Chapter 12.   Fast Fourier Transform

12.0 Introduction

A very large class of important computational problems falls under the general rubric of "Fourier transform methods" or "spectral methods." For some of these problems, the Fourier transform is simply an efficient computational tool for accomplishing certain common manipulations of data. In other cases, we have problems for which the Fourier transform (or the related "power spectrum") is itself of intrinsic interest. These two kinds of problems share a common methodology.

Largely for historical reasons the literature on Fourier and spectral methods has been disjoint from the literature on "classical" numerical analysis. Nowadays there is no justification for such a split. Fourier methods are commonplace in research and we shall not treat them as specialized or arcane. At the same time, we realize that many computer users have had relatively less experience with this field than with, say, differential equations or numerical integration. Therefore our summary of analytical results will be more complete. Numerical algorithms, per se, begin in §12.2. Various applications of Fourier transform methods are discussed in Chapter 13.

A physical process can be described either in the *time domain*, by the values of some quantity h as a function of time t, e.g., $h(t)$, or else in the *frequency domain*, where the process is specified by giving its amplitude H (generally a complex number indicating phase also) as a function of frequency f, that is $H(f)$, with $-\infty < f < \infty$. For many purposes it is useful to think of $h(t)$ and $H(f)$ as being two different *representations* of the *same* function. One goes back and forth between these two representations by means of the *Fourier transform* equations,

$$
\begin{aligned}
H(f) &= \int_{-\infty}^{\infty} h(t)e^{2\pi i f t}dt \\
h(t) &= \int_{-\infty}^{\infty} H(f)e^{-2\pi i f t}df
\end{aligned}
\tag{12.0.1}
$$

If t is measured in seconds, then f in equation (12.0.1) is in cycles per second, or Hertz (the unit of frequency). However, the equations work with other units too. If h is a function of position x (in meters), H will be a function of inverse wavelength (cycles per meter), and so on. If you are trained as a physicist or mathematician, you are probably more used to using *angular frequency* ω, which is given in *radians* per sec. The relation between ω and f, $H(\omega)$ and $H(f)$ is

$$
\omega \equiv 2\pi f \qquad H(\omega) \equiv [H(f)]_{f=\omega/2\pi}
\tag{12.0.2}
$$

and equation (12.0.1) looks like this

$$H(\omega) = \int_{-\infty}^{\infty} h(t)e^{i\omega t}dt$$

$$h(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} H(\omega)e^{-i\omega t}d\omega$$

$$(12.0.3)$$

We were raised on the ω-convention, but we changed! There are fewer factors of 2π to remember if you use the f-convention, especially when we get to discretely sampled data in §12.1.

From equation (12.0.1) it is evident at once that Fourier transformation is a *linear* operation. The transform of the sum of two functions is equal to the sum of the transforms. The transform of a constant times a function is that same constant times the transform of the function.

In the time domain, function $h(t)$ may happen to have one or more special symmetries It might be *purely real* or *purely imaginary* or it might be *even*, $h(t) = h(-t)$, or *odd*, $h(t) = -h(-t)$. In the frequency domain, these symmetries lead to relationships between $H(f)$ and $H(-f)$. The following table gives the correspondence between symmetries in the two domains:

If ...	then ...
$h(t)$ is real	$H(-f) = [H(f)]^*$
$h(t)$ is imaginary	$H(-f) = -[H(f)]^*$
$h(t)$ is even	$H(-f) = H(f)$ [i.e., $H(f)$ is even]
$h(t)$ is odd	$H(-f) = -H(f)$ [i.e., $H(f)$ is odd]
$h(t)$ is real and even	$H(f)$ is real and even
$h(t)$ is real and odd	$H(f)$ is imaginary and odd
$h(t)$ is imaginary and even	$H(f)$ is imaginary and even
$h(t)$ is imaginary and odd	$H(f)$ is real and odd

In subsequent sections we shall see how to use these symmetries to increase computational efficiency.

Here are some other elementary properties of the Fourier transform. (We'll use the "$\Longleftrightarrow$" symbol to indicate transform pairs.) If

$$h(t) \Longleftrightarrow H(f)$$

is such a pair, then other transform pairs are

$$h(at) \Longleftrightarrow \frac{1}{|a|}H(\frac{f}{a}) \qquad \text{``time scaling''} \qquad (12.0.4)$$

$$\frac{1}{|b|}h(\frac{t}{b}) \Longleftrightarrow H(bf) \qquad \text{``frequency scaling''} \qquad (12.0.5)$$

$$h(t - t_0) \Longleftrightarrow H(f)\, e^{2\pi i f t_0} \qquad \text{``time shifting''} \qquad (12.0.6)$$

$$h(t)\, e^{-2\pi i f_0 t} \Longleftrightarrow H(f - f_0) \qquad \text{``frequency shifting''} \qquad (12.0.7)$$

With two functions $h(t)$ and $g(t)$, and their corresponding Fourier transforms $H(f)$ and $G(f)$, we can form two combinations of special interest. The *convolution* of the two functions, denoted $g * h$, is defined by

$$g * h \equiv \int_{-\infty}^{\infty} g(\tau)h(t - \tau) \, d\tau \qquad (12.0.8)$$

Note that $g * h$ is a function in the time domain and that $g * h = h * g$. It turns out that the function $g * h$ is one member of a simple transform pair

$$g * h \Longleftrightarrow G(f)H(f) \qquad \text{"Convolution Theorem"} \qquad (12.0.9)$$

In other words, the Fourier transform of the convolution is just the product of the individual Fourier transforms.

The *correlation* of two functions, denoted $\text{Corr}(g, h)$, is defined by

$$\text{Corr}(g, h) \equiv \int_{-\infty}^{\infty} g(\tau + t)h(\tau) \, d\tau \qquad (12.0.10)$$

The correlation is a function of t, which is called the *lag*. It therefore lies in the time domain, and it turns out to be one member of the transform pair:

$$\text{Corr}(g, h) \Longleftrightarrow G(f)H^*(f) \qquad \text{"Correlation Theorem"} \qquad (12.0.11)$$

[More generally, the second member of the pair is $G(f)H(-f)$, but we are restricting ourselves to the usual case in which g and h are real functions, so we take the liberty of setting $H(-f) = H^*(f)$.] This result shows that multiplying the Fourier transform of one function by the complex conjugate of the Fourier transform of the other gives the Fourier transform of their correlation. The correlation of a function with itself is called its *autocorrelation*. In this case (12.0.11) becomes the transform pair

$$\text{Corr}(g, g) \Longleftrightarrow |G(f)|^2 \qquad \text{"Wiener-Khinchin Theorem"} \qquad (12.0.12)$$

The *total power* in a signal is the same whether we compute it in the time domain or in the frequency domain. This result is known as *Parseval's theorem*:

$$\text{Total Power} \equiv \int_{-\infty}^{\infty} |h(t)|^2 \, dt = \int_{-\infty}^{\infty} |H(f)|^2 \, df \qquad (12.0.13)$$

Frequently one wants to know "how much power" is contained in the frequency interval between f and $f + df$. In such circumstances one does not usually distinguish between positive and negative f, but rather regards f as varying from 0 ("zero frequency" or D.C.) to $+\infty$. In such cases, one defines the *one-sided power spectral density (PSD)* of the function h as

$$P_h(f) \equiv |H(f)|^2 + |H(-f)|^2 \qquad 0 \le f < \infty \qquad (12.0.14)$$

so that the total power is just the integral of $P_h(f)$ from $f = 0$ to $f = \infty$. When the function $h(t)$ is real, then the two terms in (12.0.14) are equal, so $P_h(f) = 2\,|H(f)|^2$.
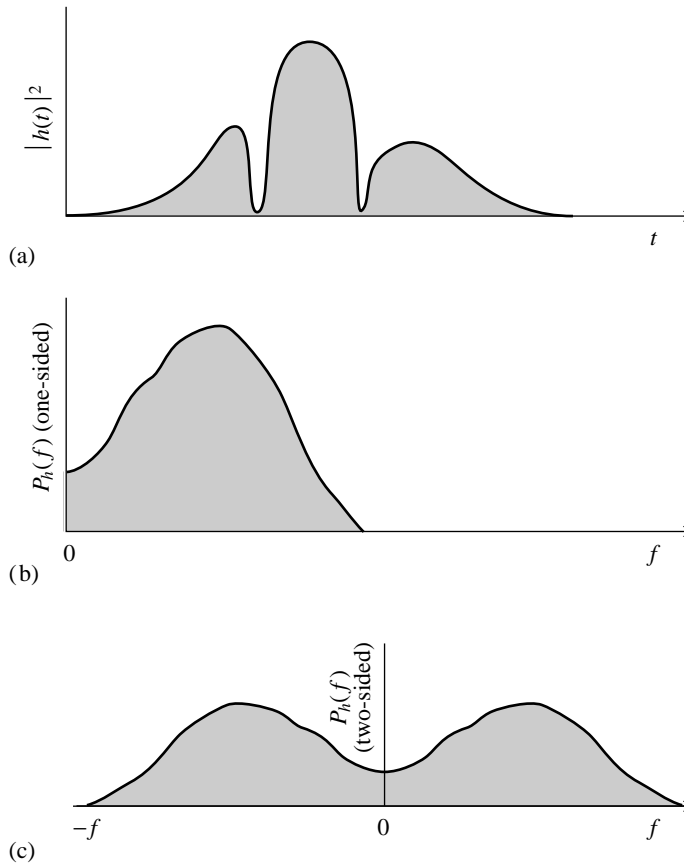
Figure 12.0.1. Normalizations of one- and two-sided power spectra. The area under the square of the function, (a), equals the area under its one-sided power spectrum at positive frequencies, (b), and also equals the area under its two-sided power spectrum at positive and negative frequencies, (c).

Be warned that one occasionally sees PSDs defined without this factor two. These, strictly speaking, are called *two-sided power spectral densities*, but some books are not careful about stating whether one- or two-sided is to be assumed. We will always use the one-sided density given by equation (12.0.14). Figure 12.0.1 contrasts the two conventions.

If the function $h(t)$ goes endlessly from $-\infty < t < \infty$, then its total power and power spectral density will, in general, be infinite. Of interest then is the *(one- or two-sided) power spectral density per unit time*. This is computed by taking a long, but finite, stretch of the function $h(t)$, computing its PSD [that is, the PSD of a function that equals $h(t)$ in the finite stretch but is zero everywhere else], and then dividing the resulting PSD by the length of the stretch used. Parseval's theorem in this case states that the integral of the one-sided PSD-per-unit-time over positive frequency is equal to the mean square amplitude of the signal $h(t)$.

You might well worry about how the PSD-per-unit-time, which is a function of frequency f, converges as one evaluates it using longer and longer stretches of data. This interesting question is the content of the subject of "power spectrum estimation," and will be considered below in §13.4–§13.7. A crude answer for

now is: The PSD-per-unit-time converges to finite values at all frequencies *except* those where $h(t)$ has a discrete sine-wave (or cosine-wave) component of finite amplitude. At those frequencies, it becomes a delta-function, i.e., a sharp spike, whose width gets narrower and narrower, but whose area converges to be the mean square amplitude of the discrete sine or cosine component at that frequency.

We have by now stated all of the analytical formalism that we will need in this chapter with one exception: In computational work, especially with experimental data, we are almost never given a continuous function $h(t)$ to work with, but are given, rather, a list of measurements of $h(t_i)$ for a discrete set of t_i's. The profound implications of this seemingly unimportant fact are the subject of the next section.

CITED REFERENCES AND FURTHER READING:

Champeney, D.C. 1973, *Fourier Transforms and Their Physical Applications* (New York: Academic Press).

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press).

12.1 Fourier Transform of Discretely Sampled Data

In the most common situations, function $h(t)$ is sampled (i.e., its value is recorded) at evenly spaced intervals in time. Let Δ denote the time interval between consecutive samples, so that the sequence of sampled values is

$$h_n = h(n\Delta) \qquad n = \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots \qquad (12.1.1)$$

The reciprocal of the time interval Δ is called the *sampling rate*; if Δ is measured in seconds, for example, then the sampling rate is the number of samples recorded per second.

Sampling Theorem and Aliasing

For any sampling interval Δ, there is also a special frequency f_c, called the *Nyquist critical frequency*, given by

$$f_c \equiv \frac{1}{2\Delta} \qquad (12.1.2)$$

If a sine wave of the Nyquist critical frequency is sampled at its positive peak value, then the next sample will be at its negative trough value, the sample after that at the positive peak again, and so on. Expressed otherwise: *Critical sampling of a sine wave is two sample points per cycle.* One frequently chooses to measure time in units of the sampling interval Δ. In this case the Nyquist critical frequency is just the constant 1/2.

The Nyquist critical frequency is important for two related, but distinct, reasons. One is good news, and the other bad news. First the good news. It is the remarkable

now is: The PSD-per-unit-time converges to finite values at all frequencies *except* those where $h(t)$ has a discrete sine-wave (or cosine-wave) component of finite amplitude. At those frequencies, it becomes a delta-function, i.e., a sharp spike, whose width gets narrower and narrower, but whose area converges to be the mean square amplitude of the discrete sine or cosine component at that frequency.

We have by now stated all of the analytical formalism that we will need in this chapter with one exception: In computational work, especially with experimental data, we are almost never given a continuous function $h(t)$ to work with, but are given, rather, a list of measurements of $h(t_i)$ for a discrete set of t_i's. The profound implications of this seemingly unimportant fact are the subject of the next section.

CITED REFERENCES AND FURTHER READING:

Champeney, D.C. 1973, *Fourier Transforms and Their Physical Applications* (New York: Academic Press).

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press).

12.1 Fourier Transform of Discretely Sampled Data

In the most common situations, function $h(t)$ is sampled (i.e., its value is recorded) at evenly spaced intervals in time. Let Δ denote the time interval between consecutive samples, so that the sequence of sampled values is

$$h_n = h(n\Delta) \qquad n = \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots \qquad (12.1.1)$$

The reciprocal of the time interval Δ is called the *sampling rate*; if Δ is measured in seconds, for example, then the sampling rate is the number of samples recorded per second.

Sampling Theorem and Aliasing

For any sampling interval Δ, there is also a special frequency f_c, called the *Nyquist critical frequency*, given by

$$f_c \equiv \frac{1}{2\Delta} \qquad (12.1.2)$$

If a sine wave of the Nyquist critical frequency is sampled at its positive peak value, then the next sample will be at its negative trough value, the sample after that at the positive peak again, and so on. Expressed otherwise: *Critical sampling of a sine wave is two sample points per cycle.* One frequently chooses to measure time in units of the sampling interval Δ. In this case the Nyquist critical frequency is just the constant 1/2.

