adjoint) is done by *pull*.

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

$$y \quad = \quad [\ a\ b\ ] \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad = \quad a x_1 + b x_2 \tag{1}$$

has an adjoint that is two assignments:

$$\begin{bmatrix} \hat{x}_1 \\ \hat{x}_2 \end{bmatrix} \quad = \quad \begin{bmatrix} a \\ b \end{bmatrix} y \tag{2}$$

$$\boxed{\text{The adjoint of a sum of } N \text{ terms is a collection of } N \text{ assignments.}}$$

Adjoint derivative

In numerical analysis, we represent the derivative of a time function by a finite difference. This subtracts neighboring time points and divides by the sample interval Δt. Finite difference amounts to convolution with the filter $(1, -1)/\Delta t$. Omitting the Δt, we express this concept as:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} = \begin{bmatrix} -1 & 1 & . & . & . & . \\ . & -1 & 1 & . & . & . \\ . & . & -1 & 1 & . & . \\ . & . & . & -1 & 1 & . \\ . & . & . & . & -1 & 1 \\ . & . & . & . & . & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} \tag{3}$$

The filter 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 $(-1, 1)$. Therefore, mathematically, we can say that the adjoint of the time derivative operation is the negative time derivative. Likewise, in the Fourier domain, 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. The last row in equation (3) is optional. 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.

Equation (3) is implemented by the code in module `igrad1` that does the operator itself (the forward operator) and its adjoint.

api/c/igrad1.c

```
void   sf_igrad1_lop(bool adj, bool add,
                     int nx, int ny, float *xx, float *yy)
/*< linear operator >*/
{
    int i;

    sf_adjnull(adj,add,nx,ny,xx,yy);
    for (i=0; i < nx-1; i++) {
        if (adj) {
            xx[i+1] += yy[i];
            xx[i]   -= yy[i];
        } else {
            yy[i] += xx[i+1] - xx[i];
        }
    }
}
```

The adjoint code may seem strange. It might seem more natural to code the adjoint to be the negative of the operator itself; and then, make the special adjustments for the boundaries. The code given, however, is correct and requires no adjustments at the ends. To see why, notice for each value of `i`, the operator itself handles one row of equation (3), while for each `i`, the adjoint handles one column. That is why coding the adjoint in this way does not require any special work on the ends. The present method of coding reminds us that the adjoint of a sum of N terms is a collection of N assignments. Think of the meaning of $y_i = y_i + a_{i,j} x_j$ for any particular i and j. The adjoint simply accumulates that same value of $a_{i,j}$ going the other direction $x_j = x_j + a_{i,j} y_i$.

Figure 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

```
for (iy=0; iy < ny; iy++)
     igrad1_lop(adj, add, nx, nx, map[iy], ruf[iy]);
```

Transient convolution

The next operator we examine is convolution. It arises in many applications; and it could be derived in many ways. A basic derivation is from the multiplication of two polynomials, say $X(Z) = x_1 + x_2 Z + x_3 Z^2 + x_4 Z^3 + x_5 Z^4 + x_6 Z^5$ times $B(Z) = b_1 + b_2 Z + b_3 Z^2 + b_4 Z^3$.[1] Identifying the k-th power of Z in the product $Y(Z) = B(Z)X(Z)$ gives the k-th row of the convolution transformation (4).

$$\mathbf{y} \quad = \quad \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} \quad = \quad \begin{bmatrix} b_1 & 0 & 0 & 0 & 0 & 0 \\ b_2 & b_1 & 0 & 0 & 0 & 0 \\ b_3 & b_2 & b_1 & 0 & 0 & 0 \\ 0 & b_3 & b_2 & b_1 & 0 & 0 \\ 0 & 0 & b_3 & b_2 & b_1 & 0 \\ 0 & 0 & 0 & b_3 & b_2 & b_1 \\ 0 & 0 & 0 & 0 & b_3 & b_2 \\ 0 & 0 & 0 & 0 & 0 & b_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} \quad = \quad \mathbf{Bx} \tag{4}$$

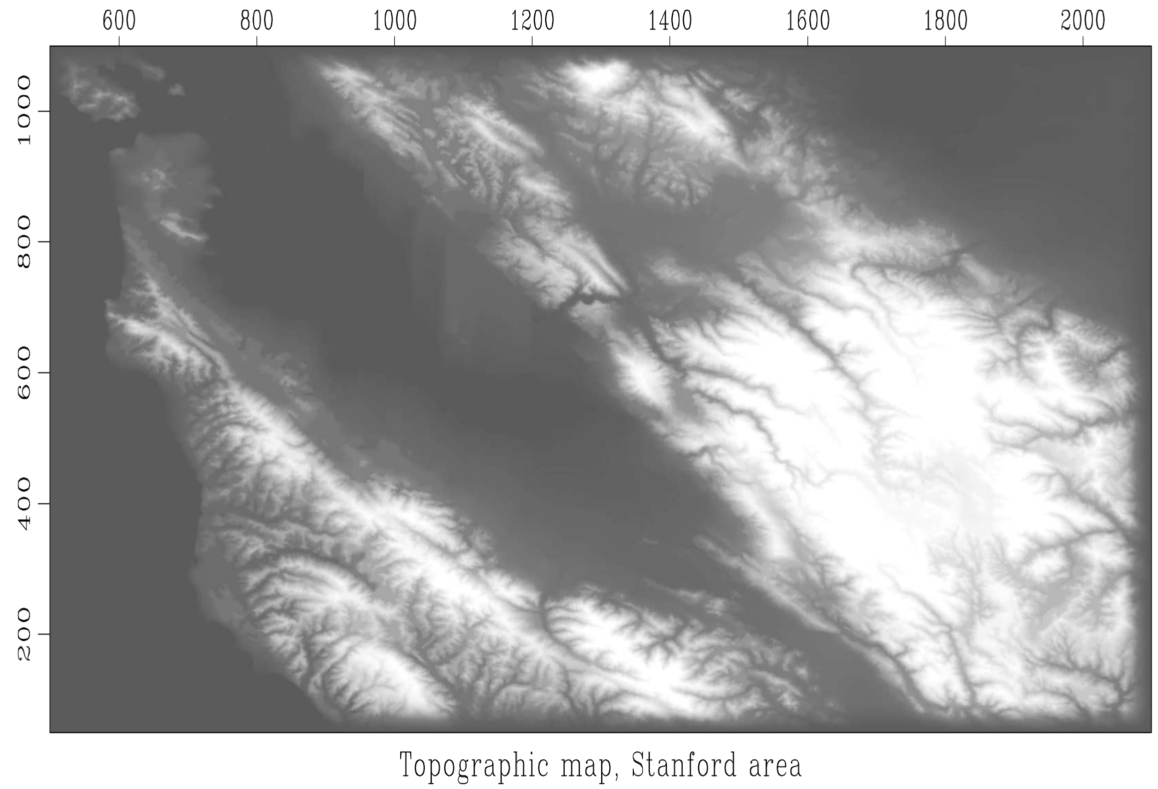
Notice that columns of equation (4) all contain the same "wavelet" but with different shifts. This signal is called the filter's impulse response.

