Chapter 2

Discrete Fourier transform

Happily, Fourier sums are exactly invertible: given the output, the input can be quickly found. Because signals can be transformed to the frequency domain, manipulated there, and then returned to the time domain, convolution and correlation can be done faster. Time derivatives can also be computed with more accuracy in the frequency domain than in the time domain. Signals can be shifted a fraction of the time sample, and they can be shifted back again exactly. In this chapter we will see how many operations we associate with the time domain can often be done better in the frequency domain. We will also examine some two-dimensional Fourier transforms.

2.1 FT AS AN INVERTIBLE MATRIX

A **Fourier sum** may be written

$$B(\omega) \quad = \quad \sum_t b_t \, e^{i\omega t} \quad = \quad \sum_t b_t \, Z^t \tag{2.1}$$

where the complex value Z is related to the real frequency ω by $Z = e^{i\omega}$. This Fourier sum is a way of building a continuous function of ω from discrete signal values b_t in the time domain. In this chapter we will study the computational tricks associated with specifying both time and frequency domains by a set of points. Begin with an example of a signal that is nonzero at four successive instants, (b_0, b_1, b_2, b_3). The transform is

$$B(\omega) \quad = \quad b_0 + b_1 Z + b_2 Z^2 + b_3 Z^3 \tag{2.2}$$

The evaluation of this polynomial can be organized as a matrix times a vector, such as

$$\begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix} \quad = \quad \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & W & W^2 & W^3 \\ 1 & W^2 & W^4 & W^6 \\ 1 & W^3 & W^6 & W^9 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{2.3}$$

Observe that the top row of the matrix evaluates the polynomial at $Z = 1$, a point where also $\omega = 0$. The second row evaluates $B_1 = B(Z = W = e^{i\omega_0})$, where ω_0 is some base frequency. The third row evaluates the Fourier transform for $2\omega_0$, and the bottom row for $3\omega_0$. The matrix could have more than four rows for more frequencies and more columns for more time points. I have made the matrix square in order to show you next how we can find the inverse matrix. The size of the matrix in (1.3) is $N = 4$. If we choose the base frequency ω_0 and hence W correctly, the inverse matrix will be

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = 1/N \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1/W & 1/W^2 & 1/W^3 \\ 1 & 1/W^2 & 1/W^4 & 1/W^6 \\ 1 & 1/W^3 & 1/W^6 & 1/W^9 \end{bmatrix} \begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix} \tag{2.4}$$

Multiplying the matrix of (1.4) with that of (1.3), we first see that the diagonals are $+1$ as desired. To have the off diagonals vanish, we need various sums, such as $1+W+W^2+W^3$ and $1+W^2+W^4+W^6$, to vanish. Every element (W^6, for example, or $1/W^9$) is a unit vector in the complex plane. In order for the sums of the unit vectors to vanish, we must ensure that the vectors pull symmetrically away from the origin. A uniform distribution of directions meets this requirement. In other words, W should be the N-th root of unity, i.e.,

$$W = \sqrt[N]{1} = e^{2\pi i/N} \tag{2.5}$$

The lowest frequency is zero, corresponding to the top row of (1.3). The next-to-the-lowest frequency we find by setting W in (1.5) to $Z = e^{i\omega_0}$. So $\omega_0 = 2\pi/N$; and for (1.4) to be inverse to (1.3), the frequencies required are

$$\omega_k = \frac{(0, 1, 2, \ldots, N-1)\, 2\pi}{N} \tag{2.6}$$

2.1.1 The Nyquist frequency

The highest frequency in equation (1.6), $\omega = 2\pi(N-1)/N$, is almost 2π. This frequency is twice as high as the Nyquist frequency $\omega = \pi$. The **Nyquist frequency** is normally thought of as the "highest possible" frequency, because $e^{i\pi t}$, for integer t, plots as $(\cdots, 1, -1, 1, -1, 1, -1, \cdots)$. The double Nyquist frequency function, $e^{i2\pi t}$, for integer t, plots as $(\cdots, 1, 1, 1, 1, 1, \cdots)$. So this frequency above the highest frequency is really zero frequency! We need to recall that $B(\omega) = B(\omega - 2\pi)$. Thus, all the frequencies near the upper end of the range (1.6) are really small negative frequencies. Negative frequencies on the interval $(-\pi, 0)$ were moved to interval $(\pi, 2\pi)$ by the matrix form of Fourier summation.