The Nyquist critical frequency is important for two related, but distinct, reasons. One is good news, and the other bad news. First the good news. It is the remarkable

fact known as the *sampling theorem*: If a continuous function $h(t)$, sampled at an interval Δ, happens to be *bandwidth limited* to frequencies smaller in magnitude than f_c, i.e., if $H(f) = 0$ for all $|f| \geq f_c$, then the function $h(t)$ is *completely determined* by its samples h_n. In fact, $h(t)$ is given explicitly by the formula

$$h(t) = \Delta \sum_{n=-\infty}^{+\infty} h_n \frac{\sin[2\pi f_c(t - n\Delta)]}{\pi(t - n\Delta)} \tag{12.1.3}$$

This is a remarkable theorem for many reasons, among them that it shows that the "information content" of a bandwidth limited function is, in some sense, infinitely smaller than that of a general continuous function. Fairly often, one is dealing with a signal that is known on physical grounds to be bandwidth limited (or at least approximately bandwidth limited). For example, the signal may have passed through an amplifier with a known, finite frequency response. In this case, the sampling theorem tells us that the entire information content of the signal can be recorded by sampling it at a rate Δ^{-1} equal to twice the maximum frequency passed by the amplifier (cf. 12.1.2).

Now the bad news. The bad news concerns the effect of sampling a continuous function that is *not* bandwidth limited to less than the Nyquist critical frequency. In that case, it turns out that all of the power spectral density that lies outside of the frequency range $-f_c < f < f_c$ is spuriously moved into that range. This phenomenon is called *aliasing*. Any frequency component outside of the frequency range $(-f_c, f_c)$ is *aliased* (falsely translated) into that range by the very act of discrete sampling. You can readily convince yourself that two waves $\exp(2\pi i f_1 t)$ and $\exp(2\pi i f_2 t)$ give the same samples at an interval Δ if and only if f_1 and f_2 differ by a multiple of $1/\Delta$, which is just the width in frequency of the range $(-f_c, f_c)$. There is little that you can do to remove aliased power once you have discretely sampled a signal. The way to overcome aliasing is to (i) know the natural bandwidth limit of the signal — or else enforce a known limit by analog filtering of the continuous signal, and then (ii) sample at a rate sufficiently rapid to give at least two points per cycle of the highest frequency present. Figure 12.1.1 illustrates these considerations.

To put the best face on this, we can take the alternative point of view: If a continuous function has been competently sampled, then, when we come to estimate its Fourier transform from the discrete samples, we can *assume* (or rather we *might as well* assume) that its Fourier transform is equal to zero outside of the frequency range in between $-f_c$ and f_c. Then we look to the Fourier transform to tell whether the continuous function *has* been competently sampled (aliasing effects minimized). We do this by looking to see whether the Fourier transform is already approaching zero as the frequency approaches f_c from below, or $-f_c$ from above. If, on the contrary, the transform is going towards some finite value, then chances are that components outside of the range have been folded back over onto the critical range.

Discrete Fourier Transform

We now estimate the Fourier transform of a function from a finite number of its sampled points. Suppose that we have N consecutive sampled values

$$h_k \equiv h(t_k), \qquad t_k \equiv k\Delta, \qquad k = 0, 1, 2, \ldots, N-1 \tag{12.1.4}$$
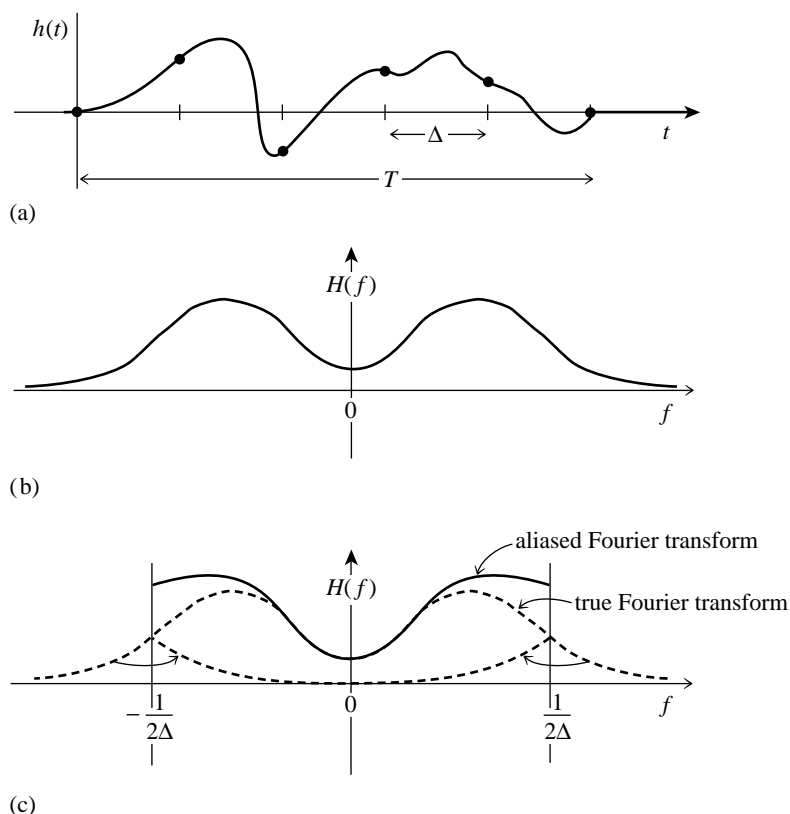
(a)

(b)

(c)

Figure 12.1.1.   The continuous function shown in (a) is nonzero only for a finite interval of time T. It follows that its Fourier transform, whose modulus is shown schematically in (b), is not bandwidth limited but has finite amplitude for all frequencies. If the original function is sampled with a sampling interval Δ, as in (a), then the Fourier transform (c) is defined only between plus and minus the Nyquist critical frequency. Power outside that range is folded over or "aliased" into the range. The effect can be eliminated only by low-pass filtering the original function *before sampling*.

so that the sampling interval is Δ. To make things simpler, let us also suppose that N is even. If the function $h(t)$ is nonzero only in a finite interval of time, then that whole interval of time is supposed to be contained in the range of the N points given. Alternatively, if the function $h(t)$ goes on forever, then the sampled points are supposed to be at least "typical" of what $h(t)$ looks like at all other times.

With N numbers of input, we will evidently be able to produce no more than N independent numbers of output. So, instead of trying to estimate the Fourier transform $H(f)$ at all values of f in the range $-f_c$ to f_c, let us seek estimates only at the discrete values

$$f_n \equiv \frac{n}{N\Delta}, \qquad n = -\frac{N}{2}, \ldots, \frac{N}{2} \qquad (12.1.5)$$

The extreme values of n in (12.1.5) correspond exactly to the lower and upper limits of the Nyquist critical frequency range. If you are really on the ball, you will have noticed that there are $N + 1$, not N, values of n in (12.1.5); it will turn out that the two extreme values of n are not independent (in fact they are equal), but all the others are. This reduces the count to N.

The remaining step is to approximate the integral in (12.0.1) by a discrete sum:

$$H(f_n) = \int_{-\infty}^{\infty} h(t)e^{2\pi i f_n t}dt \approx \sum_{k=0}^{N-1} h_k\, e^{2\pi i f_n t_k}\Delta = \Delta \sum_{k=0}^{N-1} h_k\, e^{2\pi i kn/N}$$

$$(12.1.6)$$

Here equations (12.1.4) and (12.1.5) have been used in the final equality. The final summation in equation (12.1.6) is called the *discrete Fourier transform* of the N points h_k. Let us denote it by H_n,

$$H_n \equiv \sum_{k=0}^{N-1} h_k\, e^{2\pi i kn/N} \qquad (12.1.7)$$

The discrete Fourier transform maps N complex numbers (the h_k's) into N complex numbers (the H_n's). It does not depend on any dimensional parameter, such as the time scale Δ. The relation (12.1.6) between the discrete Fourier transform of a set of numbers and their continuous Fourier transform when they are viewed as samples of a continuous function sampled at an interval Δ can be rewritten as

$$H(f_n) \approx \Delta H_n \qquad (12.1.8)$$

where f_n is given by (12.1.5).

Up to now we have taken the view that the index n in (12.1.7) varies from $-N/2$ to $N/2$ (cf. 12.1.5). You can easily see, however, that (12.1.7) is periodic in n, with period N. Therefore, $H_{-n} = H_{N-n}$ $n = 1, 2, \ldots$. With this conversion in mind, one generally lets the n in H_n vary from 0 to $N - 1$ (one complete period). Then n and k (in h_k) vary exactly over the same range, so the mapping of N numbers into N numbers is manifest. When this convention is followed, you must remember that zero frequency corresponds to $n = 0$, positive frequencies $0 < f < f_c$ correspond to values $1 \le n \le N/2 - 1$, while negative frequencies $-f_c < f < 0$ correspond to $N/2 + 1 \le n \le N - 1$. The value $n = N/2$ corresponds to *both* $f = f_c$ and $f = -f_c$.

The discrete Fourier transform has symmetry properties almost exactly the same as the continuous Fourier transform. For example, all the symmetries in the table following equation (12.0.3) hold if we read h_k for $h(t)$, H_n for $H(f)$, and H_{N-n} for $H(-f)$. (Likewise, "even" and "odd" in time refer to whether the values h_k at k and $N - k$ are identical or the negative of each other.)

The formula for the discrete *inverse* Fourier transform, which recovers the set of h_k's exactly from the H_n's is:

$$h_k = \frac{1}{N} \sum_{n=0}^{N-1} H_n\, e^{-2\pi i kn/N} \qquad (12.1.9)$$

Notice that the only differences between (12.1.9) and (12.1.7) are (i) changing the sign in the exponential, and (ii) dividing the answer by N. This means that a routine for calculating discrete Fourier transforms can also, with slight modification, calculate the inverse transforms.

The discrete form of Parseval's theorem is

$$\sum_{k=0}^{N-1} |h_k|^2 = \frac{1}{N} \sum_{n=0}^{N-1} |H_n|^2 \tag{12.1.10}$$

There are also discrete analogs to the convolution and correlation theorems (equations 12.0.9 and 12.0.11), but we shall defer them to §13.1 and §13.2, respectively.

CITED REFERENCES AND FURTHER READING:

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall).

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press).

12.2 Fast Fourier Transform (FFT)

How much computation is involved in computing the discrete Fourier transform (12.1.7) of N points? For many years, until the mid-1960s, the standard answer was this: Define W as the complex number

$$W \equiv e^{2\pi i/N} \tag{12.2.1}$$

Then (12.1.7) can be written as

$$H_n = \sum_{k=0}^{N-1} W^{nk} h_k \tag{12.2.2}$$

In other words, the vector of h_k's is multiplied by a matrix whose (n,k)th element is the constant W to the power $n \times k$. The matrix multiplication produces a vector result whose components are the H_n's. This matrix multiplication evidently requires N^2 complex multiplications, plus a smaller number of operations to generate the required powers of W. So, the discrete Fourier transform appears to be an $O(N^2)$ process. These appearances are deceiving! The discrete Fourier transform can, in fact, be computed in $O(N \log_2 N)$ operations with an algorithm called the *fast Fourier transform*, or *FFT*. The difference between $N \log_2 N$ and N^2 is immense. With $N = 10^6$, for example, it is the difference between, roughly, 30 seconds of CPU time and 2 weeks of CPU time on a microsecond cycle time computer. The existence of an FFT algorithm became generally known only in the mid-1960s, from the work of J.W. Cooley and J.W. Tukey. Retrospectively, we now know (see [1]) that efficient methods for computing the DFT had been independently discovered, and in some cases implemented, by as many as a dozen individuals, starting with Gauss in 1805!

One "rediscovery" of the FFT, that of Danielson and Lanczos in 1942, provides one of the clearest derivations of the algorithm. Danielson and Lanczos showed that a discrete Fourier transform of length N can be rewritten as the sum of two discrete Fourier transforms, each of length $N/2$. One of the two is formed from the

The discrete form of Parseval's theorem is

$$\sum_{k=0}^{N-1} |h_k|^2 = \frac{1}{N} \sum_{n=0}^{N-1} |H_n|^2 \tag{12.1.10}$$

There are also discrete analogs to the convolution and correlation theorems (equations 12.0.9 and 12.0.11), but we shall defer them to §13.1 and §13.2, respectively.

CITED REFERENCES AND FURTHER READING:

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall).

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press).

12.2 Fast Fourier Transform (FFT)

How much computation is involved in computing the discrete Fourier transform (12.1.7) of N points? For many years, until the mid-1960s, the standard answer was this: Define W as the complex number

$$W \equiv e^{2\pi i/N} \tag{12.2.1}$$

Then (12.1.7) can be written as

$$H_n = \sum_{k=0}^{N-1} W^{nk} h_k \tag{12.2.2}$$

In other words, the vector of h_k's is multiplied by a matrix whose (n, k)th element is the constant W to the power $n \times k$. The matrix multiplication produces a vector result whose components are the H_n's. This matrix multiplication evidently requires N^2 complex multiplications, plus a smaller number of operations to generate the required powers of W. So, the discrete Fourier transform appears to be an $O(N^2)$ process. These appearances are deceiving! The discrete Fourier transform can, in fact, be computed in $O(N \log_2 N)$ operations with an algorithm called the *fast Fourier transform*, or *FFT*. The difference between $N \log_2 N$ and N^2 is immense. With $N = 10^6$, for example, it is the difference between, roughly, 30 seconds of CPU time and 2 weeks of CPU time on a microsecond cycle time computer. The existence of an FFT algorithm became generally known only in the mid-1960s, from the work of J.W. Cooley and J.W. Tukey. Retrospectively, we now know (see [1]) that efficient methods for computing the DFT had been independently discovered, and in some cases implemented, by as many as a dozen individuals, starting with Gauss in 1805!

One "rediscovery" of the FFT, that of Danielson and Lanczos in 1942, provides one of the clearest derivations of the algorithm. Danielson and Lanczos showed that a discrete Fourier transform of length N can be rewritten as the sum of two discrete Fourier transforms, each of length $N/2$. One of the two is formed from the

even-numbered points of the original N, the other from the odd-numbered points. The proof is simply this:

$$
\begin{aligned}
F_k &= \sum_{j=0}^{N-1} e^{2\pi ijk/N} f_j \\
&= \sum_{j=0}^{N/2-1} e^{2\pi ik(2j)/N} f_{2j} + \sum_{j=0}^{N/2-1} e^{2\pi ik(2j+1)/N} f_{2j+1} \\
&= \sum_{j=0}^{N/2-1} e^{2\pi ikj/(N/2)} f_{2j} + W^k \sum_{j=0}^{N/2-1} e^{2\pi ikj/(N/2)} f_{2j+1} \\
&= F_k^e + W^k\, F_k^o
\end{aligned}
\qquad (12.2.3)
$$

In the last line, W is the same complex constant as in (12.2.1), F_k^e denotes the kth component of the Fourier transform of length $N/2$ formed from the even components of the original f_j's, while F_k^o is the corresponding transform of length $N/2$ formed from the odd components. Notice also that k in the last line of (12.2.3) varies from 0 to N, not just to $N/2$. Nevertheless, the transforms F_k^e and F_k^o are periodic in k with length $N/2$. So each is repeated through two cycles to obtain F_k.