Equation (4) could be rewritten as

$$\mathbf{y} \quad = \quad \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} \quad = \quad \begin{bmatrix} x_1 & 0 & 0 \\ x_2 & x_1 & 0 \\ x_3 & x_2 & x_1 \\ x_4 & x_3 & x_2 \\ x_5 & x_4 & x_3 \\ x_6 & x_5 & x_4 \\ 0 & x_6 & x_5 \\ 0 & 0 & x_6 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \quad = \quad \mathbf{Xb} \tag{5}$$

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,

[1] This book is more involved with matrices than with Fourier analysis. If it were more Fourier analysis, we would choose notation to begin subscripts from zero like this: $B(Z) = b_0 + b_1 Z + b_2 Z^2 + b_3 Z^3$.



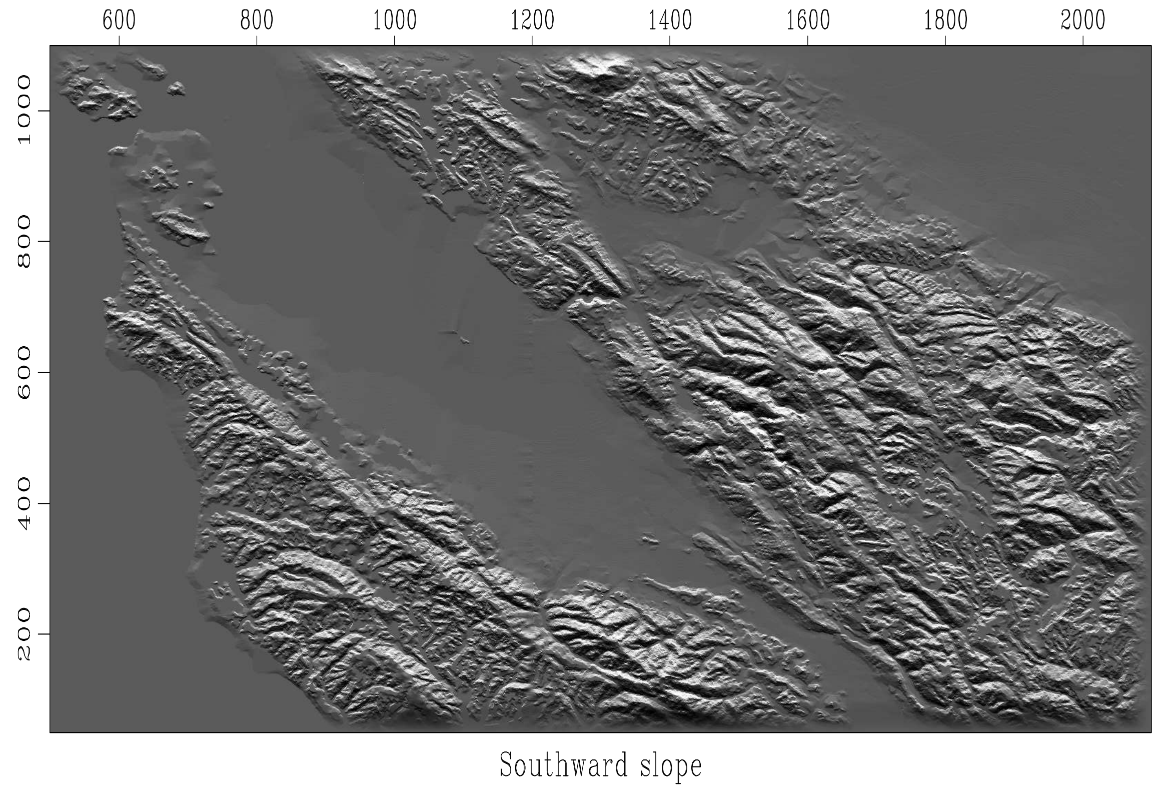


Figure 1: Topography near Stanford (top) southward slope (bottom).

we must solve for **b** and sometimes for **x**; therefore sometimes we use equation (5) and sometimes (4). Such solutions begin from the adjoints. The adjoint of equation (4) is

$$\begin{bmatrix} \hat{x}_1 \\ \hat{x}_2 \\ \hat{x}_3 \\ \hat{x}_4 \\ \hat{x}_5 \\ \hat{x}_6 \end{bmatrix} = \begin{bmatrix} b_1 & b_2 & b_3 & 0 & 0 & 0 & 0 & 0 \\ 0 & b_1 & b_2 & b_3 & 0 & 0 & 0 & 0 \\ 0 & 0 & b_1 & b_2 & b_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & b_1 & b_2 & b_3 & 0 & 0 \\ 0 & 0 & 0 & 0 & b_1 & b_2 & b_3 & 0 \\ 0 & 0 & 0 & 0 & 0 & b_1 & b_2 & b_3 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} \tag{6}$$

The adjoint ***crosscorrelates*** with the filter instead of convolving with it (because the filter is backward). Notice that each row in equation (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 (5) crosscorrelates a fixed portion of filter input across a variable portion of filter output.

$$\begin{bmatrix} \hat{b}_1 \\ \hat{b}_2 \\ \hat{b}_3 \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & 0 & 0 \\ 0 & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & 0 \\ 0 & 0 & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} \tag{7}$$

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

user/gee/tcai1.c

```
    for( b=0; b < nb; b++) {
        for( x=0; x < nx; x++) { y = x + b;
            if( adj) xx[x] += yy[y] * bb[b];
            else     yy[y] += xx[x] * bb[b];
        }
    }
```

The polynomials $X(Z)$, $B(Z)$, and $Y(Z)$ are called Z transforms. An important fact in real life (but not important here) is that the Z transforms are Fourier transforms in disguise. Each polynomial is a sum of terms, and the sum amounts to a Fourier sum when we take $Z = e^{i\omega\Delta t}$. The very expression $Y(Z) = B(Z)X(Z)$ says that a product in the frequency domain (Z has a numerical value) is a convolution in the time domain. Matrices and programs nearby are doing convolutions of coefficients.

user/gee/tcaf1.c

```
    for (b=0; b < nb; b++) {
        for (x=0; x < nx; x++) { y = x + b;
            if( adj) bb[b] += yy[y] * xx[x];
            else     yy[y] += bb[b] * xx[x];
        }
    }
```

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. Careful handling of ends is important in data in which 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 equations (4) and (5), the top two equations explicitly assume 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 equation (5) we get:

$$\begin{bmatrix} y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} = \begin{bmatrix} x_3 & x_2 & x_1 \\ x_4 & x_3 & x_2 \\ x_5 & x_4 & x_3 \\ x_6 & x_5 & x_4 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{8}$$

The adjoint is

$$\begin{bmatrix} \hat{b}_1 \\ \hat{b}_2 \\ \hat{b}_3 \end{bmatrix} = \begin{bmatrix} x_3 & x_4 & x_5 & x_6 \\ x_2 & x_3 & x_4 & x_5 \\ x_1 & x_2 & x_3 & x_4 \end{bmatrix} \begin{bmatrix} y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} \tag{9}$$

The difference between equations (9) and (7) is that here, the adjoint crosscorrelates a fixed portion of *output* across a variable portion of *input*; whereas, with (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; therefore, its location is specified by an additional parameter called its `lag`. The value of `lag` always used in this book is `lag=1`. For `lag=1` the module `icaf1` implements not equation (8) but equation (10):

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ x_3 & x_2 & x_1 \\ x_4 & x_3 & x_2 \\ x_5 & x_4 & x_3 \\ x_6 & x_5 & x_4 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{10}$$

user/gee/icaf1.c

```
    for( b=0; b < nb; b++) {
        for( y = SF_MAX(lag,b+1); y <= ny; y++) { x = y - b - 1;
            if( adj) bb[b] += yy[y-lag] * xx[x];
            else     yy[y-lag] += bb[b] * xx[x];
        }
    }
```

It may seem a little odd to put the required zeros at the beginning of the output, but filters are generally designed so the strongest coefficient is the first, namely `bb(1)`, so the alignment of input and output in equation (10) is the most common one.