2.1.2 Laying out a mesh

In theoretical work and in programs, the definition $Z = e^{i\omega\Delta t}$ is often simplified to $\Delta t = 1$, leaving us with $Z = e^{i\omega}$. How do we know whether ω is given in radians per second or radians per sample? We may not invoke a cosine or an exponential unless the argument has no physical dimensions. So where we see ω without Δt, we know it is in units of radians per sample.

In practical work, frequency is typically given in cycles or **Hertz**, f, rather than radians, ω (where $\omega = 2\pi f$). Here we will now switch to f. We will design a computer **mesh** on a physical object (such as a waveform or a function of space). We often take the mesh to begin at $t = 0$, and continue till the end $t_{\max}$ of the object, so the time range $t_{\text{range}} = t_{\max}$. Then we decide how many points we want to use. This will be the N used in the discrete Fourier-transform program. Dividing the range by the number gives a mesh interval Δt.

Now let us see what this choice implies in the frequency domain. We customarily take the maximum frequency to be the Nyquist, either $f_{\max} = .5/\Delta t$ Hz or $\omega_{\max} = \pi/\Delta t$ radians/sec. The frequency range f_{range} goes from $-.5/\Delta t$ to $.5/\Delta t$. In summary:

- $\Delta t \quad = \quad t_{\text{range}}/N$ is time **resolution**.
- $f_{\text{range}} \quad = \quad 1/\Delta t \quad = \quad N/t_{\text{range}}$ is frequency range.
- $\Delta f \quad = \quad f_{\text{range}}/N \quad = \quad 1/t_{\text{range}}$ is frequency **resolution**.

In principle, we can always increase N to refine the calculation. Notice that increasing N sharpens the time resolution (makes Δt smaller) but does not sharpen the frequency resolution Δf, which remains fixed. Increasing N increases the frequency *range,* but not the frequency *resolution.*

What if we want to increase the frequency resolution? Then we need to choose t_{range} larger than required to cover our object of interest. Thus we either record data over a larger range, or we assert that such measurements would be zero. Three equations summarize the facts:

$$\Delta t \ f_{\text{range}} = 1 \tag{2.7}$$

$$\Delta f \ t_{\text{range}} = 1 \tag{2.8}$$

$$\Delta f \ \Delta t = \frac{1}{N} \tag{2.9}$$

Increasing *range* in the time domain increases *resolution* in the frequency domain and vice versa. Increasing **resolution** in one domain does not increase **resolution** in the other.

2.2 INVERTIBLE SLOW FT PROGRAM

Because Fourier sums are exactly invertible, some other things we often require can be done exactly by doing them in the frequency domain.

Typically, signals are real valued. But the programs in this chapter are for complex-valued signals. In order to use these programs, copy the real-valued signal into a complex array, where the signal goes into the real part of the complex numbers; the imaginary parts are then automatically set to zero.

There is no universally correct choice of **scale factor** in Fourier transform: choice of scale is a matter of convenience. Equations (1.3) and (1.4) mimic the Z-transform, so their scaling factors are convenient for the convolution theorem—that a product in the frequency domain is a convolution in the time domain. Obviously, the scaling factors of equations (1.3) and (1.4) will need to be interchanged for the complementary theorem that a convolution in the frequency domain is a product in the time domain. I like to use a scale factor that keeps the sums of squares the same in the time domain as in the frequency domain. Since I almost never need the scale factor, it simplifies life to omit it from the subroutine argument list. When a scaling program is desired, we can use a simple one like `scale()` on page 386. Complex-valued data can be scaled with `scale()` merely by doubling the value of `n`.

Fourier transform is just one of many transforms discussed in this book. In the case of most other transforms, the number of output values is different than the number of inputs. In addition, inverse transforms (and conjugate transforms), which will also be represented in code included in this book, transform in reverse, outputs to inputs. Finally, we will eventually combine transformations by addition or concatenation (one occurring after the other). All these considerations are expressed in the simple program `adjnull()`, which erases output before we begin. `Adjnull()` may seem like too trivial a function to put in a library routine, but at last count, 15 other routines in this book use it.

