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,

$$H(f) = \int_{-\infty}^{\infty} h(t)e^{-2\pi ift} \, dt$$
$$h(t) = \int_{-\infty}^{\infty} H(f)e^{2\pi ift} \, df$$

(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 $\omega$, which is given in radians per sec. The relation between $\omega$ and $f$, $H(\omega)$ and $H(f)$ is

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

(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 $\omega$-convention, but we changed! There are fewer factors of $2\pi$ 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:

<table>
<thead>
<tr>
<th>If...</th>
<th>then...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h(t)$ is real</td>
<td>$H(-f) = [H(f)]^*$</td>
</tr>
<tr>
<td>$h(t)$ is imaginary</td>
<td>$H(-f) = -[H(f)]^*$</td>
</tr>
<tr>
<td>$h(t)$ is even</td>
<td>$H(-f) = H(f)$ [i.e., $H(f)$ is even]</td>
</tr>
<tr>
<td>$h(t)$ is odd</td>
<td>$H(-f) = -H(f)$ [i.e., $H(f)$ is odd]</td>
</tr>
<tr>
<td>$h(t)$ is real and even</td>
<td>$H(f)$ is real and even</td>
</tr>
<tr>
<td>$h(t)$ is real and odd</td>
<td>$H(f)$ is imaginary and odd</td>
</tr>
<tr>
<td>$h(t)$ is imaginary and even</td>
<td>$H(f)$ is imaginary and even</td>
</tr>
<tr>
<td>$h(t)$ is imaginary and odd</td>
<td>$H(f)$ is real and odd</td>
</tr>
</tbody>
</table>

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 "$\iff$" symbol to indicate transform pairs.) If

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

is such a pair, then other transform pairs are

$$h(at) \iff \frac{1}{|a|}H\left(\frac{f}{a}\right)$$

"time scaling" (12.0.4)

$$\frac{1}{|b|}h\left(\frac{t}{b}\right) \iff H(bf)$$

"frequency scaling" (12.0.5)

$$h(t-t_0) \iff H(f) e^{2\pi ift_0}$$

"time shifting" (12.0.6)

$$h(t) e^{-2\pi ift_0} \iff H(f-f_0)$$

"frequency shifting" (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

\[
g * h = \int_{-\infty}^{\infty} g(\tau) h(t-\tau) \, d\tauanumber{12.0.8}
\]

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

\[
g * h \iff G(f)H(f) \quad \text{"Convolution Theorem"}anumber{12.0.9}
\]

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

\[
\text{Corr}(g, h) = \int_{-\infty}^{\infty} g(\tau) h(\tau) \, d\tau \quad \text{(12.0.10)}
\]

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

\[
\text{Corr}(g, h) \iff G(f)H^*(f) \quad \text{"Correlation Theorem"}anumber{12.0.11}
\]

[More generally, the second member of the pair is \( G(f)H(\alpha f) \), 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

\[
\text{Corr}(g, g) \iff |G(f)|^2 \quad \text{"Wiener-Khinchin Theorem"}anumber{12.0.12}
\]

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:

\[
\text{Total Power} = \int_{-\infty}^{\infty} |h(t)|^2 \, dt = \int_{-\infty}^{\infty} |H(f)|^2 \, df \quad \text{(12.0.13)}
\]

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

\[
P_h(f) = |H(f)|^2 + |H(-f)|^2 \quad 0 \leq f < \infty \quad \text{(12.0.14)}
\]

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 \( P_h(f) = 2|H(f)|^2 \).

Be warned that one occasionally sees PSDs defined without this factor two. These, strictly speaking, are called one-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 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 \( h(t_k) \) for a discrete set of \( t_k \)'s. The profound implications of this seemingly unimportant fact are the subject of the next section.

CITED REFERENCES AND FURTHER READING:

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 \( \Delta \) denote the time interval between consecutive samples, so that the sequence of sampled values is

\[
h_n = h(n\Delta) \quad n = \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots
\]

(12.1.1)

The reciprocal of the time interval \( \Delta \) is called the sampling rate; if \( \Delta \) 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 \( \Delta \), there is also a special frequency \( f_c \), called the Nyquist critical frequency, given by

\[
f_c \equiv \frac{1}{2\Delta}
\]

(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 \( \Delta \). 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 \( \Delta \), 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

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

(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 \( \Delta^{-1} \) 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 \( \exp(2\pi_if_1t) \) and \( \exp(2\pi if_2t) \) give the same samples at an interval \( \Delta \) 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 rich 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

\[
h_k = h(t_k), \quad t_k = k\Delta, \quad k = 0, 1, 2, \ldots, N-1
\]

(12.1.4)
The remaining step is to approximate the integral in (12.0.1) by a discrete sum:

\[ H(f_n) = \int_{-\infty}^{\infty} h(t) e^{2\pi i f_n t} dt \approx \sum_{k=0}^{N-1} h_k e^{2\pi i f_n \Delta} = \Delta \sum_{k=0}^{N-1} h_k e^{2\pi ikn/N} \quad (12.1.6) \]

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 \(H_n\):

\[ H_n = \sum_{k=0}^{N-1} h_k e^{2\pi i kn/N} \quad (12.1.7) \]

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 \(\Delta\). 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 \(\Delta\) can be rewritten as

\[ H(f_n) \approx \Delta H_n \quad (12.1.8) \]

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, \(H_{n} = H_{N-n} \quad n = 1, 2, \ldots\). 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 \(1 \leq n \leq N/2 - 1\), while negative frequencies \(-f_c < f < 0\) correspond to \(N/2 + 1 \leq n \leq N - 1\). 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:

\[ h_k = \frac{1}{N} \sum_{n=0}^{N-1} H_n e^{-2\pi i kn/N} \quad (12.1.9) \]

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
\[ \sum_{k=0}^{N-1} |h_k|^2 = \frac{1}{N} \sum_{n=0}^{N-1} |H_n|^2 \]  
(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.