The **end effect**s of the convolution modules are summarized in Figure 2.

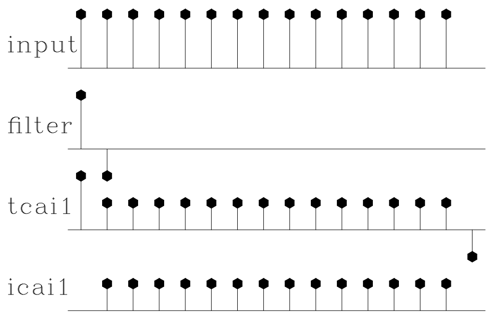


Figure 2: Example of convolution end-effects. From top to bottom: input; filter; output of `tcai1()`; output of `icaf1()` also with (`lag=1`).

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. Set a signal in a vector $\mathbf{x}$, and then to make a longer vector $\mathbf{y}$, append some zeros to $\mathbf{x}$. This zero padding can be regarded as the matrix multiplication

$$\mathbf{y} \quad = \quad \begin{bmatrix} \mathbf{I} \\ \mathbf{0} \end{bmatrix} \mathbf{x} \tag{11}$$

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:

$$\tilde{\mathbf{x}} \quad = \quad \begin{bmatrix} \mathbf{I} & \mathbf{0} \end{bmatrix} \mathbf{y} \tag{12}$$

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

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 user/gee/zpad1.c

```
    for (d=0; d < nd; d++) {
        p = d + (np-nd)/2;
        if (adj) data[d] += padd[p];
        else     padd[p] += data[d];
    }
```

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., $\mathbf{F}^{\mathrm{T}}$. The adjoint arises naturally whenever we consider energy. The statement that Fourier transforms conserve energy is $\mathbf{y}^{\mathrm{T}}\mathbf{y} = \mathbf{x}^{\mathrm{T}}\mathbf{x}$ where $\mathbf{y} = \mathbf{F}\mathbf{x}$. Substituting gives $\mathbf{F}^{\mathrm{T}}\,\mathbf{F} = \mathbf{I}$, 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

$$\text{Program} \quad = \quad \mathbf{F} \begin{bmatrix} \mathbf{I} \\ \mathbf{0} \end{bmatrix} \tag{13}$$

According to matrix algebra, the transpose of a product, say $\mathbf{AB} = \mathbf{C}$, is the product $\mathbf{C}^{\mathrm{T}} = \mathbf{B}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}$ in reverse order. Therefore, the adjoint routine is given by

$$\text{Program}^{\mathrm{T}} \quad = \quad \begin{bmatrix} \mathbf{I} & \mathbf{0} \end{bmatrix} \mathbf{F}^{\mathrm{T}} \tag{14}$$

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 the collapse of all precise analysis.

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

```
for (iz=0; iz < nz; nz++) {   z = z0 + iz * dz;
for (i3=0, i3 < n3; i3++) {  x3 = o3 + i3 * 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. Solve for `iz`; add one half; and round down to the nearest integer. The familiar computer idioms are:

```
iz = 0.5 + ( z - z0) / dz;
i3 = 0.5 + (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 C function `round()`.

Data-push binning

A most basic data modeling operation is to copy a number from an (x, y)-location on a map to a 1-D survey data track $d(s)$, where s is a coordinate running along a survey track. This copying proceeds for all s. The track could be along either a straight, curved, or arbitrary line. Let the coordinate s take on integral values. Along with the elements $d(s)$ are the coordinates $(x(s), y(s))$ where on the map the data value $d(s)$ would be recorded.

Code for the operator is shown in module `bin2`. To invert this data modeling operation,

user/gee/bin2.c

```
    for (id=0; id < nd; id++) {
        i1 = 0.5 + (xy[0][id]-o1)/d1;
        i2 = 0.5 + (xy[1][id]-o2)/d2;
        if (0<=i1 && i1<m1 &&
            0<=i2 && i2<m2) {
            im = i1+i2*m1;
            if (adj) mm[im] += dd[id];
            else     dd[id] += mm[im];
        }
    }
```

going from $d(s)$ to $(x(s), y(s))$ requires more than the adjoint operator because each bin ends up with a different number of data values. After the adjoint operation is performed, the inverse operator needs to divide the bin sum by the number of data values that landed in the bin. It is this inversion operator that is generally called “binning” (although we will use that name here for the modeling operator). 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 $\mathbf{B}$ 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:

$$\mathbf{m} \quad = \quad \mathbf{diag}(\mathbf{B}^{\mathrm{T}}\mathbf{1})^{-1}\mathbf{B}^{\mathrm{T}}\mathbf{d} \tag{15}$$

Empty bins, of course, leave us a problem because we dare not divide by the zero sum they contain. We address this zero divide issue in Chapter **??**. In Figure 3, the empty bins contain zero values.