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

2.2.1 The simple FT code

The `mple ft()` routine exhibits features found in many physics and engineering programs. For example, the time-domain signal (which I call "`tt()`"), has `nt` values subscripted, from `tt(1)` to `tt(nt)`. The first value of this signal `tt(1)` is located in real physical time at `t0`. The time interval between values is `dt`. The value of `tt(it)` is at time `t0+(it-1)*dt`. I do not use "`if`" as a pointer on the frequency axis because `if` is a keyword in most programming languages. Instead, I count along the frequency axis with a variable named `ie`.

```
subroutine simpleft( adj, add, t0,dt,tt,nt,  f0,df,  ff,nf)
integer it,ie,      adj, add,         nt,            nf
complex cexp, cmplx,                 tt(nt),        ff(nf)
real pi2, freq, time, scale,   t0,dt,       f0,df
call adjnull(       adj, add,      tt,nt*2,       ff,nf*2 )
                        pi2= 2. * 3.14159265;          scale = 1./sqrt( 1.*nt)
                        df = (1./dt) / nf
                        f0 =   - .5/dt
do ie = 1, nf {   freq= f0 + df*(ie-1)
do it = 1, nt {   time= t0 + dt*(it-1)
        if( adj == 0 )
                ff(ie)= ff(ie) + tt(it) * cexp(cmplx(0., pi2*freq*time)) * scale
        else
                tt(it)= tt(it) + ff(ie) * cexp(cmplx(0.,-pi2*freq*time)) * scale
        }}
return; end
```

The total frequency band is 2π radians per sample unit or $1/\Delta t$ Hz. Dividing the total interval by the number of points `nf` gives Δf. We could choose the frequencies to run from 0 to 2π radians/sample. That would work well for many applications, but it would be a nuisance for applications such as differentiation in the frequency domain, which require multiplication by $-i\omega$ including the **negative frequencies** as well as the positive. So it seems more natural to begin at the most negative frequency and step forward to the most positive frequency. Next, we must make a confusing choice.

2.3 SETTING UP THE FAST FOURIER TRANSFORM

Typically we **Fourier transform** seismograms about a thousand points long. Under these conditions another Fourier summation method works about a hundred times faster than those already given. Unfortunately, the faster Fourier transform program is not so transparently clear as the programs given earlier. Also, it is slightly less flexible. The speedup is so overwhelming, however, that the fast program is always used in routine work.

Flexibility may be lost because the basic fast program works with complex-valued signals, so we ordinarily convert our real signals to complex ones (by adding a zero imaginary part). More flexibility is lost because typical fast FT programs require the data length to be an integral power of 2. Thus geophysical datasets often have zeros appended (a process called "**zero pad**ding") until the data length is a power of 2. From time to time I notice clumsy computer code written to deduce a number that is a power of 2 and is larger than the length of a dataset. An answer is found by rounding up the logarithm to base 2. The more obvious and the quicker way to get the desired value, however, is with the simple Fortran function `pad2()`.

```
integer function pad2( n )
integer n
pad2 = 1
while( pad2 < n )
        pad2 = pad2 * 2
return; end
```

How fast is the fast Fourier transform method? The answer depends on the size of the data. The matrix times vector operation in (1.3) requires N^2 multiplications and additions. That determines the speed of the slow transform. For the fast method the number of adds and multiplies is proportional to $N \log_2 N$. Since $2^{10} = 1024$, the speed ratio is typically 1024/10 or about 100. In reality, the fast method is not quite that fast, depending on certain details of overhead and implementation. In 1987 I tested the three programs on a 1024-point real signal and found times

