Numerical Recipes in C

The Art of Scientific Computing

William H. Press
Harvard-Smithsonian Center for Astrophysics

Brian P. Flannery
EXXON Research *and Engineering Company*

Saul A. Teukolsky
Department of Physics, Cornell University

William T. Vetterling
Polaroid Corporation

The right of the University of Cambridge to print and sell all manner of books was granted by Henry VIII in 1534. The University has printed and published continuously since 1584.

CAMBRIDGE UNIVERSITY PRESS
Cambridge
New York Port Chester
Melbourne Sydney

Chapter 12. Fourier Transform Spectral Methods

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. In this day and age 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.

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,

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

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. 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* w, which is given in *radians* per sec. The relation between w and f, H(w) and H(f) is

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

and equation (12.0.1) looks like this

$$\begin{aligned} H(w) &= \int_{-\infty}^{\infty} h(t)e^{i\omega t}dt \\ h(t) &= \frac{1}{2\pi}\int_{-\infty}^{\infty} H(\omega)e^{-i\omega t}d\omega \end{aligned} \qquad (12.0.3)$$

We were raised on the w-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 $P_h(f) = 2\,|H(f)|^2$. 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 which 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 §12.8–§12.9. 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

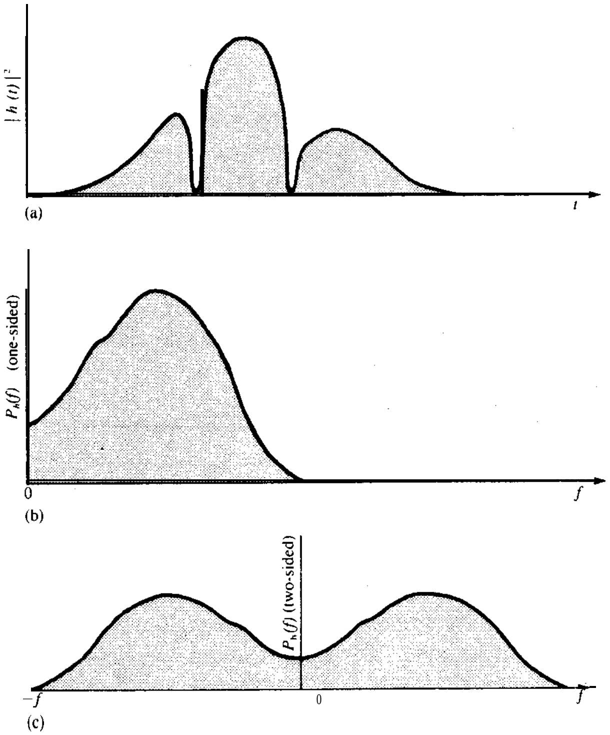

Figure 12.0.1 Normalizations of one- and two-sided power spectra. The area under the square of the function, (a), equals the area under its one-sided power spectrum at positive frequencies, (b), and also equals the area under its two-sided power spectrum at positive and negative frequencies, (c).

set of t_i's. The profound implications of this seemingly unimportant fact are the subject of the next section.

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

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

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

$$f_c \equiv \frac{1}{2\Delta} \tag{12.1.2}$$

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 ***band-width limited*** to frequencies smaller in magnitude than f_c, i.e., if $H(f) = 0$ for all $|f| > f_c$, then the function ***h(t)*** is completely ***determined*** by its samples ***h,.*** 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 band-width limited function is, in some sense, infinitely smaller than that of a general continuous function. Fairly often, one is dealing with a signal which is known on physical grounds to be band-width limited (or at least approximately band-width~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 band-width limited to less than the Nyquist critical frequency. In that case, it turns out that all of the power spectral density which 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 A if and only if f_1 and f_2 differ by a multiple of 1/A, 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 band-width 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 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

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

so that the sampling interval is A. 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