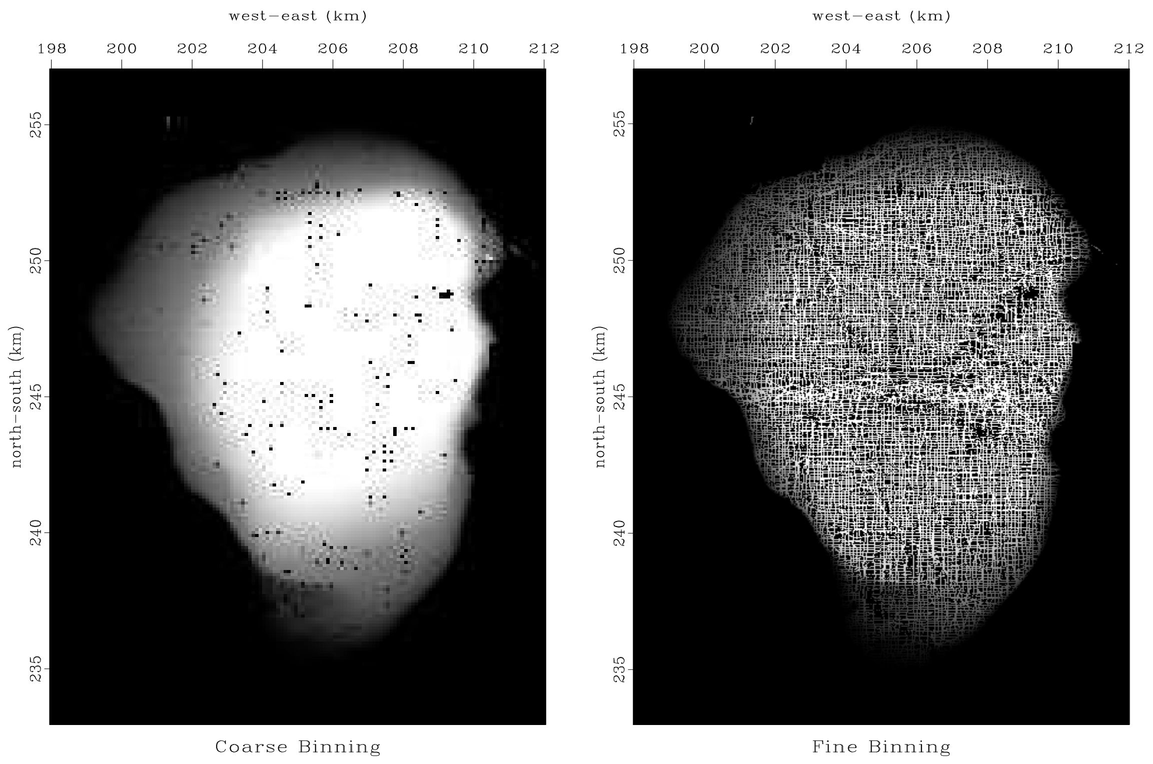
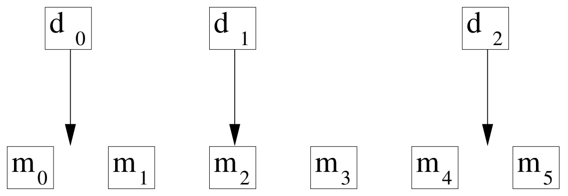


Figure 3: Binned depths of the Sea of Galilee.

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 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 4, we have the operator in equation (16).

Figure 4: Uniformly sampled model space and irregularly sampled data space corresponding to equation (16).



$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix} \approx \begin{bmatrix} .7 & .3 & . & . & . & . \\ . & . & 1 & . & . & . \\ . & . & . & . & .5 & .5 \end{bmatrix} \begin{bmatrix} m_0 \\ m_1 \\ m_2 \\ m_3 \\ m_4 \\ m_5 \end{bmatrix} \tag{16}$$

The two weights in each row sum to unity. If a binning operator was 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 the matrix is crudely a diagonal matrix like equation (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 ?? and ??, we build inverse operators.

user/gee/lint1.c

```
    for (id=0; id < nd; id++) {
        f = (coord[id]-o1)/d1;
        im=floorf(f);
        if (0 <= im && im < nm-1) {
            fx=f-im;
            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];
            }
        }
    }
```

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, in which 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 in the following, 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. Either a copy or an add is done, depending on the `add` parameter. It is either a spray, a sum, or a copy, according to the specified axis lengths.

user/gee/spraysum.c

```
    for (i3=0; i3 < SF_MAX(n3,m3); i3++) {
        x = SF_MIN(i3,n3-1);
        y = SF_MIN(i3,m3-1);
        for (i2=0; i2 < SF_MAX(n2,m2); i2++) {
            x = x*n2 + SF_MIN(i2,n2-1);
            y = y*m2 + SF_MIN(i2,m2-1);
            for (i1=0; i1 < SF_MAX(n1,m1); i1++) {
                x = x*n1 + SF_MIN(i1,n1-1);
                y = y*m1 + SF_MIN(i1,m1-1);

                if( adj)  xx[x]  +=  yy[y];
                else      yy[y]  +=  xx[x];
            }
        }
    }
```

Causal and leaky integration

Causal integration is defined as:

$$y(t) \quad = \quad \int_{-\infty}^{t} x(\tau)\; d\tau \tag{17}$$

Leaky integration is defined as:

$$y(t) \quad = \quad \int_{0}^{\infty} x(t-\tau)\; e^{-\alpha\tau}\; d\tau \tag{18}$$

As $\alpha \to 0$, leaky integration becomes causal integration. The word "leaky" comes from electrical circuit theory in which 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 (19) represents causal integration for $\rho = 1$ and leaky integration for $0 < \rho < 1$.

$$\mathbf{y} \quad = \quad \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} \quad = \quad \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \rho & 1 & 0 & 0 & 0 & 0 & 0 \\ \rho^2 & \rho & 1 & 0 & 0 & 0 & 0 \\ \rho^3 & \rho^2 & \rho & 1 & 0 & 0 & 0 \\ \rho^4 & \rho^3 & \rho^2 & \rho & 1 & 0 & 0 \\ \rho^5 & \rho^4 & \rho^3 & \rho^2 & \rho & 1 & 0 \\ \rho^6 & \rho^5 & \rho^4 & \rho^3 & \rho^2 & \rho & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} \quad = \quad \mathbf{Cx} \qquad (19)$$

(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 (19) is more efficiently thought of as the recursion

$$y_t \quad = \quad \rho\, y_{t-1} + x_t \qquad t \text{ increasing} \qquad (20)$$

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

When it comes time to think about the adjoint, however, it is easier to think of equation (19) than of equation (20). Let the matrix of equation (19) be called $\mathbf{C}$. Transposing to get $\mathbf{C}^{\mathrm{T}}$ and applying it to $\mathbf{y}$ gives us something back in the space of $\mathbf{x}$, namely $\tilde{\mathbf{x}} = \mathbf{C}^{\mathrm{T}}\mathbf{y}$. From it we see that the adjoint calculation, if done recursively, needs to be done backward, as in:

$$\tilde{x}_{t-1} \quad = \quad \rho\tilde{x}_t + y_{t-1} \qquad t \text{ decreasing} \qquad (21)$$

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 $\tilde{\mathbf{x}} = \mathbf{C}^{\mathrm{T}}\mathbf{y}$ and take care to get the ends correct. Figure 5 illustrates the program for $\rho = 1$.

Later, we consider equations to march wavefields up toward the Earth surface, a layer at a time, an operator for each layer. Then, the adjoint starts from the Earth surface and marchs down, a layer at a time, into the Earth.

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 (22), where $(a_3, a_4, \cdots) = 0$. There is a huge literature on finite-difference solutions of ordinary differential equations that api/c/causint.c

```
    t = 0.;
    if (adj) {
        for (i=nx-1; i >= 0; i--) {
            t += yy[i];
            xx[i] += t;
        }
    } else {
        for (i=0; i <= nx-1; i++) {
            t += xx[i];
            yy[i] += t;
        }
    }
```

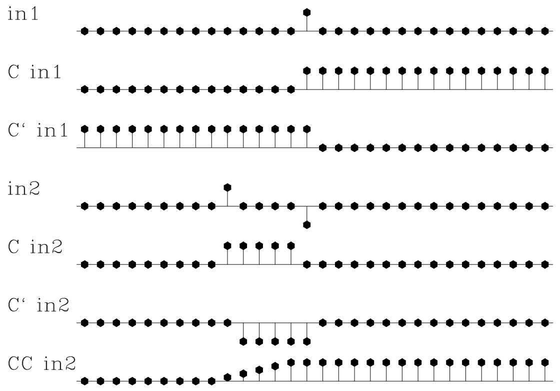


Figure 5: `in1` is an input pulse. `C in1` is its causal integral. `C' in1` is the anticausal integral of the pulse. A separated doublet is `in2`. Its causal integration is a box, and its anticausal integration is a negative box. `CC in2` is the double causal integral of `in2`. How can an equilateral triangle be built?