`slowft`	153s
`polyft`	36s
`ftu`	.7s

Below is `ftu()`, a version of the **fast Fourier transform** program. There are many versions of the program—I have chosen this one for its simplicity. Considering the complexity of the task, it is remarkable that no auxiliary memory vectors are required; indeed, the output vector lies on top of the input vector. To run this program, your first step might be to copy your real-valued signal into a complex-valued array. Then append enough zeros to fill in the remaining space.

```
subroutine ftu( signi, nx, cx )
#   complex fourier transform with unitary scaling
#
#               1         nx          signi*2*pi*i*(j-1)*(k-1)/nx
#   cx(k)  =  -------- * sum cx(j) * e
#             sqrt(nx)   j=1             for k=1,2,...,nx=2**integer
#
integer nx, i, j, k, m, istep, pad2
real    signi, scale, arg
complex cx(nx), cmplx, cw, cdel, ct
if( nx != pad2(nx) )    call erexit('ftu: nx not a power of 2')
scale = 1. / sqrt( 1.*nx)
do i= 1, nx
        cx(i) = cx(i) * scale
j = 1;  k = 1
do i= 1, nx {
        if (i<=j) { ct = cx(j); cx(j) = cx(i); cx(i) = ct }
        m = nx/2
        while (j>m && m>1) { j = j-m; m = m/2 }         # "&&" means .AND.
        j = j+m
        }
repeat {
        istep = 2*k;   cw = 1.;   arg = signi*3.14159265/k
```

```
        cdel = cmplx( cos(arg), sin(arg))
        do m= 1, k {
                do i= m, nx, istep
                        { ct=cw*cx(i+k);  cx(i+k)=cx(i)-ct;  cx(i)=cx(i)+ct }
                cw = cw * cdel
                }
        k = istep
        if(k>=nx) break
        }
return; end
```

The following two lines serve to Fourier transform a vector of 1024 complex-valued points, and then to **inverse Fourier transform** them back to the original data:

```
call ftu(  1., 1024, cx)
 call ftu( -1., 1024, cx)
```

An engineering reference given at the end of this chapter contains many other versions of the FFT program. One version transforms real-valued signals to complex-valued frequency functions in the interval $0 \leq \omega < \pi$. Others that do not transform data on top of itself may be faster with specialized computer architectures.

EXERCISES:

1 Consider an even time function that is constant for all frequencies less than ω_0 and zero for all frequencies above ω_0. What is the rate of decay of amplitude with time for this function?

2 Waves spreading from a point source decay in energy as the area on a sphere. The amplitude decays as the square root of energy. This implies a certain decay in time. The time-decay rate is the same if the waves reflect from planar interfaces. To what power of time t do the signal amplitudes decay? For waves backscattered to the source from point reflectors, energy decays as distance to the minus fourth power. What is the associated decay with time?

2.3.1 Shifted spectrum

I have omitted a bit of theory here that is found in PVI.

For an FT matrix of arbitrary size N, the desired shift is $N/2$, so values at alternate points in the time axis are multiplied by -1. A subroutine for that purpose is fth().

```
# FT a vector in a matrix, with first omega = - pi
#
subroutine  fth( adj,sign, m1, n12, cx)
integer i,       adj,      m1, n12
real sign
complex cx(m1,n12)
temporary complex temp(n12)
do i= 1, n12
        temp(i) = cx(1,i)
if( adj == 0)   { do i= 2, n12, 2
                        temp(i) =  -temp(i)
                  call ftu(  sign, n12, temp)
                }
else            { call ftu( -sign, n12, temp)
                  do i= 2, n12, 2
                        temp(i) =  -temp(i)
                }
do i= 1, n12
        cx(1,i) = temp(i)
return; end
```

To Fourier transform a 1024-point complex vector cx(1024) and then inverse transform it, you would

```
call fth( 0, 1., 1, 1024, cx)
 call fth( 1, 1., 1, 1024, cx)
```

You might wonder about the apparent redundancy of using both the argument conj and the argument sign. Having two arguments instead of one allows us to define the *forward* transform for a *time* axis with the opposite sign as the forward transform for a *space* axis.

The subroutine fth() is somewhat cluttered by the inclusion of a frequently needed practical feature---namely, the facility to extract vectors from a matrix, transform the vectors, and then restore them into the matrix.