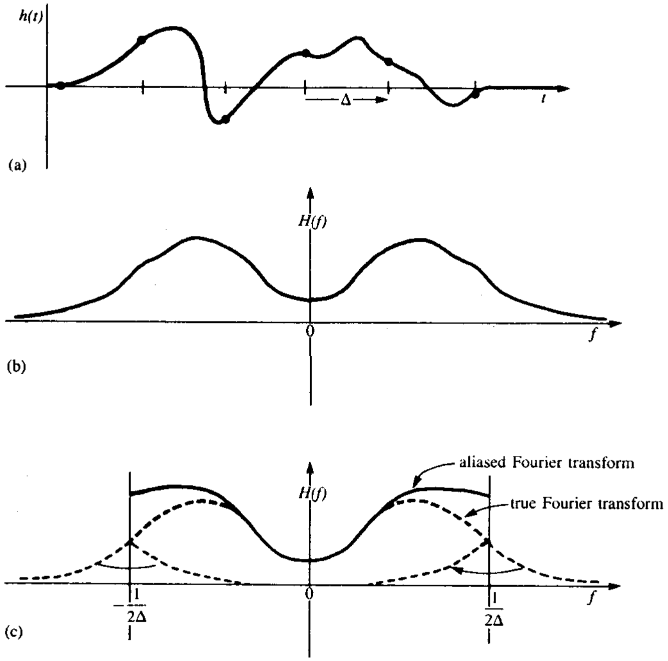

Figure 12.1.1. The continuous function shown in (a) is nonzero only for a finite interval of time T. It follows that its Fourier transform, shown schematically in (b), is not bandwidth limited but has finite amplitude for all frequencies. If the original function is sampled with a sampling interval A, 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.***

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

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:

$$H(f_n) = \int_{-\infty}^{\infty} h(t)e^{2\pi i f_n t}dt \approx \sum_{k=0}^{N-1} h_k\, e^{2\pi i f_n t_k}\Delta = \Delta \sum_{k=0}^{N-1} h_k\, e^{2\pi i k n/N} \qquad (12.1.6)$$

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,

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

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 A. 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 A 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 $1 \leq n \leq N/2 - 1$, 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:

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

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

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

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 §12.4 and §12.5, respectively.

REFERENCES AND FURTHER READING:

Brigham, E. Oran. *1974, The Fast Fourier Transform* (Englewood Cliffs, N.J.: 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-1060s, the standard answer was this: Define *W* as the complex number

$$W \equiv e^{2\pi i/N} \qquad (12.2.1)$$

Then (12.1.7) can be written as

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

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 x 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, 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, who in turn had been prodded by R.L. Garwin of IBM Yorktown Heights Research Center. Retrospectively, we now know that a few clever individuals had independently discovered, and in some cases implemented, fast Fourier transforms as many as 20 years previously (see Brigham for references).

One of the earliest "discoveries" of the FFT, that of Danielson and Lanczos in 1942, still 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:

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

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^{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 e's and o's (numbering log, N in all), there is a one-point transform that is just one of the input numbers f_n

$$F_k^{eoeeoeo\cdots oee} = f_n \quad \text{for some n} \tag{12.2.4}$$

(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 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, N combinations, so the whole algorithm is of order N log, N (assuming, as is the case, that the process of sorting into bit-reversed order is no greater in order than N log, 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 which is executed log, N times and calculates, in turn, transforms of length 2, 4, 8, . . . , 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

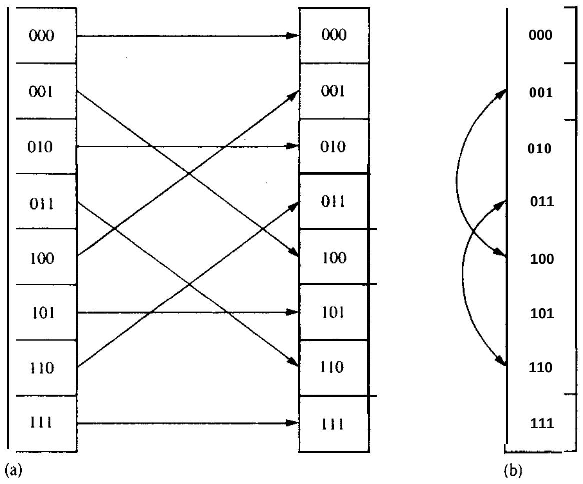

Figure 12.2.1. Reordering an array (here of length 8) by bit reversal, (a) between two arrays, versus (b) in place. Bit reversal reordering is a necessary part of the Fast Fourier Transform (FFT) algorithm.

cosines to the outer loop, where they are made only log, N times. Computation of the sines and cosines of multiple angles is through simple recurrence relations in the inner loops.