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 in equation (4), but put the top square of the matrix on the other side of the equation so our transformation can be called one of inversion or backsubstitution. To link up with applications in later chapters, I specialize to put 1s on the main diagonal and insert some bands of zeros.

$$\mathbf{Ay} \quad = \quad \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ a_1 & 1 & 0 & 0 & 0 & 0 & 0 \\ a_2 & a_1 & 1 & 0 & 0 & 0 & 0 \\ 0 & a_2 & a_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & a_2 & a_1 & 1 & 0 & 0 \\ a_5 & 0 & 0 & a_2 & a_1 & 1 & 0 \\ 0 & a_5 & 0 & 0 & a_2 & a_1 & 1 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} \quad = \quad \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} \quad = \quad \mathbf{x} \tag{22}$$

Algebraically, this operator goes under the various names, "backsolving," "polynomial division," and "deconvolution." The leaky integration transformation in equation (19) is a simple example of backsolving when $a_1 = -\rho$ and $a_2 = a_5 = 0$. To confirm, you need to verify that the matrices in equations (22) and (19) are mutually inverse.

A typical row in equation (22) says:

$$x_t \quad = \quad y_t \; + \; \sum_{\tau>0} a_\tau \; y_{t-\tau} \tag{23}$$

Change the signs of all terms in equation (23), and move some terms to the opposite side:

$$y_t \quad = \quad x_t \; - \; \sum_{\tau>0} a_\tau \; y_{t-\tau} \tag{24}$$

Equation (24) is a recursion to find y_t from the values of y at earlier times.

In the same way that equation (4) can be interpreted as $Y(Z) = B(Z)X(Z)$, equation (22) can be interpreted as $A(Z)Y(Z) = X(Z)$, which amounts to $Y(Z) = X(Z)/A(Z)$. Thus, convolution is amounts to polynomial multiplication while the backsubstitution we are doing here is called "deconvolution," and it amounts 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 (22) 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 equation (22) is `recfilt`.

We may wonder why the adjoint of $\mathbf{Ay} = \mathbf{x}$ actually is $\mathbf{A}^{\mathrm{T}}\hat{\mathbf{x}} = \mathbf{y}$. With the well-known fact that the inverse of a transpose is the transpose of the inverse we have:

$$\begin{aligned} \mathbf{y} &= \mathbf{A}^{-1}\mathbf{x} && (25) \\ \hat{\mathbf{x}} &= (\mathbf{A}^{-1})^{\mathrm{T}}\mathbf{y} && (26) \\ \hat{\mathbf{x}} &= (\mathbf{A}^{\mathrm{T}})^{-1}\mathbf{y} && (27) \\ \mathbf{A}^{\mathrm{T}}\hat{\mathbf{x}} &= \mathbf{y} && (28) \end{aligned}$$

api/c/recfilt.c

```
    for (ix=0; ix < nx; ix++) {
        tt[ix] = 0.;
    }

    if (adj) {
        for (ix = nx-1; ix >= 0; ix--) {
            tt[ix] = yy[ix];
            for (ia = 0; ia < SF_MIN(na,ny-ix-1); ia++) {
                iy = ix + ia + 1;
                tt[ix] -= aa[ia] * tt[iy];
            }
        }
        for (ix=0; ix < nx; ix++) {
            xx[ix] += tt[ix];
        }
    } else {
        for (iy = 0; iy < ny; iy++) {
            tt[iy] = xx[iy];
            for (ia = 0; ia < SF_MIN(na,iy); ia++) {
                ix = iy - ia - 1;
                tt[iy] -= aa[ia] * tt[ix];
            }
        }
        for (iy=0; iy < ny; iy++) {
            yy[iy] += tt[iy];
        }
    }
```

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 noise is 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 a causal low-cut filter (high-pass filter) by two stages that can be done in either order.

1. Apply a time derivative, actually a finite difference, convolving the data with $(1, -1)$.
2. Do a leaky integration dividing by $1 - \rho Z$ where numerically, ρ is slightly less than unity.

The convolution with $(1, -1)$ ensures the zero frequency is removed. The leaky integration almost undoes the differentiation but cannot restore the zero frequency. Adjusting the numerical value of ρ has interesting effects in the time domain and in the frequency domain. Convolving the finite difference $(1, -1)$ with the leaky integration $(1, \rho, \rho^2, \rho^3, \rho^4, \cdots)$ gives the result:

$$\begin{array}{rll} & (1, & \rho, \rho^2, \rho^3, \rho^4, \cdots) \\ - & (0, & 1, \rho, \rho^2, \rho^3, \cdots). \end{array}$$

Rearranging, it becomes:

$$\begin{array}{rll} & (1, & 0, 0, 0, 0, \cdots) \ + \\ (\rho - 1) & (0, & 1, \rho, \rho^2, \rho^3, \cdots). \end{array}$$

Because ρ is a tiny bit less than one, $(1-\rho)$ is a small number. Thus, our filter is an impulse followed by the negative of a weak decaying exponential ρ^t. If you prefer a time-symmetric (phaseless) filter, you could follow this one by its time reverse.

Roughly speaking, the cut-off frequency of the filter corresponds to matching one wavelength to the exponential decay time. More formally, the Fourier domain representation of this filter is $H(Z) = (1-Z)/(1-\rho Z)$, where Z is the unit-delay operator is $Z = e^{i\omega\Delta t}$, and where ω is the frequency. The spectral response of the filter is $|H(\omega)|$. Were we to plot this function, we would see it is nearly 1 everywhere except in a small region near $\omega = 0$ where it becomes tiny. Figure 6 compares a low-cut filter to a finite difference.