2.4 TWO-DIMENSIONAL FT

The program fth() is set up so that the vectors transformed can be either rows or columns of a two-dimensional array. To see how this works, recall that in Fortran a matrix allocated as (n1,n2) can be subscripted as a matrix (i1,i2) or as a long vector (i1 + n1*(i2-1),1), and call sub(x(i1,i2)) passes the subroutine a pointer to the (i1,i2) element. To transform an entire axis, the subroutines ft1axis() and ft2axis() are given. For a two-dimensional FT, we simply call both ft1axis() and ft2axis() in either order.

```
# 1D Fourier transform on a 2D data set along the 1-axis
#
subroutine  ft1axis( adj, sign1, n1,n2, cx)
integer i2,          adj,       n1,n2
complex cx(n1,n2)
real sign1
do i2= 1, n2
                call fth( adj, sign1, 1,n1, cx(1,i2))
return; end
```

```
# 1D Fourier transform on a 2D data set along the 2-axis
#
subroutine  ft2axis( adj, sign2, n1,n2, cx)
integer i1,          adj,       n1,n2
complex cx(n1,n2)
real    sign2
do i1= 1, n1
                call fth( adj, sign2, n1,n2, cx(i1,1))
return; end
```

I confess that there are faster ways to do things than those I have shown you above. When we are doing many FTs, for example, the overhead calculations done the first time should be saved for use on subsequent FTs, as in the subroutine rocca() included in IEI. Further, manufacturers of computers for heavy numerical use generally design special FT codes for their architecture. Although the basic fast FT used here ingeniously stores its output on top of its input, that feature is not compatible with vectorizing architectures.

2.4.1 Basics of two-dimensional Fourier transform

Before going any further, let us review some basic facts about two-dimensional Fourier transform. A two-dimensional function is represented in a computer as numerical values in a matrix, whereas a one-dimensional Fourier transform in a computer is an operation on a vector. A 2-D Fourier transform can be computed by a sequence of 1-D Fourier transforms. We can first transform each column vector of the matrix and then each row vector of the matrix. Alternately, we can first do the rows and later do the columns. This is diagrammed as follows:

$$
\begin{array}{ccc}
p(t,\ x) & \longleftrightarrow & P(t,\ k_x) \\
\updownarrow & & \updownarrow \\
P(\omega,\ x) & \longleftrightarrow & P(\omega,\ k_x)
\end{array}
$$

The diagram has the notational problem that we cannot maintain the usual convention of using a lower-case letter for the domain of physical space and an upper-case letter for the Fourier domain, because that convention cannot include the mixed objects $P(t, k_x)$ and $P(\omega, x)$. Rather than invent some new notation, it seems best to let the reader rely on the context: the arguments of the function must help name the function.

An example of two-dimensional Fourier transforms on typical deep-ocean data is shown in Figure 1.1. In the deep ocean, sediments are fine-grained