The wonderful thing about the *Danielson-Lanczos Lemma* is that it can be used recursively. Having reduced the problem of computing F_k to that of computing F_k^e and F_k^o, we can do the same reduction of F_k^e to the problem of computing the transform of *its* $N/4$ even-numbered input data and $N/4$ odd-numbered data. In other words, we can define F_k^{ee} and F_k^{eo} to be the discrete Fourier transforms of the points which are respectively even-even and even-odd on the successive subdivisions of the data.

Although there are ways of treating other cases, by far the easiest case is the one in which the original N is an integer power of 2. In fact, we categorically recommend that you *only* use FFTs with N a power of two. If the length of your data set is not a power of two, pad it with zeros up to the next power of two. (We will give more sophisticated suggestions in subsequent sections below.) With this restriction on N, it is evident that we can continue applying the Danielson-Lanczos Lemma until we have subdivided the data all the way down to transforms of length 1. What is the Fourier transform of length one? It is just the identity operation that copies its one input number into its one output slot! In other words, for every pattern of $\log_2 N$ e's and o's, there is a one-point transform that is just one of the input numbers f_n

$$
F_k^{eoeeoeo\cdots oee} = f_n \qquad \text{for some } n \qquad (12.2.4)
$$

(Of course this one-point transform actually does not depend on k, since it is periodic in k with period 1.)

The next trick is to figure out which value of n corresponds to which pattern of e's and o's in equation (12.2.4). The answer is: Reverse the pattern of e's and o's, then let $e = 0$ and $o = 1$, and you will have, *in binary* the value of n. Do you see why it works? It is because the successive subdivisions of the data into even and odd are tests of successive low-order (least significant) bits of n. This idea of *bit reversal* can be exploited in a very clever way which, along with the Danielson-Lanczos
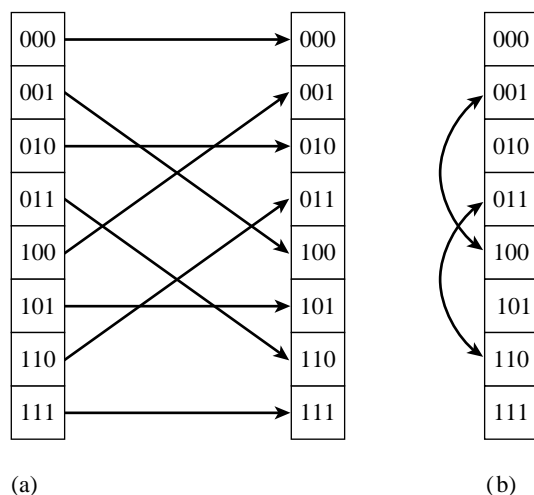
Figure 12.2.1.   Reordering an array (here of length 8) by bit reversal, (a) between two arrays, versus (b) in place. Bit reversal reordering is a necessary part of the fast Fourier transform (FFT) algorithm.

Lemma, makes FFTs practical: Suppose we take the original vector of data f_j and rearrange it into bit-reversed order (see Figure 12.2.1), so that the individual numbers are in the order not of j, but of the number obtained by bit-reversing j. Then the bookkeeping on the recursive application of the Danielson-Lanczos Lemma becomes extraordinarily simple. The points as given are the one-point transforms. We combine adjacent pairs to get two-point transforms, then combine adjacent pairs of pairs to get 4-point transforms, and so on, until the first and second halves of the whole data set are combined into the final transform. Each combination takes of order N operations, and there are evidently $\log_2 N$ combinations, so the whole algorithm is of order $N \log_2 N$ (assuming, as is the case, that the process of sorting into bit-reversed order is no greater in order than $N \log_2 N$).

This, then, is the structure of an FFT algorithm: It has two sections. The first section sorts the data into bit-reversed order. Luckily this takes no additional storage, since it involves only swapping pairs of elements. (If k_1 is the bit reverse of k_2, then k_2 is the bit reverse of k_1.) The second section has an outer loop that is executed $\log_2 N$ times and calculates, in turn, transforms of length $2, 4, 8, \ldots, N$. For each stage of this process, two nested inner loops range over the subtransforms already computed and the elements of each transform, implementing the Danielson-Lanczos Lemma. The operation is made more efficient by restricting external calls for trigonometric sines and cosines to the outer loop, where they are made only $\log_2 N$ times. Computation of the sines and cosines of multiple angles is through simple recurrence relations in the inner loops (cf. 5.5.6).

The FFT routine given below is based on one originally written by N. M. Brenner. The input quantities are the number of complex data points (nn), the data array (data[1..2*nn]), and isign, which should be set to either ± 1 and is the sign of i in the exponential of equation (12.1.7). When isign is set to -1, the routine thus calculates the inverse transform (12.1.9) — except that it does not multiply by the normalizing factor $1/N$ that appears in that equation. You can do that yourself.

Notice that the argument nn is the number of *complex* data points. The actual

length of the real array (`data[1..2*nn]`) is 2 times `nn`, with each complex value occupying two consecutive locations. In other words, `data[1]` is the real part of f_0, `data[2]` is the imaginary part of f_0, and so on up to `data[2*nn-1]`, which is the real part of f_{N-1}, and `data[2*nn]`, which is the imaginary part of f_{N-1}. The FFT routine gives back the F_n's packed in exactly the same fashion, as `nn` complex numbers.

The real and imaginary parts of the zero frequency component F_0 are in `data[1]` and `data[2]`; the smallest nonzero positive frequency has real and imaginary parts in `data[3]` and `data[4]`; the smallest (in magnitude) nonzero negative frequency has real and imaginary parts in `data[2*nn-1]` and `data[2*nn]`. Positive frequencies increasing in magnitude are stored in the real-imaginary pairs `data[5]`, `data[6]` up to `data[nn-1]`, `data[nn]`. Negative frequencies of increasing magnitude are stored in `data[2*nn-3]`, `data[2*nn-2]` down to `data[nn+3]`, `data[nn+4]`. Finally, the pair `data[nn+1]`, `data[nn+2]` contain the real and imaginary parts of the one aliased point that contains the most positive and the most negative frequency. You should try to develop a familiarity with this storage arrangement of complex spectra, also shown in Figure 12.2.2, since it is the practical standard.

This is a good place to remind you that you can also use a routine like `four1` *without modification* even if your input data array is zero-offset, that is has the range `data[0..2*nn-1]`. In this case, simply decrement the pointer to `data` by one when `four1` is invoked, e.g., `four1(data-1,1024,1);`. The real part of f_0 will now be returned in `data[0]`, the imaginary part in `data[1]`, and so on. See §1.2.

```c
#include <math.h>
#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr

void four1(float data[], unsigned long nn, int isign)
```
Replaces `data[1..2*nn]` by its discrete Fourier transform, if `isign` is input as 1; or replaces `data[1..2*nn]` by `nn` times its inverse discrete Fourier transform, if `isign` is input as −1. `data` is a complex array of length `nn` or, equivalently, a real array of length 2*`nn`. `nn` MUST be an integer power of 2 (this is not checked for!).
```c
{
    unsigned long n,mmax,m,j,istep,i;
    double wtemp,wr,wpr,wpi,wi,theta;        Double precision for the trigonomet-
    float tempr,tempi;                             ric recurrences.

    n=nn << 1;
    j=1;
    for (i=1;i<n;i+=2) {                     This is the bit-reversal section of the
        if (j > i) {                                routine.
            SWAP(data[j],data[i]);           Exchange the two complex numbers.
            SWAP(data[j+1],data[i+1]);
        }
        m=n >> 1;
        while (m >= 2 && j > m) {
            j -= m;
            m >>= 1;
        }
        j += m;
    }
    Here begins the Danielson-Lanczos section of the routine.
    mmax=2;
    while (n > mmax) {                       Outer loop executed log₂ nn times.
        istep=mmax << 1;
        theta=isign*(6.28318530717959/mmax); Initialize the trigonometric recurrence.
        wtemp=sin(0.5*theta);
        wpr = -2.0*wtemp*wtemp;
```

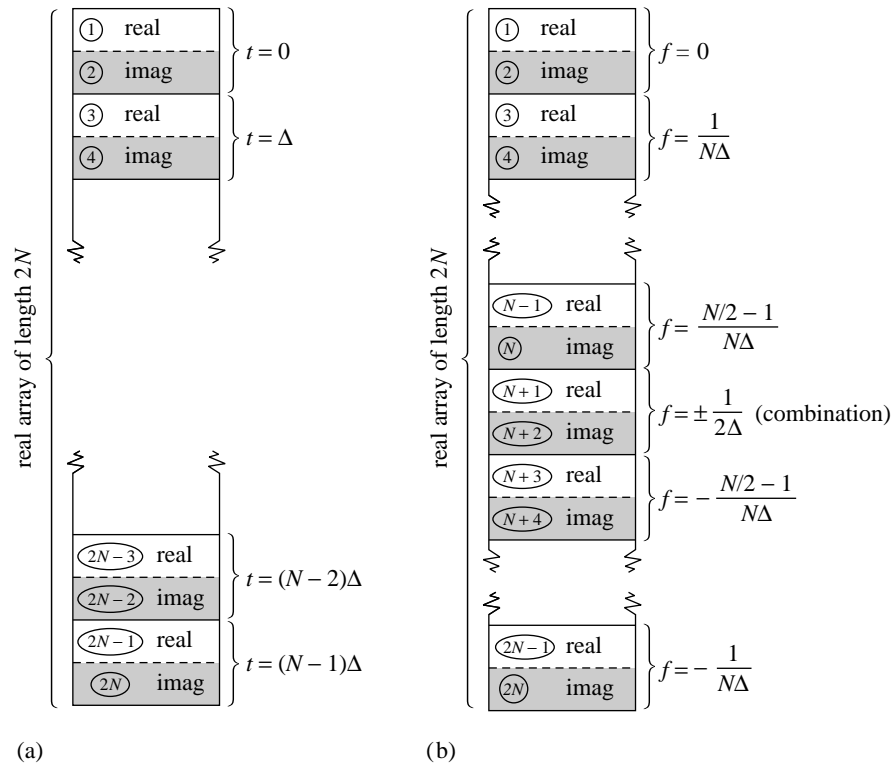(a)   (b)

Figure 12.2.2.   Input and output arrays for FFT. (a) The input array contains N (a power of 2) complex time samples in a real array of length $2N$, with real and imaginary parts alternating. (b) The output array contains the complex Fourier spectrum at N values of frequency. Real and imaginary parts again alternate. The array starts with zero frequency, works up to the most positive frequency (which is ambiguous with the most negative frequency). Negative frequencies follow, from the second-most negative up to the frequency just below zero.

```
        wpi=sin(theta);
        wr=1.0;
        wi=0.0;
        for (m=1;m<mmax;m+=2) {                    Here are the two nested inner loops.
            for (i=m;i<=n;i+=istep) {
                j=i+mmax;                          This is the Danielson-Lanczos for-
                tempr=wr*data[j]-wi*data[j+1];         mula:
                tempi=wr*data[j+1]+wi*data[j];
                data[j]=data[i]-tempr;
                data[j+1]=data[i+1]-tempi;
                data[i] += tempr;
                data[i+1] += tempi;
            }
            wr=(wtemp=wr)*wpr-wi*wpi+wr;          Trigonometric recurrence.
            wi=wi*wpr+wtemp*wpi+wi;
        }
        mmax=istep;
    }
}
```

(A double precision version of `four1`, named `dfour1`, is used by the routine `mpmul` in §20.6. You can easily make the conversion, or else get the converted routine from the *Numerical Recipes* diskette.)

Other FFT Algorithms

We should mention that there are a number of variants on the basic FFT algorithm given above. As we have seen, that algorithm first rearranges the input elements into bit-reverse order, then builds up the output transform in $\log_2 N$ iterations. In the literature, this sequence is called a *decimation-in-time* or *Cooley-Tukey* FFT algorithm. It is also possible to derive FFT algorithms that first go through a set of $\log_2 N$ iterations on the input data, and rearrange the *output* values into bit-reverse order. These are called *decimation-in-frequency* or *Sande-Tukey* FFT algorithms. For some applications, such as convolution (§13.1), one takes a data set into the Fourier domain and then, after some manipulation, back out again. In these cases it is possible to avoid all bit reversing. You use a decimation-in-frequency algorithm (without its bit reversing) to get into the "scrambled" Fourier domain, do your operations there, and then use an inverse algorithm (without *its* bit reversing) to get back to the time domain. While elegant in principle, this procedure does not in practice save much computation time, since the bit reversals represent only a small fraction of an FFT's operations count, and since most useful operations in the frequency domain require a knowledge of which points correspond to which frequencies.

Another class of FFTs subdivides the initial data set of length N not all the way down to the trivial transform of length 1, but rather only down to some other small power of 2, for example $N = 4$, *base-4 FFTs*, or $N = 8$, *base-8 FFTs*. These small transforms are then done by small sections of highly optimized coding which take advantage of special symmetries of that particular small N. For example, for $N = 4$, the trigonometric sines and cosines that enter are all ± 1 or 0, so many multiplications are eliminated, leaving largely additions and subtractions. These can be faster than simpler FFTs by some significant, but not overwhelming, factor, e.g., 20 or 30 percent.

There are also FFT algorithms for data sets of length N not a power of two. They work by using relations analogous to the Danielson-Lanczos Lemma to subdivide the initial problem into successively smaller problems, not by factors of 2, but by whatever small prime factors happen to divide N. The larger that the largest prime factor of N is, the worse this method works. If N is prime, then no subdivision is possible, and the user (whether he knows it or not) is taking a *slow* Fourier transform, of order N^2 instead of order $N \log_2 N$. Our advice is to stay clear of such FFT implementations, with perhaps one class of exceptions, the *Winograd Fourier transform algorithms*. Winograd algorithms are in some ways analogous to the base-4 and base-8 FFTs. Winograd has derived highly optimized codings for taking small-N discrete Fourier transforms, e.g., for $N = 2, 3, 4, 5, 7, 8, 11, 13, 16$. The algorithms also use a new and clever way of combining the subfactors. The method involves a reordering of the data both before the hierarchical processing and after it, but it allows a significant reduction in the number of multiplications in the algorithm. For some especially favorable values of N, the Winograd algorithms can be significantly (e.g., up to a factor of 2) faster than the simpler FFT algorithms of the nearest integer power of 2. This advantage in speed, however, must be weighed against the considerably more complicated data indexing involved in these transforms, and the fact that the Winograd transform cannot be done "in place."