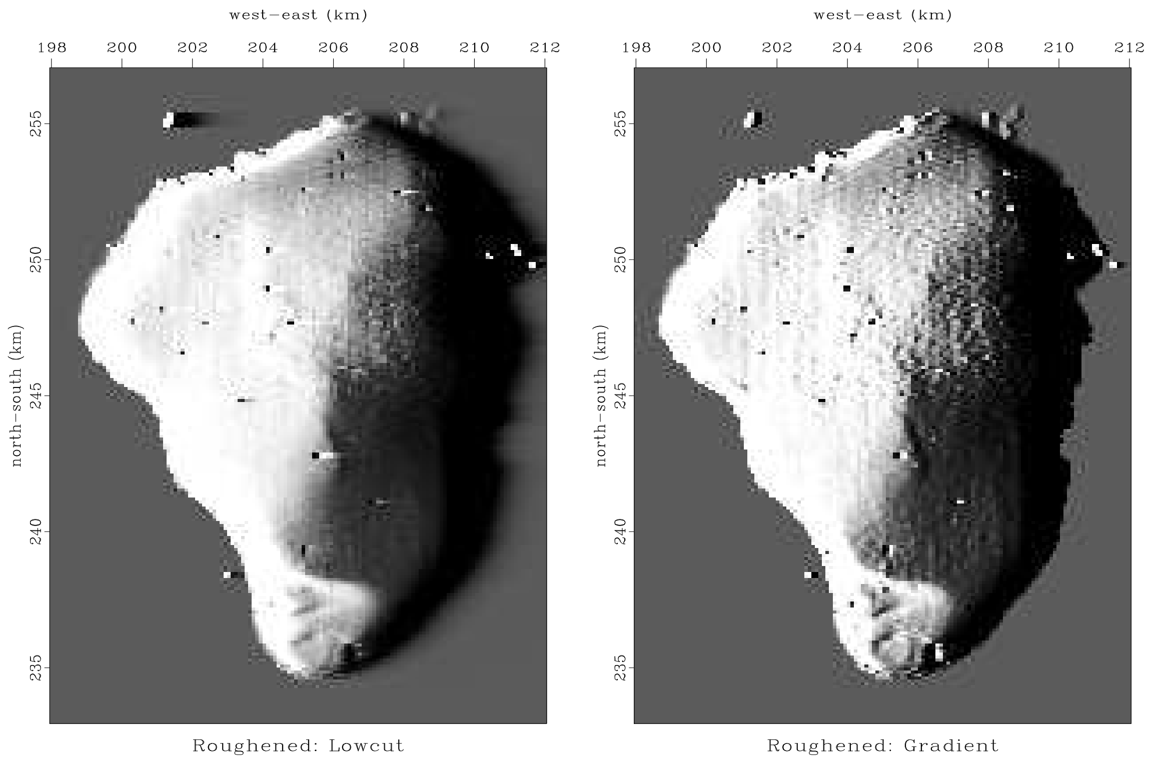


Figure 6: The depth of the Sea of Galilee after roughening. On the left, the smoothing is done by low-cut filtering on the horizontal axis. On the right it is a finite difference. We see which is which because of a few scattered impulses (navigation failure) outside the lake. Both results solve the problem of Figure 3 that it is too smooth to see interesting features.

Smoothing with box and triangle

Simple "**smoothing**" is a common application of filtering. A smoothing filter is one with all positive coefficients. On the time axis, smoothing is often done with a single-pole damped exponential function. On space axes, however, people generally prefer a symmetrical function. We begin with rectangle and triangle functions. When the function width is chosen to be long, then the computation time can be large, but recursion can shorten it immensely.

The inverse of any polynomial reverberates forever, although it might drop off fast enough for any practical need. On the other hand, a rational filter can suddenly drop to zero and stay there. Let us look at a popular rational filter, the rectangle or "**box car**":

$$\frac{1-Z^5}{1-Z} \quad = \quad 1+Z+Z^2+Z^3+Z^4 \tag{29}$$

The filter of equation (29) gives a moving average under a *rectangular* window. It is a basic smoothing filter. A clever way to apply it is to move the rectangle by adding a new value at one end while dropping an old value from the other end. This approach is formalized by the polynomial division algorithm, which can be simplified, because so many coefficients are either one or zero. To find the recursion associated with $Y(Z) = X(Z)(1-Z^5)/(1-Z)$, we identify the coefficient of Z^t in $(1-Z)Y(Z) = X(Z)(1-Z^5)$. The result is:

$$y_t \quad = \quad y_{t-1} + x_t - x_{t-5}. \tag{30}$$

This approach boils down to the program which is so fast it is almost free! Its last line

api/c/triangle.c

```
    tmp2 = tmp + 2*nb;

    wt = 1.0/(2*nb-1);
    for (i=0; i < nx; i++) {
        x[o+i*d] = (tmp[i+1] - tmp2[i])*wt;
    }
```

scales the output by dividing by the rectangle length. With this scaling, the zero-frequency component of the input is unchanged, while other frequencies are suppressed.

Triangle smoothing is rectangle smoothing done twice. For a mathematical description of the triangle filter, we simply square equation (29). Convolving a rectangle function with itself many times yields a result that mathematically tends toward a **Gaussian** function. Despite the sharp corner on the top of the triangle function, it has a shape remarkably similar to a Gaussian. Convolve a triangle with itself and you see a very nice approximation to a Gaussian (the central limit theorem).

With filtering, **end effect**s can be a nuisance, especially on space axes. Filtering increases the length of the data, but people generally want to keep input and output the same length (for various practical reasons), especially on a space axis. Suppose the five-point signal $(1,1,1,1,1)$ is smoothed using the `boxconv()` program with the three-point smoothing filter $(1,1,1)/3$. The output is $5+3-1$ points long, namely, $(1,2,3,3,3,2,1)/3$.

We could simply abandon the points off the ends, but I like to **fold** them back in, getting instead $(1+2, 3, 3, 3, 1+2)$. An advantage of the folding is that a constant-valued signal is unchanged by the smoothing. Folding is desirable because a smoothing filter is a low-pass filter that naturally should pass the lowest frequency $\omega = 0$ without distortion. The result is like a wave reflected by a **zero-slope** end condition. Impulses are smoothed into triangles except near the boundaries. What happens near the boundaries is shown in Figure 7. At

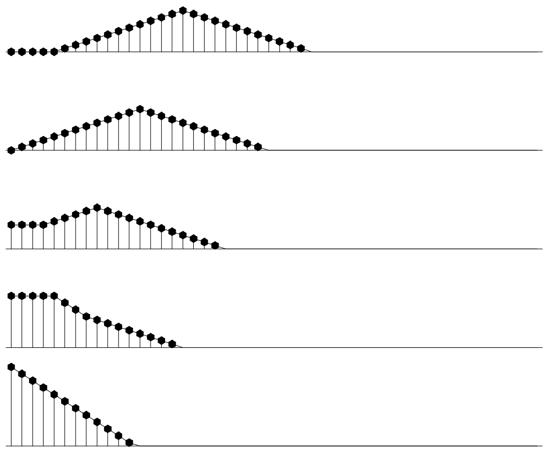


Figure 7: Edge effects when smoothing an impulse with a triangle function. Inputs are spikes at various distances from the edge.

the side boundary is only half a triangle, but it is twice as tall.

Why this end treatment? Consider a survey of water depth in an area of the deep ocean. All the depths are strongly positive with interesting but small variations on them. Ordinarily we can enhance high-frequency fluctuations by one minus a low-pass filter, say $H = 1 - L$. If this subtraction is to work, it is important that the L truly cancel the 1 near zero frequency.

Figure 7 was derived from the routine `triangle()`.

api/c/triangle.c

```
        tmp1 = tmp + nb;
        tmp2 = tmp + 2*nb;

        wt = 1.0/(nb*nb);
        for (i=0; i < nx; i++) {
            x[o+i*d] = (2.*tmp1[i] - tmp[i] - tmp2[i])*wt;
        }
```

Nearest-neighbor normal moveout (NMO)

Normal-moveout correction 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. The whole process is called "**NMO stack**ing." The adjoint to this operation is to begin from a model that ideally is the zero-offset trace, and spray this model to all offsets. From a matrix viewpoint, stacking is like a *row* vector of NMO operators, and modeling is like a *column*. An example is shown in Figure 8.