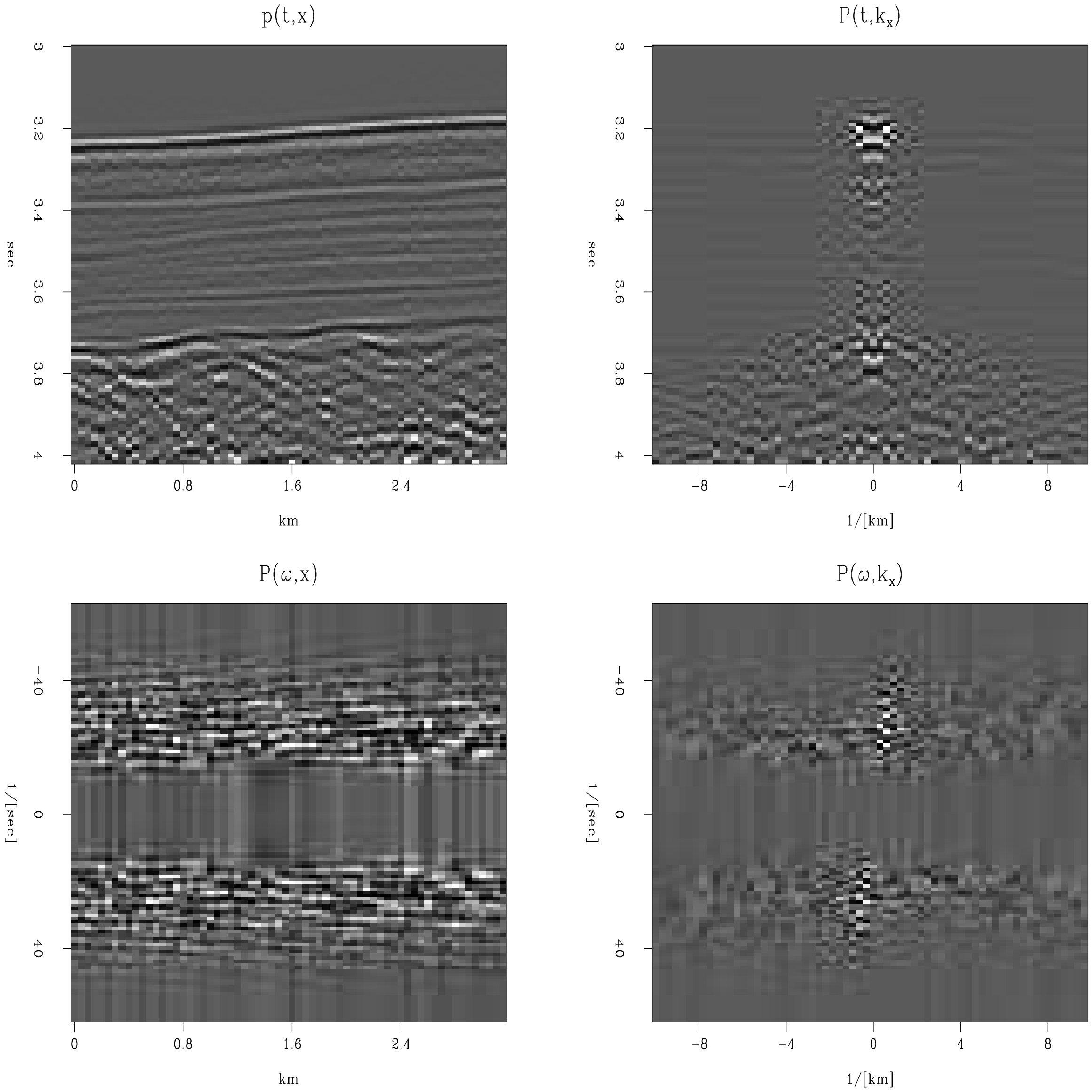

Figure 2.1: A deep-marine dataset $p(t, x)$ from Alaska (U.S. Geological Survey) and the *real* part of various Fourier transforms of it. Because of the long traveltime through the water, the time axis does not begin at $t = 0$. dft-plane4 [ER]

and deposit slowly in flat, regular, horizontal beds. The lack of permeable rocks such as sandstone severely reduces the potential for petroleum production from the deep ocean. The fine-grained shales overlay irregular, igneous, basement rocks. In the plot of $P(t,k_x)$, the lateral continuity of the sediments is shown by the strong spectrum at low k_x. The igneous rocks show a k_x spectrum extending to such large k_x that the deep data may be somewhat spatially aliased (sampled too coarsely). The plot of $P(\omega,x)$ shows that the data contains no low-frequency energy. The dip of the sea floor shows up in (ω,k_x)-space as the energy crossing the origin at an angle.

Altogether, the two-dimensional Fourier transform of a collection of seismograms involves only twice as much computation as the one-dimensional Fourier transform of each seismogram. This is lucky. Let us write some equations to establish that the asserted procedure does indeed do a 2-D Fourier transform. Say first that any function of x and t may be expressed as a superposition of sinusoidal functions:

$$p(t,x) \quad = \quad \int\int \; e^{-i\omega t + ik_x x} \; P(\omega,k_x) \; d\omega \; dk_x \tag{2.10}$$

The double integration can be nested to show that the temporal transforms are done first (inside):

$$\begin{aligned} p(t,x) \quad &= \quad \int \; e^{i\,k_x x} \; \left[\int \; e^{-i\omega t} \; P(\omega,k_x) \; d\omega\right] dk_x \\ &= \quad \int \; e^{i\,k_x x} \; P(t,k_x) \; dk_x \end{aligned}$$

The quantity in brackets is a Fourier transform over ω done for each and every k_x. Alternately, the nesting could be done with the k_x-integral on the inside. That would imply rows first instead of columns (or vice versa). It is the separability of $\exp(-i\omega t + i\,k_x x)$ into a product of exponentials that makes the computation this easy and cheap.