The FFT routine given below is based on one originally written by N. Brenner of Lincoln Laboratories. 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 which contains the

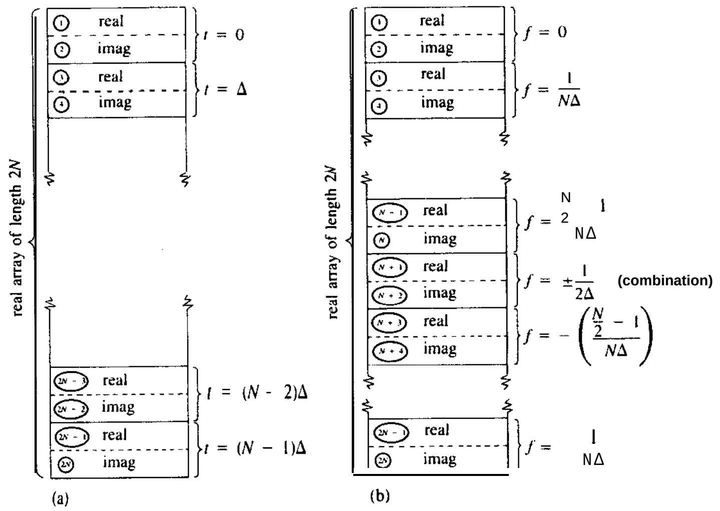

Figure 12.2.2. Input and output arrays for FFT. (a) The input array contains N (a power of 2) complex time samples in a real array of length $2N$, with real and imaginary parts alternating. (b) The output array contains the complex Fourier spectrum at N values of frequency. Real and imaginary parts again 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.

most positive and the most negative frequency. You should try to develop a familiarity with this storage arrangement of complex spectra, also shown in the 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-l, 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.

```
#include <math.h>

#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr

void four1(data,nn,isign)
float data[];
int nn,isign;
Replaces data by its discrete Fourier transform, if isign is input as 1; or replaces data by
nn times its inverse discrete Fourier transform, if isign is input as -1. data is a complex
array of length nn, input as a real array data[1. .2*nn]. nn MUST be an integer power of
2 (this is not checked for!).
{
    int n,mmax,m,j,istep,i;
    double wtemp,wr,wpr,wpi,wi,theta;          Double precision for the trigonometric recurrences.
    float tempr,tempi;
    n=nn << 1;
    j=1;
    for (i=1;i<n;i+=2) {                          This IS the bit-reversal section of the routine
        if (j > i) {
            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;
    }
    mmax=2;                                   Here begins the Danielson-Lanczos Section of the routine.
    while (n > mmax) {                        Outer loop executed log2 nn times.
        istep=2*mmax;
        theta=6.28318530717959/(isign*mmax);      Initialize for the trigonometric recurrence.
        wtemp=sin(0.5*theta);
        wpr = -2.0*wtemp*wtemp;
        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 formula:
                tempr=wr*data[j]-wi*data[j+1];
                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;
            }                                     Trigonometric recurrence.
            wr=(wtemp=wr)*wpr-wi*wpi+wr;
            wi=wi*wpr+wtemp*wpi+wi;
        }
        mmax=istep;
    }
}
```

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, 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 which first go through a set of log, 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 (§12.4), 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, 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 N^{th} 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.

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. Oran. 1974, *The Fast Fourier Transform* (Englewood Cliffs, N.J.: Prentice-Hall).

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

Beauchamp, K.G. 1975, *Walsh Functions and Their Applications* (New York: Academic Press) [a non-Fourier transform of recent interest].