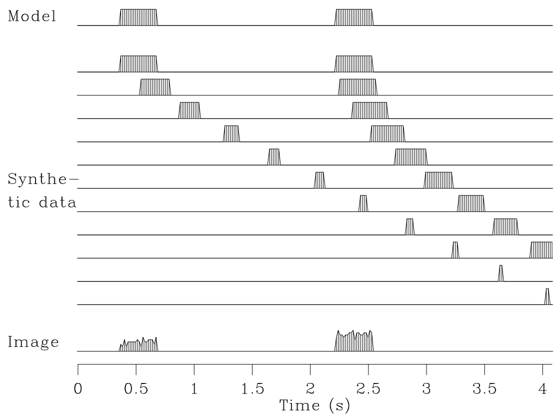


Figure 8: Hypothetical model, synthetic data, and model image.

A module that does reverse moveout is `hypotenusei`. Given a zero-offset trace, it makes another at nonzero offset. The adjoint does the usual normal moveout correction. My 1992 textbook*PVI* (Earth Soundings Analysis : Processing Versus Inversion) illustrates many additional features of NMO. 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.

Coding chains and arrays

With a collection of operators, we can build more elaborate operators. An amazing thing about a matrix is that its elements may be matrices. A row is a matrix containing side-by-side matrices. Rows are done by subroutine `row0` also in module `smallchain3`. An operator product $\mathbf{A} = \mathbf{BC}$ is represented in the subroutine `chain2( op1, op2, ...)`. As you read these codes, please remember the output is the last argument only when the output is $\mathbf{d}$. When the output is $\mathbf{m}$, the output is the second from last.

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 push concepts to the limit.

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 with components packed with user/gee/hypotenusei.c

```
void hypotenusei_set(float t0 /* time origin */,
                     float dt /* time sampling */,
                     float xs /* offset times slowness */)
/*< set up >*/
{
    int it;
    float t, z2;

    for (it=0; it < nt; it++) {
        t = t0 + dt*it;
        z2 =  t * t - xs * xs;
        iz[it] = ( z2 >= 0.)? 0.5 + (sqrtf(z2) - t0) /dt: -1;
    }
}

void hypotenusei_lop(bool adj, bool add,
                     int n1, int n2, float *zz, float *tt)
/*< linear operator >*/
{
    int  it;

    sf_adjnull(adj,add,n1,n2,zz,tt);

    for (it=0; it < nt; it++) {
        if (iz[it] < 0) continue;

        if (adj)
            zz[iz[it]] +=  tt[it];
        else
            tt[it] +=  zz[iz[it]];
    }
}
```

user/gee/imospray.c

```
void imospray_lop(bool adj, bool add, int n1, int n2,
                  float *stack, float *gather)
/*< linear operator >*/
{
    int ix;
    float x;

    sf_adjnull(adj,add,n1,n2,stack,gather);

    for (ix=0; ix < nx; ix++) {
        x = x0 + dx*ix;

        hypotenusei_set (t0, dt, x);
        hypotenusei_lop (adj, true, nt, nt, stack, gather+ix*nt);
    }
}
```

api/c/chain.c

```
void sf_chain( sf_operator oper1     /* outer operator */,
               sf_operator oper2     /* inner operator */,
               bool adj              /* adjoint flag */,
               bool add              /* addition flag */,
               int nm                /* model size */,
               int nd                /* data size */,
               int nt                /* intermediate size */,
               /*@out@*/ float* mod  /* [nm] model */,
               /*@out@*/ float* dat  /* [nd] data */,
               float* tmp            /* [nt] intermediate */)
/*< Chains two operators, computing oper1{oper2{mod}}
  or its adjoint. The tmp array is used for temporary storage. >*/
{
    if (adj) {
        oper1 (true, false, nt, nd, tmp, dat);
        oper2 (true, add, nm, nt, mod, tmp);
    } else {
        oper2 (false, false, nm, nt, mod, tmp);
        oper1 (false, add, nt, nd, tmp, dat);
    }
}
```

api/c/chain.c

```
void sf_array( sf_operator oper1     /* top operator */,
               sf_operator oper2     /* bottom operator */,
               bool adj              /* adjoint flag */,
               bool add              /* addition flag */,
               int nm                /* model size */,
               int nd1               /* top data size */,
               int nd2               /* bottom data size */,
               /*@out@*/ float* mod  /* [nm] model */,
               /*@out@*/ float* dat1 /* [nd1] top data */,
               /*@out@*/ float* dat2 /* [nd2] bottom data */)
/*< Constructs an array of two operators,
  computing {oper1{mod},oper2{mod}} or its adjoint. >*/
{
    if (adj) {
        oper1 (true, add,  nm, nd1, mod, dat1);
        oper2 (true, true, nm, nd2, mod, dat2);
    } else {
        oper1 (false, add, nm, nd1, mod, dat1);
        oper2 (false, add, nm, nd2, mod, dat2);
    }
}
```

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 mixed sets of 1-D, 2-D, and 3-D objects. One space that is a set of three cubes is the Earth's magnetic field, which has three components, each component being a function of three-dimensional physical space. (The 3-D *physical space* we live in is not the abstract ***vector space*** of models and data so abundant in this book. In this book the word "space" without an adjective means the " vector space.") Other common spaces are physical space and Fourier 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 $\mathbf{d}$. A notation that does transpose and complex conjugate at the same time is $\mathbf{d}^{\mathrm{T}}\,\mathbf{d}$. In theoretical work, the "size of a vector" means the vector's norm. In computational work the "size 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; therefore, 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 $\mathbf{d}^{\mathrm{T}}\mathbf{W}\mathbf{d}$. Standard notation for norms uses a double absolute value, where $||\mathbf{d}|| = \mathbf{d}^{\mathrm{T}}\mathbf{W}\mathbf{d}$. A central concept with norms is the triangle inequality, $||\mathbf{d}_1 + \mathbf{d}_2|| \leq ||\mathbf{d}_1|| + ||\mathbf{d}_2||$ a proof you might recall (or reproduce with the use of dot products).

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 if the program for the adjoint of an operator is precisely consistent with the operator. 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 operator and its adjoint are two routines. The first amounts to the matrix multiplication $\mathbf{Fm}$. The adjoint routine computes $\mathbf{F}^{\mathrm{T}}\mathbf{d}$, where $\mathbf{F}^{\mathrm{T}}$ is the **conjugate-transpose** matrix. In later chapters we solve huge sets of simultaneous equations in which both routines are required. If the pair of routines are inconsistent, we may be doomed from the start. The dot-product test is a simple test for verifying that the two routines are adjoint to each other.