2.4.2 Signs in Fourier transforms

In Fourier transforming t-, x-, and z-coordinates, we must choose a sign convention for each coordinate. Of the two alternative sign conventions, electrical engineers have chosen one and physicists another. While both have good reasons for their choices, our circumstances more closely resemble those of physicists, so we will use their convention. For the *inverse* Fourier transform, our choice is

$$p(t,x,z) \quad = \quad \int\int\int \; e^{-i\omega t \,+\, ik_x x \,+\, ik_z z} \; P(\omega,k_x,k_z) \; d\omega \, dk_x \, dk_z \tag{2.11}$$

For the *forward* Fourier transform, the space variables carry a *negative* sign, and time carries a *positive* sign.

Let us see the reasons why electrical engineers have made the opposite choice, and why we go with the physicists. Essentially, engineers transform only the time axis, whereas physicists transform both time and space axes. Both are simplifying their lives by their choice of sign convention, but physicists complicate their time axis in order to simplify their many space axes. The engineering choice minimizes the number of minus signs associated with the time axis, because for engineers, d/dt is associated with $i\omega$ instead of, as is the case for us and for physicists, with $-i\omega$. We confirm this with equation (1.11). Physicists and geophysicists deal with many more independent variables than time. Besides the obvious three space axes are their mutual combinations, such as midpoint and offset.

You might ask, why not make *all* the signs positive in equation (1.11)? The reason is that in that case waves would not move in a positive direction along the space axes. This would be especially unnatural when the space axis was a radius. Atoms, like geophysical sources, always radiate from a point to infinity, not the other way around. Thus, in equation (1.11) the sign of the spatial frequencies must be opposite that of the temporal frequency.

The only good reason I know to choose the engineering convention is that we might compute with an array processor built and microcoded by engineers. Conflict of sign convention is not a problem for the programs that transform complex-valued time functions to complex-valued frequency functions, because there the sign convention is under the user's control. But sign conflict does make a difference when we use any program that converts real-time functions to complex frequency functions. The way to live in both worlds is to imagine that the frequencies produced by such a program do not range from 0 to $+\pi$ as the program description says, but from 0 to $-\pi$. Alternately, we could always take the complex conjugate of the transform, which would swap the sign of the ω-axis.

2.4.3 Examples of 2-D FT

An example of a two-dimensional Fourier transform of a pulse is shown in Figure 1.2. Notice the location of the pulse. It is closer to the time axis than the frequency axis. This will affect the real part of the FT in a certain way (see exercises). Notice the broadening of the pulse. It was an impulse smoothed over time (vertically) by convolution with (1,1) and over space (horizontally) with (1,4,6,4,1). This will affect the real part of the FT in another way.

Another example of a two-dimensional Fourier transform is given in Figure 1.3. This example simulates an impulsive air wave originating at a point on the x-axis. We see a wave propagating in each direction from the location of the source of the wave. In Fourier space there are also two lines, one for each wave. Notice that there are other lines which do not go through

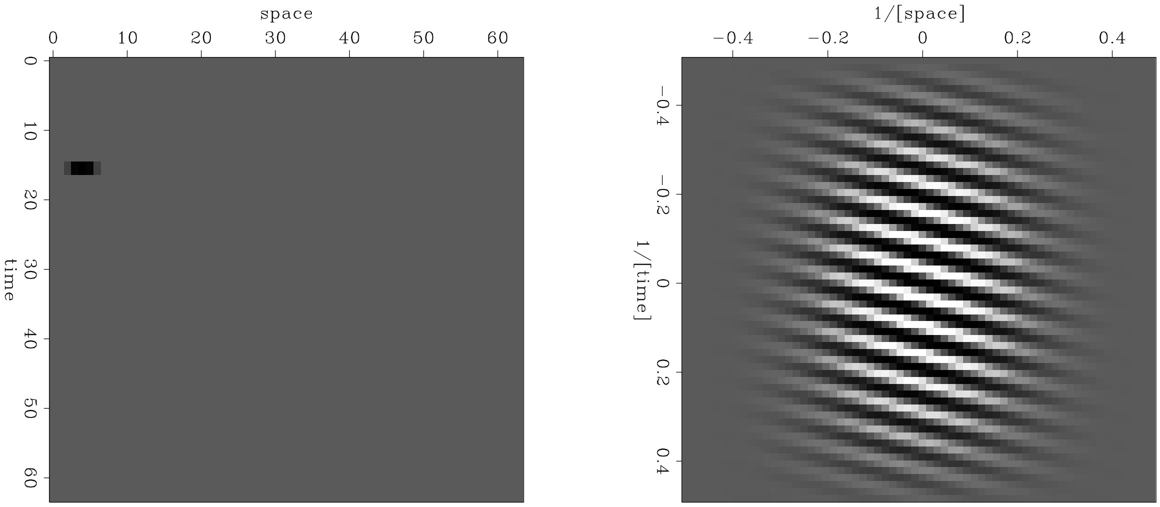

Figure 2.2: A broadened pulse (left) and the real part of its FT (right). dft-ft2dofpulse [ER]