Finally, an interesting class of transforms for doing convolutions quickly are number theoretic transforms. These schemes replace floating-point arithmetic with

integer arithmetic modulo some large prime $N+1$, and the Nth root of 1 by the modulo arithmetic equivalent. Strictly speaking, these are not *Fourier* transforms at all, but the properties are quite similar and computational speed can be far superior. On the other hand, their use is somewhat restricted to quantities like correlations and convolutions since the transform itself is not easily interpretable as a "frequency" spectrum.

CITED REFERENCES AND FURTHER READING:

Nussbaumer, H.J. 1982, *Fast Fourier Transform and Convolution Algorithms* (New York: Springer-Verlag).

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press).

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Bloomfield, P. 1976, *Fourier Analysis of Time Series – An Introduction* (New York: Wiley).

Van Loan, C. 1992, *Computational Frameworks for the Fast Fourier Transform* (Philadelphia: S.I.A.M.).

Beauchamp, K.G. 1984, *Applications of Walsh Functions and Related Functions* (New York: Academic Press) [non-Fourier transforms].

Heideman, M.T., Johnson, D.H., and Burris, C.S. 1984, *IEEE ASSP Magazine*, pp. 14–21 (October).

12.3 FFT of Real Functions, Sine and Cosine Transforms

It happens frequently that the data whose FFT is desired consist of real-valued samples f_j, $j = 0 \ldots N-1$. To use four1, we put these into a complex array with all imaginary parts set to zero. The resulting transform F_n, $n = 0 \ldots N-1$ satisfies $F_{N-n}{}^* = F_n$. Since this complex-valued array has real values for F_0 and $F_{N/2}$, and $(N/2) - 1$ other independent values $F_1 \ldots F_{N/2-1}$, it has the same $2(N/2 - 1) + 2 = N$ "degrees of freedom" as the original, real data set. However, the use of the full complex FFT algorithm for real data is inefficient, both in execution time and in storage required. You would think that there is a better way.

There are *two* better ways. The first is "mass production": Pack two separate real functions into the input array in such a way that their individual transforms can be separated from the result. This is implemented in the program twofft below. This may remind you of a one-cent sale, at which you are coerced to purchase two of an item when you only need one. However, remember that for correlations and convolutions the Fourier transforms of two functions are involved, and this is a handy way to do them both at once. The second method is to pack the real input array cleverly, without extra zeros, into a complex array of half its length. One then performs a complex FFT on this shorter length; the trick is then to get the required answer out of the result. This is done in the program realft below.

integer arithmetic modulo some large prime $N+1$, and the Nth root of 1 by the modulo arithmetic equivalent. Strictly speaking, these are not *Fourier* transforms at all, but the properties are quite similar and computational speed can be far superior. On the other hand, their use is somewhat restricted to quantities like correlations and convolutions since the transform itself is not easily interpretable as a "frequency" spectrum.

CITED REFERENCES AND FURTHER READING:

Nussbaumer, H.J. 1982, *Fast Fourier Transform and Convolution Algorithms* (New York: Springer-Verlag).

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press).

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Bloomfield, P. 1976, *Fourier Analysis of Time Series – An Introduction* (New York: Wiley).

Van Loan, C. 1992, *Computational Frameworks for the Fast Fourier Transform* (Philadelphia: S.I.A.M.).

Beauchamp, K.G. 1984, *Applications of Walsh Functions and Related Functions* (New York: Academic Press) [non-Fourier transforms].

Heideman, M.T., Johnson, D.H., and Burris, C.S. 1984, *IEEE ASSP Magazine*, pp. 14–21 (October).

12.3 FFT of Real Functions, Sine and Cosine Transforms

It happens frequently that the data whose FFT is desired consist of real-valued samples f_j, $j = 0 \ldots N - 1$. To use four1, we put these into a complex array with all imaginary parts set to zero. The resulting transform F_n, $n = 0 \ldots N - 1$ satisfies $F_{N-n}{}^* = F_n$. Since this complex-valued array has real values for F_0 and $F_{N/2}$, and $(N/2) - 1$ other independent values $F_1 \ldots F_{N/2-1}$, it has the same $2(N/2 - 1) + 2 = N$ "degrees of freedom" as the original, real data set. However, the use of the full complex FFT algorithm for real data is inefficient, both in execution time and in storage required. You would think that there is a better way.

There are *two* better ways. The first is "mass production": Pack two separate real functions into the input array in such a way that their individual transforms can be separated from the result. This is implemented in the program twofft below. This may remind you of a one-cent sale, at which you are coerced to purchase two of an item when you only need one. However, remember that for correlations and convolutions the Fourier transforms of two functions are involved, and this is a handy way to do them both at once. The second method is to pack the real input array cleverly, without extra zeros, into a complex array of half its length. One then performs a complex FFT on this shorter length; the trick is then to get the required answer out of the result. This is done in the program realft below.

Transform of Two Real Functions Simultaneously

First we show how to exploit the symmetry of the transform F_n to handle two real functions at once: Since the input data f_j are real, the components of the discrete Fourier transform satisfy

$$F_{N-n} = (F_n)^* \tag{12.3.1}$$

where the asterisk denotes complex conjugation. By the same token, the discrete Fourier transform of a purely imaginary set of g_j's has the opposite symmetry.

$$G_{N-n} = -(G_n)^* \tag{12.3.2}$$

Therefore we can take the discrete Fourier transform of two real functions each of length N simultaneously by packing the two data arrays as the real and imaginary parts, respectively, of the complex input array of `four1`. Then the resulting transform array can be unpacked into two complex arrays with the aid of the two symmetries. Routine `twofft` works out these ideas.

```
void twofft(float data1[], float data2[], float fft1[], float fft2[],
    unsigned long n)
Given two real input arrays data1[1..n] and data2[1..n], this routine calls four1 and
returns two complex output arrays, fft1[1..2n] and fft2[1..2n], each of complex length
n (i.e., real length 2*n), which contain the discrete Fourier transforms of the respective data
arrays. n MUST be an integer power of 2.
{
    void four1(float data[], unsigned long nn, int isign);
    unsigned long nn3,nn2,jj,j;
    float rep,rem,aip,aim;

    nn3=1+(nn2=2+n+n);
    for (j=1,jj=2;j<=n;j++,jj+=2) {            Pack the two real arrays into one com-
        fft1[jj-1]=data1[j];                      plex array.
        fft1[jj]=data2[j];
    }
    four1(fft1,n,1);                           Transform the complex array.
    fft2[1]=fft1[2];
    fft1[2]=fft2[2]=0.0;
    for (j=3;j<=n+1;j+=2) {
        rep=0.5*(fft1[j]+fft1[nn2-j]);         Use symmetries to separate the two trans-
        rem=0.5*(fft1[j]-fft1[nn2-j]);             forms.
        aip=0.5*(fft1[j+1]+fft1[nn3-j]);
        aim=0.5*(fft1[j+1]-fft1[nn3-j]);
        fft1[j]=rep;                           Ship them out in two complex arrays.
        fft1[j+1]=aim;
        fft1[nn2-j]=rep;
        fft1[nn3-j] = -aim;
        fft2[j]=aip;
        fft2[j+1] = -rem;
        fft2[nn2-j]=aip;
        fft2[nn3-j]=rem;
    }
}
```

What about the reverse process? Suppose you have two complex transform arrays, each of which has the symmetry (12.3.1), so that you know that the inverses of both transforms are real functions. Can you invert both in a single FFT? This is even easier than the other direction. Use the fact that the FFT is linear and form the sum of the first transform plus i times the second. Invert using `four1` with `isign` $= -1$. The real and imaginary parts of the resulting complex array are the two desired real functions.

FFT of Single Real Function

To implement the second method, which allows us to perform the FFT of a *single* real function without redundancy, we split the data set in half, thereby forming two real arrays of half the size. We can apply the program above to these two, but of course the result will not be the transform of the original data. It will be a schizophrenic combination of two transforms, each of which has half of the information we need. Fortunately, this schizophrenia is treatable. It works like this:

The right way to split the original data is to take the even-numbered f_j as one data set, and the odd-numbered f_j as the other. The beauty of this is that we can take the original real array and treat it as a complex array h_j of half the length. The first data set is the real part of this array, and the second is the imaginary part, as prescribed for `twofft`. No repacking is required. In other words $h_j = f_{2j} + if_{2j+1}$, $j = 0, \ldots, N/2 - 1$. We submit this to `four1`, and it will give back a complex array $H_n = F_n^e + iF_n^o$, $n = 0, \ldots, N/2 - 1$ with

$$
\begin{aligned}
F_n^e &= \sum_{k=0}^{N/2-1} f_{2k}\, e^{2\pi i k n/(N/2)} \\
F_n^o &= \sum_{k=0}^{N/2-1} f_{2k+1}\, e^{2\pi i k n/(N/2)}
\end{aligned}
\tag{12.3.3}
$$

The discussion of program `twofft` tells you how to separate the two transforms F_n^e and F_n^o out of H_n. How do you work them into the transform F_n of the original data set f_j? Simply glance back at equation (12.2.3):

$$
F_n = F_n^e + e^{2\pi i n/N} F_n^o \qquad n = 0, \ldots, N - 1 \tag{12.3.4}
$$

Expressed directly in terms of the transform H_n of our real (masquerading as complex) data set, the result is

$$
F_n = \frac{1}{2}(H_n + H_{N/2-n}{}^*) - \frac{i}{2}(H_n - H_{N/2-n}{}^*)e^{2\pi i n/N} \qquad n = 0, \ldots, N - 1 \tag{12.3.5}
$$

A few remarks:

- Since $F_{N-n}{}^* = F_n$ there is no point in saving the entire spectrum. The positive frequency half is sufficient and can be stored in the same array as the original data. The operation can, in fact, be done in place.
- Even so, we need values H_n, $n = 0, \ldots, N/2$ whereas `four1` gives only the values $n = 0, \ldots, N/2 - 1$. Symmetry to the rescue, $H_{N/2} = H_0$.

- The values F_0 and $F_{N/2}$ are real and independent. In order to actually get the entire F_n in the original array space, it is convenient to put $F_{N/2}$ into the imaginary part of F_0.

- Despite its complicated form, the process above is invertible. First peel $F_{N/2}$ out of F_0. Then construct

$$
\begin{aligned}
F_n^e &= \frac{1}{2}(F_n + F_{N/2-n}^*) \\
F_n^o &= \frac{1}{2}e^{-2\pi i n/N}(F_n - F_{N/2-n}^*)
\end{aligned}
\qquad n = 0, \ldots, N/2 - 1 \quad (12.3.6)
$$

and use `four1` to find the inverse transform of $H_n = F_n^{(1)} + iF_n^{(2)}$. Surprisingly, the actual algebraic steps are virtually identical to those of the forward transform.

Here is a representation of what we have said:

```
#include <math.h>

void realft(float data[], unsigned long n, int isign)
```
Calculates the Fourier transform of a set of n real-valued data points. Replaces this data (which is stored in array `data[1..n]`) by the positive frequency half of its complex Fourier transform. The real-valued first and last components of the complex transform are returned as elements `data[1]` and `data[2]`, respectively. n must be a power of 2. This routine also calculates the inverse transform of a complex data array if it is the transform of real data. (Result in this case must be multiplied by 2/n.)
```
{
    void four1(float data[], unsigned long nn, int isign);
    unsigned long i,i1,i2,i3,i4,np3;
    float c1=0.5,c2,h1r,h1i,h2r,h2i;
    double wr,wi,wpr,wpi,wtemp,theta;            Double precision for the trigonomet-
                                                     ric recurrences.
    theta=3.141592653589793/(double) (n>>1);     Initialize the recurrence.
    if (isign == 1) {
        c2 = -0.5;
        four1(data,n>>1,1);                      The forward transform is here.
    } else {
        c2=0.5;                                  Otherwise set up for an inverse trans-
        theta = -theta;                              form.
    }
    wtemp=sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi=sin(theta);
    wr=1.0+wpr;
    wi=wpi;
    np3=n+3;
    for (i=2;i<=(n>>2);i++) {                     Case i=1 done separately below.
        i4=1+(i3=np3-(i2=1+(i1=i+i-1)));
        h1r=c1*(data[i1]+data[i3]);              The two separate transforms are sep-
        h1i=c1*(data[i2]-data[i4]);                  arated out of data.
        h2r = -c2*(data[i2]+data[i4]);
        h2i=c2*(data[i1]-data[i3]);
        data[i1]=h1r+wr*h2r-wi*h2i;              Here they are recombined to form
        data[i2]=h1i+wr*h2i+wi*h2r;                  the true transform of the origi-
        data[i3]=h1r-wr*h2r+wi*h2i;                  nal real data.
        data[i4] = -h1i+wr*h2i+wi*h2r;
        wr=(wtemp=wr)*wpr-wi*wpi+wr;             The recurrence.
        wi=wi*wpr+wtemp*wpi+wi;
    }
    if (isign == 1) {
```

```
        data[1] = (h1r=data[1])+data[2];              Squeeze the first and last data to-
        data[2] = h1r-data[2];                            gether to get them all within the
    } else {                                              original array.
        data[1]=c1*((h1r=data[1])+data[2]);
        data[2]=c1*(h1r-data[2]);
        four1(data,n>>1,-1);                          This is the inverse transform for the
    }                                                     case isign=-1.
}
```

Fast Sine and Cosine Transforms

Among their other uses, the Fourier transforms of functions can be used to solve differential equations (see §19.4). The most common boundary conditions for the solutions are 1) they have the value zero at the boundaries, or 2) their derivatives are zero at the boundaries. In the first instance, the natural transform to use is the *sine* transform, given by

$$F_k = \sum_{j=1}^{N-1} f_j \sin(\pi jk/N) \qquad \text{sine transform} \qquad (12.3.7)$$

where f_j, $j = 0, \ldots, N - 1$ is the data array, and $f_0 \equiv 0$.

At first blush this appears to be simply the imaginary part of the discrete Fourier transform. However, the argument of the sine differs by a factor of two from the value that would make this so. The sine transform uses *sines only* as a complete set of functions in the interval from 0 to 2π, and, as we shall see, the cosine transform uses *cosines only*. By contrast, the normal FFT uses both sines and cosines, but only half as many of each. (See Figure 12.3.1.)