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
\[ W = e^{2\pi i/N} \]  
(12.2.1)

Then (12.1.7) can be written as
\[ H_n = \sum_{k=0}^{N-1} W^{nk} h_k \]  
(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:
\[
F_k = \sum_{j=0}^{N/2-1} e^{2\pi ijk/N} f_j = \sum_{j=0}^{N/2-1} e^{2\pi i(k/2)j/N} f_{2j} + \sum_{j=0}^{N/2-1} e^{2\pi i(k/2+1)j/N} f_{2j+1} 
= \sum_{j=0}^{N/2-1} e^{2\pi i(k/2)j/(N/2)} f_{2j} + e^{\pi ik} \sum_{j=0}^{N/2-1} e^{2\pi i(k/2+1)j/(N/2)} f_{2j+1} 
= F_k^e + W^k F_k^o
\]  
(12.2.3)

In the last line, \( W \) is the same complex constant as in (12.2.1). \( F_k^e \) denotes the \( k \)th 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 \( F_k^e \) and \( F_k^o \) 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 \( f_n \),
\[ F_k^{eeoo...oo} = f_n \]  
(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...
Lemma makes FFT's practical: Suppose we take the original vector of data \( f_j \) and rearrange it into bit-reversed order (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 four-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 \( \text{data}[1..2*nn] \), and \( \text{isign} \), which should be set to either \( \pm 1 \) and is the sign of \( i \) in the exponential of equation (12.1.7). When \( \text{isign} \) 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 \( \text{data}[1..2*nn] \) is \( 2 \times \text{nn} \), with each complex value occupying two consecutive locations. In other words, \( \text{data}[1] \) is the real part of \( f_0 \), \( \text{data}[2] \) is the imaginary part of \( f_0 \), and so on up to \( \text{data}[2*\text{nn}-1] \), which is the real part of \( f_{\text{nn}-1} \), and \( \text{data}[2*\text{nn}] \), which is the imaginary part of \( f_{\text{nn}-1} \). 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 \( \text{data}[1] \) and \( \text{data}[2] \); the smallest nonzero positive frequency has real and imaginary parts in \( \text{data}[3] \) and \( \text{data}[4] \); the smallest (in magnitude) nonzero negative frequency has real and imaginary parts in \( \text{data}[2*\text{nn}-1] \) and \( \text{data}[2*\text{nn}] \). Positive frequencies increasing in magnitude are stored in the real-imaginary pairs \( \text{data}[5] \), \( \text{data}[6] \) up to \( \text{data}[2*\text{nn}-1] \), \( \text{data}[\text{nn}] \). Negative frequencies of increasing magnitude are stored in \( \text{data}[2*\text{nn}-1] \) down to \( \text{data}[\text{nn}] \). Finally, the pair \( \text{data}[\text{nn}+1] \), \( \text{data}[\text{nn}+2] \) 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 \texttt{four1} without modification even if your input data array is zero-offset, that is has the range \( \text{data}[0..2*\text{nn}-1] \). In this case, simply decrement the pointer to \( \text{data} \) by one when \texttt{four1} is invoked, e.g., \texttt{four1(data-1,1024,1)}; The real part of \( f_0 \) will now be returned in \( \text{data}[0] \), the imaginary part in \( \text{data}[1] \), and so on. See §12.2.

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

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 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!).
{ 
  unsigned long n,mm,m,j,i,step,i;
  double wtemp,wr,wi,xi,theta;
  float temp,twmpi;
  n=nn << 1;
  j=1;
  for (i=1;i<nn;i<<=2) {
    step = j >> 1;
    SWAP(data[j],data[i]);
    SWAP(data[j+1],data[i+1]);
  }
  n>>=1;
  while ((m = 2 & j) > 0) {
    j -= m;
    n >>= 1;
  }
  j <<= m;
  if (isign > 0)
  } else
  } else
  if (isign == -1) {
    mm = 2*nn - 1;
    wtemp = sin(0.5*theta);
    theta = isign*(0.2881885767593992); // Outer loop executed log2 nn times.
    while (m > maxm) {
      isign = maxm < 1;
      temp = sin(0.5*theta);
      wtemp = -2.0*wtemp*wtemp;
      theta = isign*(0.2881885767593992); // Initialize the trigonometric recurrence.
      temp = 2.0*temp*temp - 1.0;
      t = 2.0*temp*
      } else
    } else
  }
}
```
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 blocks 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 \( \pm 1 \) 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

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.)