`the origin; these lines are called ‘‘spatial aliases.’’ Each actually goes through the origin of another square plane that is not shown, but which we can imagine alongside the one shown. These other planes are periodic replicas of the one shown.`

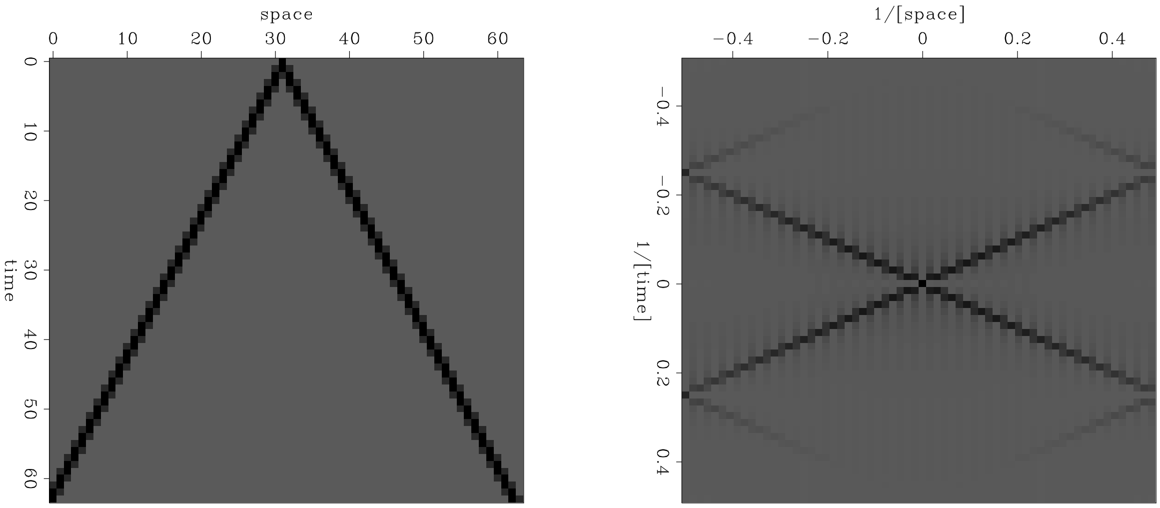

Figure 2.3: A simulated air wave (left) and the amplitude of its FT (right). dft-airwave [ER]

EXERCISES:

1 `Write ftlag() starting from ftlagslow() and fth().`

2 Most time functions are real. Their imaginary part is zero. Show that this means that $F(\omega, k)$ can be determined from $F(-\omega, -k)$.

3 What would change in Figure 1.2 if the pulse were moved (a) earlier on the t-axis, and (b) further on the x-axis? What would change in Figure 1.2 if instead the time axis were smoothed with (1,4,6,4,1) and the space axis with (1,1)?

4 What would Figure 1.3 look like on an earth with half the earth velocity?

5 Numerically (or theoretically) compute the two-dimensional spectrum of a plane wave [$\delta(t-px)$], where the plane wave has a randomly fluctuating amplitude: say, rand(x) is a random number between ± 1, and the randomly modulated plane wave is [$(1 \ + \ .2\,\mathrm{rand}(x))\,\delta(t-px)$].

6 Explain the horizontal ‘‘layering’’ in Figure 1.1 in the plot of $P(\omega, x)$. What determines the ‘‘layer’’ separation? What determines the ‘‘layer’’ slope?

2.5 References

Special issue on fast Fourier transform, June 1969: IEEE Trans. on Audio and Electroacoustics (now known as IEEE Trans. on Acoustics, Speech, and Signal Processing), **AU-17**, entire issue (66-172).

Index