The expression (12.3.7) can be "force-fit" into a form that allows its calculation via the FFT. The idea is to extend the given function rightward past its last tabulated value. We extend the data to twice their length in such a way as to make them an *odd* function about $j = N$, with $f_N = 0$,

$$f_{2N-j} \equiv -f_j \qquad j = 0, \ldots, N - 1 \qquad (12.3.8)$$

Consider the FFT of this extended function:

$$F_k = \sum_{j=0}^{2N-1} f_j e^{2\pi ijk/(2N)} \qquad (12.3.9)$$

The half of this sum from $j = N$ to $j = 2N - 1$ can be rewritten with the substitution $j' = 2N - j$

$$\sum_{j=N}^{2N-1} f_j e^{2\pi ijk/(2N)} = \sum_{j'=1}^{N} f_{2N-j'} e^{2\pi i(2N-j')k/(2N)}$$

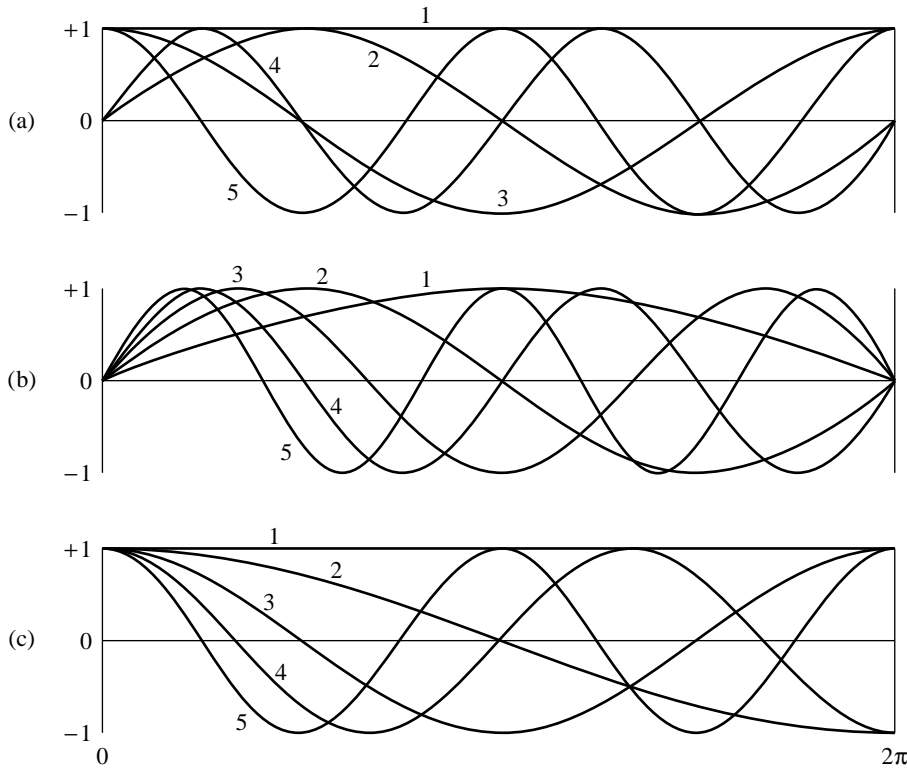$$= -\sum_{j'=0}^{N-1} f_{j'} e^{-2\pi ij'k/(2N)} \qquad (12.3.10)$$

Figure 12.3.1. Basis functions used by the Fourier transform (a), sine transform (b), and cosine transform (c), are plotted. The first five basis functions are shown in each case. (For the Fourier transform, the real and imaginary parts of the basis functions are both shown.) While some basis functions occur in more than one transform, the basis sets are distinct. For example, the sine transform functions labeled (1), (3), (5) are not present in the Fourier basis. Any of the three sets can expand any function in the interval shown; however, the sine or cosine transform best expands functions matching the boundary conditions of the respective basis functions, namely zero function values for sine, zero derivatives for cosine.

so that

$$F_k = \sum_{j=0}^{N-1} f_j \left[e^{2\pi ijk/(2N)} - e^{-2\pi ijk/(2N)} \right]$$

$$= 2i \sum_{j=0}^{N-1} f_j \sin(\pi jk/N) \qquad (12.3.11)$$

Thus, up to a factor $2i$ we get the sine transform from the FFT of the extended function.

This method introduces a factor of two inefficiency into the computation by extending the data. This inefficiency shows up in the FFT output, which has zeros for the real part of every element of the transform. For a one-dimensional problem, the factor of two may be bearable, especially in view of the simplicity of the method. When we work with partial differential equations in two or three dimensions, though, the factor becomes four or eight, so efforts to eliminate the inefficiency are well rewarded.

From the original real data array f_j we will construct an auxiliary array y_j and apply to it the routine realft. The output will then be used to construct the desired transform. For the sine transform of data f_j, $j = 1, \ldots, N - 1$, the auxiliary array is

$$y_0 = 0$$

$$y_j = \sin(j\pi/N)(f_j + f_{N-j}) + \frac{1}{2}(f_j - f_{N-j}) \qquad j = 1, \ldots, N - 1 \tag{12.3.12}$$

This array is of the same dimension as the original. Notice that the first term is symmetric about $j = N/2$ and the second is antisymmetric. Consequently, when realft is applied to y_j, the result has real parts R_k and imaginary parts I_k given by

$$
\begin{aligned}
R_k &= \sum_{j=0}^{N-1} y_j \cos(2\pi jk/N) \\
&= \sum_{j=1}^{N-1} (f_j + f_{N-j}) \sin(j\pi/N) \cos(2\pi jk/N) \\
&= \sum_{j=0}^{N-1} 2f_j \sin(j\pi/N) \cos(2\pi jk/N) \\
&= \sum_{j=0}^{N-1} f_j \left[\sin\frac{(2k+1)j\pi}{N} - \sin\frac{(2k-1)j\pi}{N} \right] \\
&= F_{2k+1} - F_{2k-1} \tag{12.3.13}
\end{aligned}
$$

$$
\begin{aligned}
I_k &= \sum_{j=0}^{N-1} y_j \sin(2\pi jk/N) \\
&= \sum_{j=1}^{N-1} (f_j - f_{N-j})\frac{1}{2} \sin(2\pi jk/N) \\
&= \sum_{j=0}^{N-1} f_j \sin(2\pi jk/N) \\
&= F_{2k} \tag{12.3.14}
\end{aligned}
$$

Therefore F_k can be determined as follows:

$$F_{2k} = I_k \qquad F_{2k+1} = F_{2k-1} + R_k \qquad k = 0, \ldots, (N/2 - 1) \tag{12.3.15}$$

The even terms of F_k are thus determined very directly. The odd terms require a recursion, the starting point of which follows from setting $k = 0$ in equation (12.3.15) and using $F_1 = -F_{-1}$:

$$F_1 = \frac{1}{2}R_0 \tag{12.3.16}$$

The implementing program is

```
#include <math.h>

void sinft(float y[], int n)
Calculates the sine transform of a set of n real-valued data points stored in array y[1..n].
The number n must be a power of 2. On exit y is replaced by its transform. This program,
without changes, also calculates the inverse sine transform, but in this case the output array
should be multiplied by 2/n.
{
    void realft(float data[], unsigned long n, int isign);
    int j,n2=n+2;
    float sum,y1,y2;
    double theta,wi=0.0,wr=1.0,wpi,wpr,wtemp;       Double precision in the trigono-
                                                       metric recurrences.
    theta=3.14159265358979/(double) n;              Initialize the recurrence.
    wtemp=sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi=sin(theta);
    y[1]=0.0;
    for (j=2;j<=(n>>1)+1;j++) {
        wr=(wtemp=wr)*wpr-wi*wpi+wr;                Calculate the sine for the auxiliary array.
        wi=wi*wpr+wtemp*wpi+wi;                     The cosine is needed to continue the recurrence.
        y1=wi*(y[j]+y[n2-j]);                       Construct the auxiliary array.
        y2=0.5*(y[j]-y[n2-j]);
        y[j]=y1+y2;                                 Terms j and N − j are related.
        y[n2-j]=y1-y2;
    }
    realft(y,n,1);                                  Transform the auxiliary array.
    y[1]*=0.5;                                      Initialize the sum used for odd terms below.
    sum=y[2]=0.0;
    for (j=1;j<=n-1;j+=2) {
        sum += y[j];
        y[j]=y[j+1];                                Even terms determined directly.
        y[j+1]=sum;                                 Odd terms determined by this running sum.
    }
}
```

The sine transform, curiously, is its own inverse. If you apply it twice, you get the original data, but multiplied by a factor of $N/2$.

The other common boundary condition for differential equations is that the derivative of the function is zero at the boundary. In this case the natural transform is the *cosine* transform. There are several possible ways of defining the transform. Each can be thought of as resulting from a different way of extending a given array to create an even array of double the length, and/or from whether the extended array contains $2N - 1$, $2N$, or some other number of points. In practice, only two of the numerous possibilities are useful so we will restrict ourselves to just these two.

The first form of the cosine transform uses $N + 1$ data points:

$$F_k = \frac{1}{2}[f_0 + (-1)^k f_N] + \sum_{j=1}^{N-1} f_j \cos(\pi jk/N) \tag{12.3.17}$$

It results from extending the given array to an even array about $j = N$, with

$$f_{2N-j} = f_j, \qquad j = 0, \dots, N-1 \tag{12.3.18}$$

If you substitute this extended array into equation (12.3.9), and follow steps analogous to those leading up to equation (12.3.11), you will find that the Fourier transform is

just twice the cosine transform (12.3.17). Another way of thinking about the formula (12.3.17) is to notice that it is the Chebyshev Gauss-Lobatto quadrature formula (see §4.5), often used in Clenshaw-Curtis adaptive quadrature (see §5.9, equation 5.9.4).

Once again the transform can be computed without the factor of two inefficiency. In this case the auxiliary function is

$$y_j = \frac{1}{2}(f_j + f_{N-j}) - \sin(j\pi/N)(f_j - f_{N-j}) \qquad j = 0, \ldots, N-1 \quad (12.3.19)$$

Instead of equation (12.3.15), `realft` now gives

$$F_{2k} = R_k \qquad F_{2k+1} = F_{2k-1} + I_k \qquad k = 0, \ldots, (N/2 - 1) \qquad (12.3.20)$$

The starting value for the recursion for odd k in this case is

$$F_1 = \frac{1}{2}(f_0 - f_N) + \sum_{j=1}^{N-1} f_j \cos(j\pi/N) \qquad (12.3.21)$$

This sum does not appear naturally among the R_k and I_k, and so we accumulate it during the generation of the array y_j.

Once again this transform is its own inverse, and so the following routine works for both directions of the transformation. Note that although this form of the cosine transform has $N + 1$ input and output values, it passes an array only of length N to `realft`.

```
#include <math.h>
#define PI 3.141592653589793

void cosft1(float y[], int n)
```
Calculates the cosine transform of a set `y[1..n+1]` of real-valued data points. The transformed data replace the original data in array `y`. `n` must be a power of 2. This program, without changes, also calculates the inverse cosine transform, but in this case the output array should be multiplied by `2/n`.
```
{
    void realft(float data[], unsigned long n, int isign);
    int j,n2;
    float sum,y1,y2;
    double theta,wi=0.0,wpi,wpr,wr=1.0,wtemp;
    Double precision for the trigonometric recurrences.

    theta=PI/n;                             Initialize the recurrence.
    wtemp=sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi=sin(theta);
    sum=0.5*(y[1]-y[n+1]);
    y[1]=0.5*(y[1]+y[n+1]);
    n2=n+2;
    for (j=2;j<=(n>>1);j++) {               j=n/2+1 unnecessary since y[n/2+1] unchanged.
        wr=(wtemp=wr)*wpr-wi*wpi+wr;        Carry out the recurrence.
        wi=wi*wpr+wtemp*wpi+wi;
        y1=0.5*(y[j]+y[n2-j]);              Calculate the auxiliary function.
        y2=(y[j]-y[n2-j]);
        y[j]=y1-wi*y2;                      The values for j and N − j are related.
        y[n2-j]=y1+wi*y2;
        sum += wr*y2;                       Carry along this sum for later use in unfold-
    }                                           ing the transform.
```

```
    realft(y,n,1);                              Calculate the transform of the auxiliary func-
    y[n+1]=y[2];                                    tion.
    y[2]=sum;                                   sum is the value of F₁ in equation (12.3.21).
    for (j=4;j<=n;j+=2) {
        sum += y[j];                            Equation (12.3.20).
        y[j]=sum;
    }
}
```

The second important form of the cosine transform is defined by

$$F_k = \sum_{j=0}^{N-1} f_j \cos \frac{\pi k(j + \frac{1}{2})}{N} \tag{12.3.22}$$

with inverse

$$f_j = \frac{2}{N} \sum_{k=0}^{N-1}{}' F_k \cos \frac{\pi k(j + \frac{1}{2})}{N} \tag{12.3.23}$$

Here the prime on the summation symbol means that the term for $k = 0$ has a coefficient of $\frac{1}{2}$ in front. This form arises by extending the given data, defined for $j = 0, \ldots, N-1$, to $j = N, \ldots, 2N-1$ in such a way that it is even about the point $N - \frac{1}{2}$ and periodic. (It is therefore also even about $j = -\frac{1}{2}$.) The form (12.3.23) is related to Gauss-Chebyshev quadrature (see equation 4.5.19), to Chebyshev approximation (§5.8, equation 5.8.7), and Clenshaw-Curtis quadrature (§5.9).

This form of the cosine transform is useful when solving differential equations on "staggered" grids, where the variables are centered midway between mesh points. It is also the standard form in the field of data compression and image processing.

The auxiliary function used in this case is similar to equation (12.3.19):

$$y_j = \frac{1}{2}(f_j + f_{N-j-1}) - \sin \frac{\pi(j + \frac{1}{2})}{N}(f_j - f_{N-j-1}) \qquad j = 0, \ldots, N-1 \tag{12.3.24}$$

Carrying out the steps similar to those used to get from (12.3.12) to (12.3.15), we find

$$F_{2k} = \cos \frac{\pi k}{N} R_k - \sin \frac{\pi k}{N} I_k \tag{12.3.25}$$

$$F_{2k-1} = \sin \frac{\pi k}{N} R_k + \cos \frac{\pi k}{N} I_k + F_{2k+1} \tag{12.3.26}$$

Note that equation (12.3.26) gives

$$F_{N-1} = \frac{1}{2} R_{N/2} \tag{12.3.27}$$

Thus the even components are found directly from (12.3.25), while the odd components are found by recursing (12.3.26) down from $k = N/2 - 1$, using (12.3.27) to start.