I will tell you first what the dot-product test is, and then explain how it works. Take a model space vector $\mathbf{m}$ filled with random numbers, and likewise a data space vector $\mathbf{d}$ filled with random numbers. Use your forward modeling code to compute:

$$\begin{aligned} \mathbf{m} &\Leftarrow \text{random} && (31) \\ \mathbf{d} &\Leftarrow \text{random} && (32) \\ \hat{\mathbf{d}} &= \mathbf{Fm} && (33) \\ \hat{\mathbf{m}} &= \mathbf{F}^{\mathrm{T}}\mathbf{d} && (34) \end{aligned}$$

You should find these two inner products are equal:

$$\hat{\mathbf{m}} \cdot \mathbf{m} = \hat{\mathbf{d}} \cdot \mathbf{d} \qquad (35)$$

If they are, it means what you coded for $\mathbf{F}^{\mathrm{T}}$ is indeed the adjoint of $\mathbf{F}$. There is a glib way of saying why this must be so:

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

$$\mathbf{d}^{\mathrm{T}}(\mathbf{F}\mathbf{m}) = (\mathbf{F}^{\mathrm{T}}\mathbf{d})^{\mathrm{T}}\mathbf{m} \tag{37}$$

This glib way is more concrete with explicit summation. We may express $\sum_i \sum_j d_i F_{ij} m_j$ in two different ways.

$$\sum_i d_i (\sum_j F_{ij} m_m) = \sum_j (\sum_i d_i F_{ij}) m_j \tag{38}$$

$$= \sum_j (\sum_i F_{ij} d_i) m_j \tag{39}$$

$$\mathbf{d}^{\mathrm{T}} \cdot (\mathbf{F}\mathbf{m}) = (\mathbf{F}^{\mathrm{T}}\mathbf{d}) \cdot \mathbf{m} \tag{40}$$

$$\mathbf{d}^{\mathrm{T}} \cdot \hat{\mathbf{d}} = \hat{\mathbf{m}} \cdot \mathbf{m} \tag{41}$$

Should $\mathbf{F}$ contain complex numbers, the dot-product test is a comparison for both real parts and for imaginary parts.

The program for applying the dot product test is `dottest` on this page. The C 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 `dottest` 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, the second time with `add = true` to test if the operator can be used properly for accumulating results, for example. $\mathbf{d} \leftarrow \mathbf{d} + \mathbf{F}\mathbf{m}$.

api/c/dottest.c

```
    /* < L m1, d2 > = < m1, L* d2 > (w/o add) */
    oper(false, false, nm, nd, mod1, dat1);
    dot1[0] = cblas_dsdot( nd, dat1, 1, dat2, 1);

    oper(true, false, nm, nd, mod2, dat2);
    dot1[1] = cblas_dsdot( nm, mod1, 1, mod2, 1);

    /* < L m1, d2 > = < m1, L* d2 > (w/  add) */
    oper(false, true, nm, nd, mod1, dat1);
    dot2[0] = cblas_dsdot( nd, dat1, 1, dat2, 1);

    oper(true, true, nm, nd, mod2, dat2);
    dot2[1] = cblas_dsdot( nm, mod1, 1, mod2, 1);
```

I ran the dot product test on many operators and was surprised and delighted to find that for small operators it is generally satisfied to an accuracy near the computing precision. For large operators, precision can become and issue. Every time I encountered a relative discrepancy of 10^{-5} or more on a small operator (small data and model spaces), I was later able to uncover a conceptual or programming error. Naturally, when I run dot-product tests, I scale the implied matrix to a small size both 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 matched by like 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. Therefore, the dot-product test $\hat{\mathbf{m}} \cdot \mathbf{m} = \hat{\mathbf{d}} \cdot \mathbf{d}$ 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.

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 offers a program for converting linear operator programs into their adjoints. Actually, it does even more. He says:[2]

> Given a Fortran routine (or collection of routines) for a function, TAMC produces Fortran routines for the computation of the derivatives of this function. The derivatives are computed in the reverse mode (adjoint model) or in the forward mode (tangent-linear model). In both modes Jacobian-Matrix products can be computed.

The word "adjoint"

In mathematics, the word "**adjoint**" has two meanings. One, the so-called **Hilbert adjoint**, is generally found in physics and engineering and it is the one used in this book. In linear algebra there is a different matrix, called the **adjugate** matrix. It is a matrix with elements that 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 in which 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, "conjugate transpose" 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.

[2]`http://www.autodiff.com/tamc/`

Inverse operator

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

$$\mathbf{d}_{\text{obs}} \quad \approx \quad \mathbf{d}_{\text{model}} \quad = \quad \mathbf{Fm} \tag{42}$$

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

$$\hat{\mathbf{m}}_1 \quad = \quad (\mathbf{F}^{\text{T}}\mathbf{F})^{-1}\mathbf{F}^{\text{T}}\mathbf{d} \tag{43}$$

$$\hat{\mathbf{m}}_2 \quad = \quad \mathbf{F}^{\text{T}}(\mathbf{F}\mathbf{F}^{\text{T}})^{-1}\mathbf{d} \tag{44}$$

Some reasons for the literature being huge are the many questions about the existence, quality, and cost of the inverse operators. Let us quickly see why these two solutions are reasonable. Inserting equation (42) into equation (43), and inserting equation (44) into equation (42), we get the reasonable statements:

$$\hat{\mathbf{m}}_1 \quad = \quad (\mathbf{F}^{\text{T}}\mathbf{F})^{-1}(\mathbf{F}^{\text{T}}\mathbf{F})\mathbf{m} \quad = \quad \mathbf{m} \tag{45}$$

$$\hat{\mathbf{d}}_{\text{model}} \quad = \quad (\mathbf{F}\mathbf{F}^{\text{T}})(\mathbf{F}\mathbf{F}^{\text{T}})^{-1}\mathbf{d} \quad = \quad \mathbf{d} \tag{46}$$

Equation (45) says the estimate $\hat{\mathbf{m}}_1$ gives the correct model $\mathbf{m}$ if you start from the modeled data. Equation (46) says the model estimate $\hat{\mathbf{m}}_2$ gives the modeled data if we derive $\hat{\mathbf{m}}_2$ from the modeled data. Both these statements are delightful. Now, let us return to the problem of the inverse matrices.

Normally, 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 $\mathbf{F}^{\text{T}}\mathbf{F}$ nor $\mathbf{F}\mathbf{F}^{\text{T}}$ is invertible. This difficulty is solved by "**damping**" as we see in later chapters. If it happens that $\mathbf{F}\mathbf{F}^{\text{T}}$ or $\mathbf{F}^{\text{T}}\mathbf{F}$ equals $\mathbf{I}$ (unitary operator), then the adjoint operator $\mathbf{F}^{\text{T}}$ is the inverse $\mathbf{F}^{-1}$ by either equation (43) or (44).

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. The iterative methods we learn here for giant problems are also excellent for smaller problems; therefore we rarely here speak of inverse matrices or worry much if neither $\mathbf{F}\mathbf{F}^{\text{T}}$ nor $\mathbf{F}^{\text{T}}\mathbf{F}$ is an identity.

EXERCISES:

1 Consider the matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & \rho & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{47}$$

and others like it with ρ in other locations. Show what combination of these matrices will represent the leaky integration matrix in equation (19). What is the adjoint?

2 Modify the calculation in Figure 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 $\mathbf{C}^{\mathrm{T}}\mathbf{C}$ or $\mathbf{C}\mathbf{C}^{\mathrm{T}}$.

4 Modify `leakint` by changing the diagonal to contain 1/2 instead of 1. Notice how time alignment changes in Figure 5.

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