Since the transform is not self-inverting, we have to reverse the above steps to find the inverse. Here is the routine:

```
#include <math.h>
#define PI 3.141592653589793

void cosft2(float y[], int n, int isign)
```
Calculates the "staggered" cosine transform of a set `y[1..n]` of real-valued data points. The transformed data replace the original data in array `y`. `n` must be a power of 2. Set `isign` to $+1$ for a transform, and to -1 for an inverse transform. For an inverse transform, the output array should be multiplied by `2/n`.
```
{
    void realft(float data[], unsigned long n, int isign);
    int i;
    float sum,sum1,y1,y2,ytemp;
    double theta,wi=0.0,wi1,wpi,wpr,wr=1.0,wr1,wtemp;
    Double precision for the trigonometric recurrences.

    theta=0.5*PI/n;                              Initialize the recurrences.
    wr1=cos(theta);
    wi1=sin(theta);
    wpr = -2.0*wi1*wi1;
    wpi=sin(2.0*theta);
    if (isign == 1) {                            Forward transform.
        for (i=1;i<=n/2;i++) {
            y1=0.5*(y[i]+y[n-i+1]);              Calculate the auxiliary function.
            y2=wi1*(y[i]-y[n-i+1]);
            y[i]=y1+y2;
            y[n-i+1]=y1-y2;
            wr1=(wtemp=wr1)*wpr-wi1*wpi+wr1;     Carry out the recurrence.
            wi1=wi1*wpr+wtemp*wpi+wi1;
        }
        realft(y,n,1);                           Transform the auxiliary function.
        for (i=3;i<=n;i+=2) {                    Even terms.
            wr=(wtemp=wr)*wpr-wi*wpi+wr;
            wi=wi*wpr+wtemp*wpi+wi;
            y1=y[i]*wr-y[i+1]*wi;
            y2=y[i+1]*wr+y[i]*wi;
            y[i]=y1;
            y[i+1]=y2;
        }
        sum=0.5*y[2];                            Initialize recurrence for odd terms
        for (i=n;i>=2;i-=2) {                        with ½R_{N/2}.
            sum1=sum;                            Carry out recurrence for odd terms.
            sum += y[i];
            y[i]=sum1;
        }
    } else if (isign == -1) {                    Inverse transform.
        ytemp=y[n];
        for (i=n;i>=4;i-=2) y[i]=y[i-2]-y[i];    Form difference of odd terms.
        y[2]=2.0*ytemp;
        for (i=3;i<=n;i+=2) {                    Calculate R_k and I_k.
            wr=(wtemp=wr)*wpr-wi*wpi+wr;
            wi=wi*wpr+wtemp*wpi+wi;
            y1=y[i]*wr+y[i+1]*wi;
            y2=y[i+1]*wr-y[i]*wi;
            y[i]=y1;
            y[i+1]=y2;
        }
        realft(y,n,-1);
        for (i=1;i<=n/2;i++) {                   Invert auxiliary array.
            y1=y[i]+y[n-i+1];
            y2=(0.5/wi1)*(y[i]-y[n-i+1]);
            y[i]=0.5*(y1+y2);
            y[n-i+1]=0.5*(y1-y2);
            wr1=(wtemp=wr1)*wpr-wi1*wpi+wr1;
            wi1=wi1*wpr+wtemp*wpi+wi1;
```

```
            }
        }
}
```

An alternative way of implementing this algorithm is to form an auxiliary function by copying the even elements of f_j into the first $N/2$ locations, and the odd elements into the next $N/2$ elements in reverse order. However, it is not easy to implement the alternative algorithm without a temporary storage array and we prefer the above in-place algorithm.

Finally, we mention that there exist fast cosine transforms for small N that do not rely on an auxiliary function or use an FFT routine. Instead, they carry out the transform directly, often coded in hardware for fixed N of small dimension [1].

CITED REFERENCES AND FURTHER READING:

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall), §10–10.

Sorensen, H.V., Jones, D.L., Heideman, M.T., and Burris, C.S. 1987, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-35, pp. 849–863.

Hou, H.S. 1987, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-35, pp. 1455–1461 [see for additional references].

Hockney, R.W. 1971, in *Methods in Computational Physics*, vol. 9 (New York: Academic Press).

Temperton, C. 1980, *Journal of Computational Physics*, vol. 34, pp. 314–329.

Clarke, R.J. 1985, *Transform Coding of Images*, (Reading, MA: Addison-Wesley).

Gonzalez, R.C., and Wintz, P. 1987, *Digital Image Processing*, (Reading, MA: Addison-Wesley).

Chen, W., Smith, C.H., and Fralick, S.C. 1977, *IEEE Transactions on Communications*, vol. COM-25, pp. 1004–1009. [1]

12.4 FFT in Two or More Dimensions

Given a complex function $h(k_1, k_2)$ defined over the two-dimensional grid $0 \le k_1 \le N_1 - 1$, $0 \le k_2 \le N_2 - 1$, we can define its two-dimensional discrete Fourier transform as a complex function $H(n_1, n_2)$, defined over the same grid,

$$H(n_1, n_2) \equiv \sum_{k_2=0}^{N_2-1} \sum_{k_1=0}^{N_1-1} \exp(2\pi i k_2 n_2 / N_2) \, \exp(2\pi i k_1 n_1 / N_1) \, h(k_1, k_2)$$

(12.4.1)

By pulling the "subscripts 2" exponential outside of the sum over k_1, or by reversing the order of summation and pulling the "subscripts 1" outside of the sum over k_2, we can see instantly that the two-dimensional FFT can be computed by taking one-dimensional FFTs sequentially on each index of the original function. Symbolically,

$$
\begin{aligned}
H(n_1, n_2) &= \text{FFT-on-index-1} \left(\text{FFT-on-index-2} \left[h(k_1, k_2) \right] \right) \\
&= \text{FFT-on-index-2} \left(\text{FFT-on-index-1} \left[h(k_1, k_2) \right] \right)
\end{aligned}
$$

(12.4.2)

```
        }
    }
}
```

An alternative way of implementing this algorithm is to form an auxiliary function by copying the even elements of f_j into the first $N/2$ locations, and the odd elements into the next $N/2$ elements in reverse order. However, it is not easy to implement the alternative algorithm without a temporary storage array and we prefer the above in-place algorithm.

Finally, we mention that there exist fast cosine transforms for small N that do not rely on an auxiliary function or use an FFT routine. Instead, they carry out the transform directly, often coded in hardware for fixed N of small dimension [1].

CITED REFERENCES AND FURTHER READING:

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall), §10–10.

Sorensen, H.V., Jones, D.L., Heideman, M.T., and Burris, C.S. 1987, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-35, pp. 849–863.

Hou, H.S. 1987, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-35, pp. 1455–1461 [see for additional references].

Hockney, R.W. 1971, in *Methods in Computational Physics*, vol. 9 (New York: Academic Press).

Temperton, C. 1980, *Journal of Computational Physics*, vol. 34, pp. 314–329.

Clarke, R.J. 1985, *Transform Coding of Images*, (Reading, MA: Addison-Wesley).

Gonzalez, R.C., and Wintz, P. 1987, *Digital Image Processing*, (Reading, MA: Addison-Wesley).

Chen, W., Smith, C.H., and Fralick, S.C. 1977, *IEEE Transactions on Communications*, vol. COM-25, pp. 1004–1009. [1]

12.4 FFT in Two or More Dimensions

Given a complex function $h(k_1, k_2)$ defined over the two-dimensional grid $0 \le k_1 \le N_1 - 1$, $0 \le k_2 \le N_2 - 1$, we can define its two-dimensional discrete Fourier transform as a complex function $H(n_1, n_2)$, defined over the same grid,

$$H(n_1, n_2) \equiv \sum_{k_2=0}^{N_2-1} \sum_{k_1=0}^{N_1-1} \exp(2\pi i k_2 n_2/N_2) \, \exp(2\pi i k_1 n_1/N_1) \, h(k_1, k_2)$$

(12.4.1)

By pulling the "subscripts 2" exponential outside of the sum over k_1, or by reversing the order of summation and pulling the "subscripts 1" outside of the sum over k_2, we can see instantly that the two-dimensional FFT can be computed by taking one-dimensional FFTs sequentially on each index of the original function. Symbolically,

$$H(n_1, n_2) = \text{FFT-on-index-1} \left(\text{FFT-on-index-2} \left[h(k_1, k_2) \right] \right)$$
$$= \text{FFT-on-index-2} \left(\text{FFT-on-index-1} \left[h(k_1, k_2) \right] \right)$$

(12.4.2)

For this to be practical, of course, both N_1 and N_2 should be some efficient length for an FFT, usually a power of 2. Programming a two-dimensional FFT, using (12.4.2) with a one-dimensional FFT routine, is a bit clumsier than it seems at first. Because the one-dimensional routine requires that its input be in consecutive order as a one-dimensional complex array, you find that you are endlessly copying things out of the multidimensional input array and then copying things back into it. This is not recommended technique. Rather, you should use a multidimensional FFT routine, such as the one we give below.

The generalization of (12.4.1) to more than two dimensions, say to L-dimensions, is evidently

$$
H(n_1, \ldots, n_L) \equiv \sum_{k_L=0}^{N_L-1} \cdots \sum_{k_1=0}^{N_1-1} \exp(2\pi i k_L n_L / N_L) \times \cdots
$$
$$
\times \exp(2\pi i k_1 n_1 / N_1) \, h(k_1, \ldots, k_L) \tag{12.4.3}
$$

where n_1 and k_1 range from 0 to $N_1 - 1, \ldots, n_L$ and k_L range from 0 to $N_L - 1$. How many calls to a one-dimensional FFT are in (12.4.3)? Quite a few! For each value of $k_1, k_2, \ldots, k_{L-1}$ you FFT to transform the L index. Then for each value of $k_1, k_2, \ldots, k_{L-2}$ and n_L you FFT to transform the $L-1$ index. And so on. It is best to rely on someone else having done the bookkeeping for once and for all.

The inverse transforms of (12.4.1) or (12.4.3) are just what you would expect them to be: Change the i's in the exponentials to $-i$'s, and put an overall factor of $1/(N_1 \times \cdots \times N_L)$ in front of the whole thing. Most other features of multidimensional FFTs are also analogous to features already discussed in the one-dimensional case:

- Frequencies are arranged in wrap-around order in the transform, but now for each separate dimension.
- The input data are also treated as if they were wrapped around. If they are discontinuous across this periodic identification (in any dimension) then the spectrum will have some excess power at high frequencies because of the discontinuity. The fix, if you care, is to remove multidimensional linear trends.
- If you are doing spatial filtering and are worried about wrap-around effects, then you need to zero-pad all around the border of the multidimensional array. However, be sure to notice how costly zero-padding is in multidimensional transforms. If you use too thick a zero-pad, you are going to waste a *lot* of storage, especially in 3 or more dimensions!
- Aliasing occurs as always if sufficient bandwidth limiting does not exist along one or more of the dimensions of the transform.

The routine `fourn` that we furnish herewith is a descendant of one written by N. M. Brenner. It requires as input (i) a scalar, telling the number of dimensions, e.g., 2; (ii) a vector, telling the length of the array in each dimension, e.g., (32,64). Note that these lengths *must all* be powers of 2, and are the numbers of *complex* values in each direction; (iii) the usual scalar equal to ± 1 indicating whether you want the transform or its inverse; and, finally (iv) the array of data.

A few words about the data array: `fourn` accesses it as a one-dimensional array of real numbers, that is, `data[1..(2N_1N_2...N_L)]`, of length equal to twice
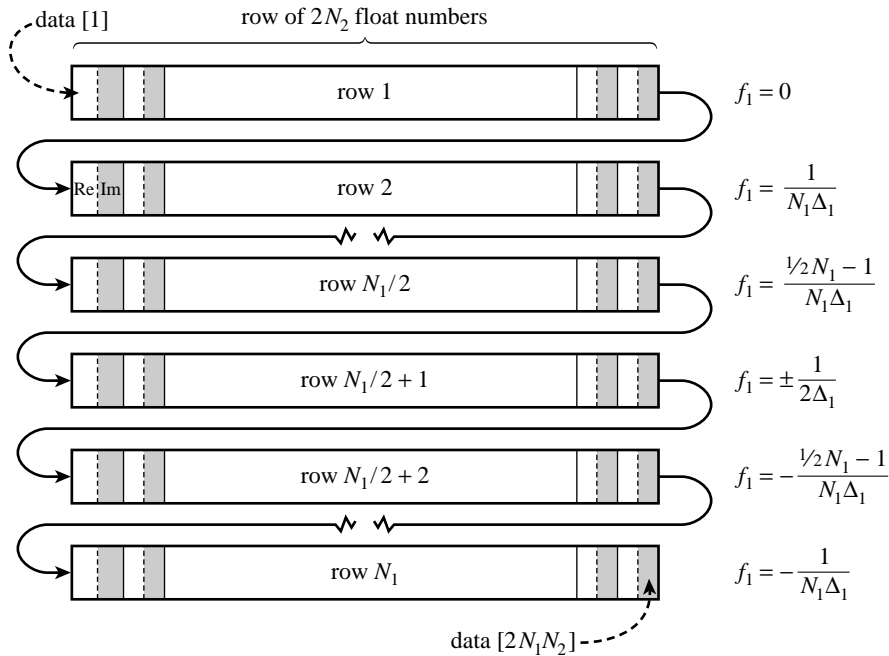
Figure 12.4.1. Storage arrangement of frequencies in the output $H(f_1, f_2)$ of a two-dimensional FFT. The input data is a two-dimensional $N_1 \times N_2$ array $h(t_1, t_2)$ (stored by rows of complex numbers). The output is also stored by complex rows. Each row corresponds to a particular value of f_1, as shown in the figure. Within each row, the arrangement of frequencies f_2 is exactly as shown in Figure 12.2.2. Δ_1 and Δ_2 are the sampling intervals in the 1 and 2 directions, respectively. The total number of (real) array elements is $2N_1 N_2$. The program `fourn` can also do more than two dimensions, and the storage arrangement generalizes in the obvious way.

the product of the lengths of the L dimensions. It assumes that the array represents an L-dimensional complex array, with individual components ordered as follows: (i) each complex value occupies two sequential locations, real part followed by imaginary; (ii) the first subscript changes least rapidly as one goes through the array; the last subscript changes most rapidly (that is, "store by rows," the C norm); (iii) subscripts range from 1 to their maximum values ($N_1, N_2, \ldots, N_L$, respectively), rather than from 0 to $N_1 - 1$, $N_2 - 1, \ldots$, $N_L - 1$. Almost all failures to get `fourn` to work result from improper understanding of the above ordering of the data array, so take care! (Figure 12.4.1 illustrates the format of the output array.)

```
#include <math.h>
#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr

void fourn(float data[], unsigned long nn[], int ndim, int isign)
```
Replaces `data` by its `ndim`-dimensional discrete Fourier transform, if `isign` is input as 1. `nn[1..ndim]` is an integer array containing the lengths of each dimension (number of complex values), which MUST all be powers of 2. `data` is a real array of length twice the product of these lengths, in which the data are stored as in a multidimensional complex array: real and imaginary parts of each element are in consecutive locations, and the rightmost index of the array increases most rapidly as one proceeds along `data`. For a two-dimensional array, this is equivalent to storing the array by rows. If `isign` is input as -1, `data` is replaced by its inverse transform times the product of the lengths of all dimensions.

```
{
    int idim;
    unsigned long i1,i2,i3,i2rev,i3rev,ip1,ip2,ip3,ifp1,ifp2;
    unsigned long ibit,k1,k2,n,nprev,nrem,ntot;
    float tempi,tempr;
    double theta,wi,wpi,wpr,wr,wtemp;                  Double precision for trigonometric recur-
                                                           rences.
    for (ntot=1,idim=1;idim<=ndim;idim++)              Compute total number of complex val-
        ntot *= nn[idim];                                 ues.
    nprev=1;
    for (idim=ndim;idim>=1;idim--) {                   Main loop over the dimensions.
        n=nn[idim];
        nrem=ntot/(n*nprev);
        ip1=nprev << 1;
        ip2=ip1*n;
        ip3=ip2*nrem;
        i2rev=1;
        for (i2=1;i2<=ip2;i2+=ip1) {                   This is the bit-reversal section of the
            if (i2 < i2rev) {                              routine.
                for (i1=i2;i1<=i2+ip1-2;i1+=2) {
                    for (i3=i1;i3<=ip3;i3+=ip2) {
                        i3rev=i2rev+i3-i2;
                        SWAP(data[i3],data[i3rev]);
                        SWAP(data[i3+1],data[i3rev+1]);
                    }
                }
            }
            ibit=ip2 >> 1;
            while (ibit >= ip1 && i2rev > ibit) {
                i2rev -= ibit;
                ibit >>= 1;
            }
            i2rev += ibit;
        }
        ifp1=ip1;                                      Here begins the Danielson-Lanczos sec-
        while (ifp1 < ip2) {                               tion of the routine.
            ifp2=ifp1 << 1;
            theta=isign*6.28318530717959/(ifp2/ip1);   Initialize for the trig. recur-
            wtemp=sin(0.5*theta);                          rence.
            wpr = -2.0*wtemp*wtemp;
            wpi=sin(theta);
            wr=1.0;
            wi=0.0;
            for (i3=1;i3<=ifp1;i3+=ip1) {
                for (i1=i3;i1<=i3+ip1-2;i1+=2) {
                    for (i2=i1;i2<=ip3;i2+=ifp2) {
                        k1=i2;                         Danielson-Lanczos formula:
                        k2=k1+ifp1;
                        tempr=(float)wr*data[k2]-(float)wi*data[k2+1];
                        tempi=(float)wr*data[k2+1]+(float)wi*data[k2];
                        data[k2]=data[k1]-tempr;
                        data[k2+1]=data[k1+1]-tempi;
                        data[k1] += tempr;
                        data[k1+1] += tempi;
                    }
                }
                wr=(wtemp=wr)*wpr-wi*wpi+wr;           Trigonometric recurrence.
                wi=wi*wpr+wtemp*wpi+wi;
            }
            ifp1=ifp2;
        }
        nprev *= n;
    }
}
```

CITED REFERENCES AND FURTHER READING:

Nussbaumer, H.J. 1982, *Fast Fourier Transform and Convolution Algorithms* (New York: Springer-Verlag).

12.5 Fourier Transforms of Real Data in Two and Three Dimensions

Two-dimensional FFTs are particularly important in the field of image processing. An image is usually represented as a two-dimensional array of pixel intensities, real (and usually positive) numbers. One commonly desires to filter high, or low, frequency spatial components from an image; or to convolve or deconvolve the image with some instrumental point spread function. Use of the FFT is by far the most efficient technique.

In three dimensions, a common use of the FFT is to solve Poisson's equation for a potential (e.g., electromagnetic or gravitational) on a three-dimensional lattice that represents the discretization of three-dimensional space. Here the source terms (mass or charge distribution) and the desired potentials are also real. In two and three dimensions, with large arrays, memory is often at a premium. It is therefore important to perform the FFTs, insofar as possible, on the data "in place." We want a routine with functionality similar to the multidimensional FFT routine `fourn` (§12.4), but which operates on real, not complex, input data. We give such a routine in this section. The development is analogous to that of §12.3 leading to the one-dimensional routine `realft`. (You might wish to review that material at this point, particularly equation 12.3.5.)

It is convenient to think of the independent variables $n_1, \ldots, n_L$ in equation (12.4.3) as representing an L-dimensional vector $\vec{n}$ in wave-number space, with values on the lattice of integers. The transform $H(n_1, \ldots, n_L)$ is then denoted $H(\vec{n})$.

It is easy to see that the transform $H(\vec{n})$ is periodic in each of its L dimensions. Specifically, if $\vec{P}_1, \vec{P}_2, \vec{P}_3, \ldots$ denote the vectors $(N_1, 0, 0, \ldots)$, $(0, N_2, 0, \ldots)$, $(0, 0, N_3, \ldots)$, and so forth, then

$$H(\vec{n} \pm \vec{P}_j) = H(\vec{n}) \qquad j = 1, \ldots, L \tag{12.5.1}$$

Equation (12.5.1) holds for any input data, real or complex. When the data is real, we have the additional symmetry

$$H(-\vec{n}) = H(\vec{n})^* \tag{12.5.2}$$

Equations (12.5.1) and (12.5.2) imply that the full transform can be trivially obtained from the subset of lattice values $\vec{n}$ that have

$$
\begin{aligned}
0 &\leq n_1 \leq N_1 - 1 \\
0 &\leq n_2 \leq N_2 - 1 \\
&\cdots \\
0 &\leq n_L \leq \frac{N_L}{2}
\end{aligned}
\tag{12.5.3}
$$

CITED REFERENCES AND FURTHER READING:

Nussbaumer, H.J. 1982, *Fast Fourier Transform and Convolution Algorithms* (New York: Springer-Verlag).

12.5 Fourier Transforms of Real Data in Two and Three Dimensions

Two-dimensional FFTs are particularly important in the field of image processing. An image is usually represented as a two-dimensional array of pixel intensities, real (and usually positive) numbers. One commonly desires to filter high, or low, frequency spatial components from an image; or to convolve or deconvolve the image with some instrumental point spread function. Use of the FFT is by far the most efficient technique.

In three dimensions, a common use of the FFT is to solve Poisson's equation for a potential (e.g., electromagnetic or gravitational) on a three-dimensional lattice that represents the discretization of three-dimensional space. Here the source terms (mass or charge distribution) and the desired potentials are also real. In two and three dimensions, with large arrays, memory is often at a premium. It is therefore important to perform the FFTs, insofar as possible, on the data "in place." We want a routine with functionality similar to the multidimensional FFT routine `fourn` (§12.4), but which operates on real, not complex, input data. We give such a routine in this section. The development is analogous to that of §12.3 leading to the one-dimensional routine `realft`. (You might wish to review that material at this point, particularly equation 12.3.5.)

It is convenient to think of the independent variables $n_1, \ldots, n_L$ in equation (12.4.3) as representing an L-dimensional vector $\vec{n}$ in wave-number space, with values on the lattice of integers. The transform $H(n_1, \ldots, n_L)$ is then denoted $H(\vec{n})$.

It is easy to see that the transform $H(\vec{n})$ is periodic in each of its L dimensions. Specifically, if $\vec{P}_1, \vec{P}_2, \vec{P}_3, \ldots$ denote the vectors $(N_1, 0, 0, \ldots)$, $(0, N_2, 0, \ldots)$, $(0, 0, N_3, \ldots)$, and so forth, then

$$H(\vec{n} \pm \vec{P}_j) = H(\vec{n}) \qquad j = 1, \ldots, L \qquad (12.5.1)$$

Equation (12.5.1) holds for any input data, real or complex. When the data is real, we have the additional symmetry

$$H(-\vec{n}) = H(\vec{n})^* \qquad (12.5.2)$$

Equations (12.5.1) and (12.5.2) imply that the full transform can be trivially obtained from the subset of lattice values $\vec{n}$ that have

$$0 \le n_1 \le N_1 - 1$$
$$0 \le n_2 \le N_2 - 1$$
$$\ldots \qquad (12.5.3)$$
$$0 \le n_L \le \frac{N_L}{2}$$

In fact, this set of values is overcomplete, because there are additional symmetry relations among the transform values that have $n_L = 0$ and $n_L = N_L/2$. However these symmetries are complicated and their use becomes extremely confusing. Therefore, we will compute our FFT on the lattice subset of equation (12.5.3), even though this requires a small amount of extra storage for the answer, i.e., the transform is not *quite* "in place." (Although an in-place transform is in fact possible, we have found it virtually impossible to explain to any user how to unscramble its output, i.e., where to find the real and imaginary components of the transform at some particular frequency!)

We will implement the multidimensional real Fourier transform for the three dimensional case $L = 3$, with the input data stored as a real, three-dimensional array data[1..nn1][1..nn2][1..nn3]. This scheme will allow two-dimensional data to be processed with effectively no loss of efficiency simply by choosing nn1 = 1. (Note that it must be the *first* dimension that is set to 1.) The output spectrum comes back packaged, logically at least, as a *complex*, three-dimensional array that we can call SPEC[1..nn1][1..nn2][1..nn3/2+1] (cf. equation 12.5.3). In the first two of its three dimensions, the respective frequency values f_1 or f_2 are stored in wrap-around order, that is with zero frequency in the first index value, the smallest positive frequency in the second index value, the smallest *negative* frequency in the *last* index value, and so on (cf. the discussion leading up to routines four1 and fourn). The third of the three dimensions returns only the positive half of the frequency spectrum. Figure 12.5.1 shows the logical storage scheme. The returned portion of the complex output spectrum is shown as the unshaded part of the lower figure.

The physical, as opposed to logical, packaging of the output spectrum is necessarily a bit different from the logical packaging, because C does not have a convenient, portable mechanism for equivalencing real and complex arrays. The subscript range SPEC[1..nn1][1..nn2][1..nn3/2] is returned in the input array data[1..nn1][1..nn2][1..nn3], with the correspondence
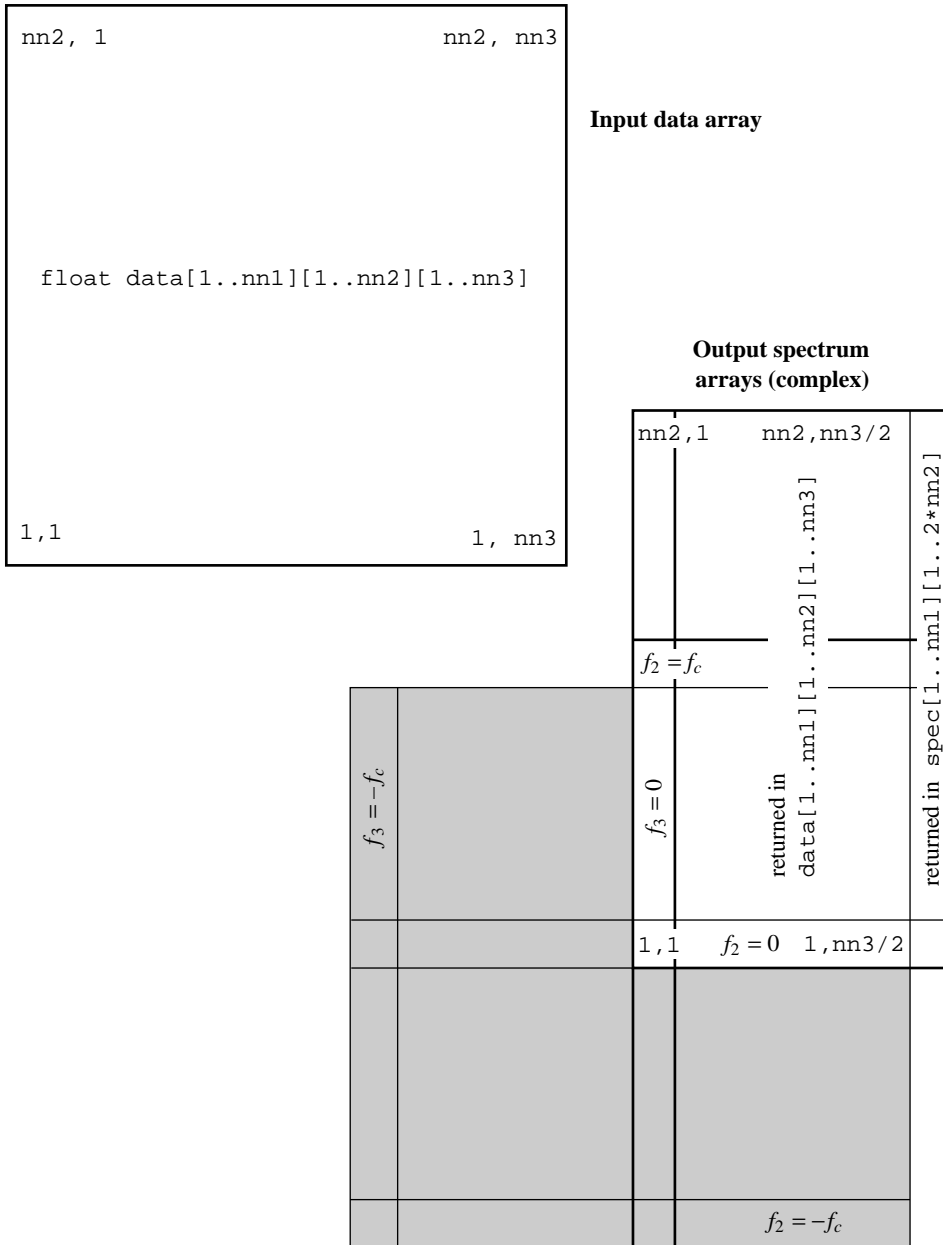
$$\text{Re}(\text{SPEC}[\text{i1}][\text{i2}][\text{i3}]) = \text{data}[\text{i1}][\text{i2}][2*\text{i3}-1]$$

$$\text{Im}(\text{SPEC}[\text{i1}][\text{i2}][\text{i3}]) = \text{data}[\text{i1}][\text{i2}][2*\text{i3}]$$

$$(12.5.4)$$

The remaining "plane" of values, SPEC[1..nn1][1..nn2][nn3/2+1], is returned in the two-dimensional float array speq[1..nn1][1..2*nn2],with the correspondence

$$\text{Re}(\text{SPEC}[\text{i1}][\text{i2}][\text{nn3/2+1}]) = \text{speq}[\text{i1}][2*\text{i2}-1]$$

$$\text{Im}(\text{SPEC}[\text{i1}][\text{i2}][\text{nn3/2+1}]) = \text{speq}[\text{i1}][2*\text{i2}]$$

$$(12.5.5)$$

Note that speq contains frequency components whose third component f_3 is at the Nyquist critical frequency $\pm f_c$. In some applications these values will in fact be ignored or set to zero, since they are intrinsically aliased between positive and negative frequencies.

With this much introduction, the implementing procedure, called rlft3, is something of an anticlimax. Look in the innermost loop in the procedure, and you will see equation (12.3.5) implemented on the *last* transform index. The case of i3=1 is coded separately, to account for the fact that speq is to be filled instead of

Input data array

Output spectrum arrays (complex)

Figure 12.5.1.   Input and output data arrangement for `rlft3`. All arrays shown are presumed to have a first (leftmost) dimension of range [1..nn1], coming out of the page. The input data array is a real, three-dimensional array `data[1..nn1][1..nn2][1..nn3]`. (For two-dimensional data, one sets `nn1 = 1`.) The output data can be viewed as a single complex array with dimensions [1..nn1][1..nn2][1..nn3/2+1] (cf. equation 12.5.3), corresponding to the frequency components f_1 and f_2 being stored in wrap-around order, but only positive f_3 values being stored (others being obtainable by symmetry). The output data is actually returned mostly in the input array `data`, but partly stored in the real array `speq[1..nn1][1..2*nn2]`. See text for details.

overwriting the input array of data. The three enclosing `for` loops (indices `i2`, `i3`, and `i1`, from inside to outside) could in fact be done in any order — their actions all commute. We chose the order shown because of the following considerations: (i) `i3` should not be the inner loop, because if it is, then the recurrence relations on `wr` and `wi` become burdensome. (ii) On virtual-memory machines, `i1` should be the outer loop, because (with C order of array storage) this results in the array `data`, which might be very large, being accessed in block sequential order.

Note that the work done in `rlft3` is quite (logarithmically) small, compared to the associated complex FFT, `fourn`. Since C does not have a convenient `complex` type, the operations are carried out explicitly below in terms of real and imaginary parts. The routine `rlft3` is based on an earlier routine by G.B. Rybicki.

```
#include <math.h>

void rlft3(float ***data, float **speq, unsigned long nn1, unsigned long nn2,
    unsigned long nn3, int isign)
```
Given a three-dimensional real array `data[1..nn1][1..nn2][1..nn3]` (where `nn1 = 1` for the case of a logically two-dimensional array), this routine returns (for `isign=1`) the complex fast Fourier transform as two complex arrays: On output, `data` contains the zero and positive frequency values of the third frequency component, while `speq[1..nn1][1..2*nn2]` contains the Nyquist critical frequency values of the third frequency component. First (and second) frequency components are stored for zero, positive, and negative frequencies, in standard wrap-around order. See text for description of how complex values are arranged. For `isign=-1`, the inverse transform (times `nn1*nn2*nn3/2` as a constant multiplicative factor) is performed, with output `data` (viewed as a real array) deriving from input `data` (viewed as complex) and `speq`. For inverse transforms on data not generated first by a forward transform, make sure the complex input data array satisfies property (12.5.2). The dimensions `nn1`, `nn2`, `nn3` must always be integer powers of 2.
```
{
    void fourn(float data[], unsigned long nn[], int ndim, int isign);
    void nrerror(char error_text[]);
    unsigned long i1,i2,i3,j1,j2,j3,nn[4],ii3;
    double theta,wi,wpi,wpr,wr,wtemp;
    float c1,c2,h1r,h1i,h2r,h2i;

    if (1+&data[nn1][nn2][nn3]-&data[1][1][1] != nn1*nn2*nn3)
        nrerror("rlft3: problem with dimensions or contiguity of data array\n");
    c1=0.5;
    c2 = -0.5*isign;
    theta=isign*(6.28318530717959/nn3);
    wtemp=sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi=sin(theta);
    nn[1]=nn1;
    nn[2]=nn2;
    nn[3]=nn3 >> 1;
    if (isign == 1) {                              Case of forward transform.
        fourn(&data[1][1][1]-1,nn,3,isign);        Here is where most all of the com-
        for (i1=1;i1<=nn1;i1++)                         pute time is spent.
            for (i2=1,j2=0;i2<=nn2;i2++) {         Extend data periodically into speq.
                speq[i1][++j2]=data[i1][i2][1];
                speq[i1][++j2]=data[i1][i2][2];
            }
    }
    for (i1=1;i1<=nn1;i1++) {
        j1=(i1 != 1 ? nn1-i1+2 : 1);
        Zero frequency is its own reflection, otherwise locate corresponding negative frequency
        in wrap-around order.
        wr=1.0;                                    Initialize trigonometric recurrence.
        wi=0.0;
        for (ii3=1,i3=1;i3<=(nn3>>2)+1;i3++,ii3+=2) {
```

Figure 12.5.2.   (a) A two-dimensional image with intensities either purely black or purely white. (b) The same image, after it has been low-pass filtered using `rlft3`. Regions with fine-scale features become gray.

```
for (i2=1;i2<=nn2;i2++) {
    if (i3 == 1) {                      Equation (12.3.5).
        j2=(i2 != 1 ? ((nn2-i2)<<1)+3 : 1);
        h1r=c1*(data[i1][i2][1]+speq[j1][j2]);
        h1i=c1*(data[i1][i2][2]-speq[j1][j2+1]);
        h2i=c2*(data[i1][i2][1]-speq[j1][j2]);
        h2r= -c2*(data[i1][i2][2]+speq[j1][j2+1]);
        data[i1][i2][1]=h1r+h2r;
        data[i1][i2][2]=h1i+h2i;
        speq[j1][j2]=h1r-h2r;
        speq[j1][j2+1]=h2i-h1i;
    } else {
        j2=(i2 != 1 ? nn2-i2+2 : 1);
        j3=nn3+3-(i3<<1);
        h1r=c1*(data[i1][i2][ii3]+data[j1][j2][j3]);
        h1i=c1*(data[i1][i2][ii3+1]-data[j1][j2][j3+1]);
        h2i=c2*(data[i1][i2][ii3]-data[j1][j2][j3]);
        h2r= -c2*(data[i1][i2][ii3+1]+data[j1][j2][j3+1]);
        data[i1][i2][ii3]=h1r+wr*h2r-wi*h2i;
        data[i1][i2][ii3+1]=h1i+wr*h2i+wi*h2r;
        data[j1][j2][j3]=h1r-wr*h2r+wi*h2i;
        data[j1][j2][j3+1]= -h1i+wr*h2i+wi*h2r;
    }
}
    wr=(wtemp=wr)*wpr-wi*wpi+wr;            Do the recurrence.
    wi=wi*wpr+wtemp*wpi+wi;
    }
}
if (isign == -1)                            Case of reverse transform.
    fourn(&data[1][1][1]-1,nn,3,isign);
}
```

We now give some fragments from notional calling programs, to clarify the use of `rlft3` for two- and three-dimensional data. Note again that the routine does not actually distinguish between two and three dimensions; two is treated like three, but with the first dimension having length 1. Since the first dimension is the outer loop, virtually no inefficiency is introduced.

The first program fragment FFTs a two-dimensional data array, allows for some

processing on it, e.g., filtering, and then takes the inverse transform. Figure 12.5.2 shows an example of the use of this kind of code: A sharp image becomes blurry when its high-frequency spatial components are suppressed by the factor (here) $\max\left(1 - 6f^2/f_c^2, 0\right)$. The second program example illustrates a three-dimensional transform, where the three dimensions have different lengths. The third program example is an example of convolution, as it might occur in a program to compute the potential generated by a three-dimensional distribution of sources.

```
#include <stdlib.h>
#include "nrutil.h"
#define N2 256
#define N3 256                          Note that the first component must be set to 1.

int main(void)        /* example1 */
This fragment shows how one might filter a 256 by 256 digital image.
{
    void rlft3(float ***data, float **speq, unsigned long nn1,
        unsigned long nn2, unsigned long nn3, int isign);
    float ***data, **speq;

    data=f3tensor(1,1,1,N2,1,N3);
    speq=matrix(1,1,1,2*N2);
/*    ...*/                  Here the image would be loaded into data.
    rlft3(data,speq,1,N2,N3,1);
/*    ...*/                  Here the arrays data and speq would be multiplied by a
    rlft3(data,speq,1,N2,N3,-1);         suitable filter function (of frequency).
/*    ...*/                  Here the filtered image would be unloaded from data.
    free_matrix(speq,1,1,1,2*N2);
    free_f3tensor(data,1,1,1,N2,1,N3);
    return 0;
}

#define N1 32
#define N2 64
#define N3 16

int main(void)        /* example2 */
This fragment shows how one might FFT a real three-dimensional array of size 32 by 64 by 16.
{
    void rlft3(float ***data, float **speq, unsigned long nn1,
        unsigned long nn2, unsigned long nn3, int isign);
    int j;
    float ***data,**speq;

    data=f3tensor(1,N1,1,N2,1,N3);
    speq=matrix(1,N1,1,2*N2);
/*    ...*/                  Here load data.
    rlft3(data,speq,N1,N2,N3,1);
/*    ...*/                  Here unload data and speq.
    free_matrix(speq,1,N1,1,2*N2);
    free_f3tensor(data,1,N1,1,N2,1,N3);
    return 0;
}

#define N 32

int main(void)        /* example3 */
This fragment shows how one might convolve two real, three-dimensional arrays of size 32 by
32 by 32, replacing the first array by the result.
{
    void rlft3(float ***data, float **speq, unsigned long nn1,
        unsigned long nn2, unsigned long nn3, int isign);
```

```
    int j;
    float fac,r,i,***data1,***data2,**speq1,**speq2,*sp1,*sp2;

    data1=f3tensor(1,N,1,N,1,N);
    data2=f3tensor(1,N,1,N,1,N);
    speq1=matrix(1,N,1,2*N);
    speq2=matrix(1,N,1,2*N);

/*   ...*/
    rlft3(data1,speq1,N,N,N,1);          FFT both input arrays.
    rlft3(data2,speq2,N,N,N,1);
    fac=2.0/(N*N*N);                     Factor needed to get normalized inverse.
    sp1 = &data1[1][1][1];
    sp2 = &data2[1][1][1];
    for (j=1;j<=N*N*N/2;j++) {           Note how this can be made a single for-loop in-
        r = sp1[0]*sp2[0] - sp1[1]*sp2[1];     stead of three nested ones by using
        i = sp1[0]*sp2[1] + sp1[1]*sp2[0];     the pointers sp1 and sp2.
        sp1[0] = fac*r;
        sp1[1] = fac*i;
        sp1 += 2;
        sp2 += 2;
    }
    sp1 = &speq1[1][1];
    sp2 = &speq2[1][1];
    for (j=1;j<=N*N;j++) {
        r = sp1[0]*sp2[0] - sp1[1]*sp2[1];
        i = sp1[0]*sp2[1] + sp1[1]*sp2[0];
        sp1[0] = fac*r;
        sp1[1] = fac*i;
        sp1 += 2;
        sp2 += 2;
    }
    rlft3(data1,speq1,N,N,N,-1);         Inverse FFT the product of the two FFTs.
/*   ...*/
    free_matrix(speq2,1,N,1,2*N);
    free_matrix(speq1,1,N,1,2*N);
    free_f3tensor(data2,1,N,1,N,1,N);
    free_f3tensor(data1,1,N,1,N,1,N);
    return 0;
}
```

To extend `rlft3` to four dimensions, you simply add an additional (outer) nested `for` loop in `i0`, analogous to the present `i1`. (Modifying the routine to do an *arbitrary* number of dimensions, as in `fourn`, is a good programming exercise for the reader.)

CITED REFERENCES AND FURTHER READING:

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall).

Swartztrauber, P. N. 1986, *Mathematics of Computation*, vol. 47, pp. 323–346.

12.6 External Storage or Memory-Local FFTs

Sometime in your life, you might have to compute the Fourier transform of a *really large* data set, larger than the size of your computer's physical memory. In such a case, the data will be stored on some external medium, such as magnetic or optical tape or disk. Needed is an algorithm that makes some manageable number of sequential passes through the external data, processing it on the fly and outputting intermediate results to other external media, which can be read on subsequent passes.

In fact, an algorithm of just this description was developed by Singleton [1] very soon after the discovery of the FFT. The algorithm requires four sequential storage devices, each capable of holding half of the input data. The first half of the input data is initially on one device, the second half on another.

Singleton's algorithm is based on the observation that it is possible to bit-reverse 2^M values by the following sequence of operations: On the first pass, values are read alternately from the two input devices, and written to a single output device (until it holds half the data), and then to the other output device. On the second pass, the output devices become input devices, and vice versa. Now, we copy *two* values from the first device, then *two* values from the second, writing them (as before) first to fill one output device, then to fill a second. Subsequent passes read 4, 8, etc., input values at a time. After completion of pass $M - 1$, the data are in bit-reverse order.

Singleton's next observation is that it is possible to alternate the passes of essentially this bit-reversal technique with passes that implement one stage of the Danielson-Lanczos combination formula (12.2.3). The scheme, roughly, is this: One starts as before with half the input data on one device, half on another. In the first pass, one complex value is read from each input device. Two combinations are formed, and one is written to each of two output devices. After this "computing" pass, the devices are rewound, and a "permutation" pass is performed, where groups of values are read from the first input device and alternately written to the first and second output devices; when the first input device is exhausted, the second is similarly processed. This sequence of computing and permutation passes is repeated $M - K - 1$ times, where 2^K is the size of internal buffer available to the program. The second phase of the computation consists of a final K computation passes. What distinguishes the second phase from the first is that, now, the permutations are local enough to do in place during the computation. There are thus no separate permutation passes in the second phase. In all, there are $2M - K - 2$ passes through the data.

Here is an implementation of Singleton's algorithm, based on [1]:

```
#include <stdio.h>
#include <math.h>
#include "nrutil.h"
#define KBF 128

void fourfs(FILE *file[5], unsigned long nn[], int ndim, int isign)
```
One- or multi-dimensional Fourier transform of a large data set stored on external media. On input, `ndim` is the number of dimensions, and `nn[1..ndim]` contains the lengths of each dimension (number of real and imaginary value pairs), which must be powers of two. `file[1..4]` contains the stream pointers to 4 temporary files, each large enough to hold half of the data. The four streams must be opened in the system's "binary" (as opposed to "text") mode. The input data must be in C normal order, with its first half stored in file `file[1]`, its second half in `file[2]`, in native floating point form. KBF real numbers are processed per buffered read or write. `isign` should be set to 1 for the Fourier transform, to -1 for its inverse. On output, values in the array `file` may have been permuted; the first half of the result is stored in `file[3]`, the second half in `file[4]`. N.B.: For `ndim > 1`, the output is stored by columns, i.e., *not* in C normal order; in other words, the output is the transpose of that which would have been produced by routine `fourn`.
```
{
    void fourew(FILE *file[5], int *na, int *nb, int *nc, int *nd);
    unsigned long j,j12,jk,k,kk,n=1,mm,kc=0,kd,ks,kr,nr,ns,nv;
    int cc,na,nb,nc,nd;
```