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, $j = 0. . .N - 1$. To use fourl, 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 fourl. 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(data1,data2,fft1,fft2,n)
float data1[],data2[],fft1[],fft2[];
int n;
Given two real input arrays data1 Cl. .n] and data2 [1. n] , this routine calls four1 and returns
two complex output arrays, fft1 and fft2, each of complex length n (i.e. real dimensions
[1..2n]), which contain the discrete Fourier transforms of the respective datas. n MUST
be an integer power of 2.
{
    int nn3,nn2,jj,j;
    float rep,rem,aip,aim;
    void four1();

    nn3=1+(nn2=2+n+n);
    for (j=1,jj=2;j<=n;j++,jj+=2) {
        fft1[jj-1]=data1[j];                 Pack the two real arrays into one complex ar-
        fft1[jj]=data2[j];                       ray.
    }
    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 set of two transforms each of which has half of the information we need. Fortunately, this is schizophrenia of a treatable form. 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

$$F_n^e = \sum_{k=0}^{N/2-1} f_{2k}\, e^{2\pi ikn/(N/2)}$$
$$F_n^o = \sum_{k=0}^{N/2-1} f_{2k+1}\, e^{2\pi ikn/(N/2)} \qquad (12.3.3)$$

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? We recommend a quick glance back at equation (12.2.3):

$$F_n = F_n^e + e^{2\pi in/N} F_n^o \qquad n = 0 \ldots N-1 \qquad (12.3.4)$$

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

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

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

$$F_n^e = \frac{1}{2}(F_n + F_{N/2-n}^*)$$
$$F_n^o = \frac{1}{2}e^{-2\pi in/N}(F_n - F_{N/2-n}^*) \qquad n = 0 \ldots N/2-1 \qquad (12.3.6)$$

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(data,n,isign)
float data[];
int n,isign;
```

Calculates the Fourier Transform of a set of `2n` real-valued data points. Replaces this data (which is stored in array `data[1. .2n]`) 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 l/n.)

```
{
    int i,i1,i2,i3,i4,n2p3;
    float c1=0.5,c2,h1r,h1i,h2r,h2i;
    double wr,wi,wpr,wpi,wtemp,theta;     Double precision for the trigonometric recur-
    void four1();                             rences.

    theta=3.141592653589793/(double) n;   Initialize the recurrence.
    if (isign == 1) {
        c2 = -0.6;
        four1(data,n,1);                  The forward transform is here.
    } else {
        c2=0.5;                           Otherwise set up for an inverse transform.
        theta = -theta;
    }
    wtemp=sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi=sin(theta);
    wr=1.0+wpr;
    wi=wpi;
    n2p3=2*n+3;
    for (i=2;i<=n/2;i++) {                Case i=1 done separately below.
        i4=1+(i3=n2p3-(i2=1+(i1=i+i-1)));
        h1r=c1*(data[i1]+data[i3]);       The two separate transforms are separated
        h1i=c1*(data[i2]-data[i4]);           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 the true
        data[i2]=h1i+wr*h2i+wi*h2r;           transform of the original real data.
        data[i3]=h1r-wr*h2r+wi*h2i;
        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 together to get
        data[2] = h1r-data[2];                      them all within The original array.
    } else {
        data[1]=c1*((h1r=data[1])+data[2]);
        data[2]=c1*(h1r-data[2]);
        four1(data,n,-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 Chapter 17). 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 these instances, two more transforms arise naturally, the sine transform and the cosine transform, given by

$$
\begin{aligned}
F_k &= \sum_{j=1}^{N-1} f_j \sin(\pi jk/N) \qquad \text{sine transform}\\
F_k &= \sum_{j=0}^{N-1} f_j \cos(\pi jk/N) \qquad \text{cosine transform}
\end{aligned}
\tag{12.3.7}
$$

where f_j, $j = 0 \dots N-1$ is the dataarray.

At first blush these appear to be simply the imaginary and real parts respectively of the discrete Fourier transform. However, the argument of the sine and cosine differ by a factor of two from the value that would make that so. The sine transform uses sines only as a complete set of functions in the interval from 0 to 2π, and the cosine transform uses cosines only. By contrast, the normal FFT uses both sines and cosines. (See Figure 12.3.1.)

However, the sine and cosine transforms can be "force-fit" into a form which allows their calculation via the FFT. The idea is to extend the given function rightward past its last tabulated value. In the case of the sine transform, we extend the data to twice their length in such a way as to make them an *odd* function about $j = N$, with $f_N = 0$,

$$f_{2N-j} \equiv -f_j \qquad j = 0, \dots, N-1 \tag{12.3.8}$$

When a FFT is performed on this extended function, it reduces to the sine transform by symmetry:

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

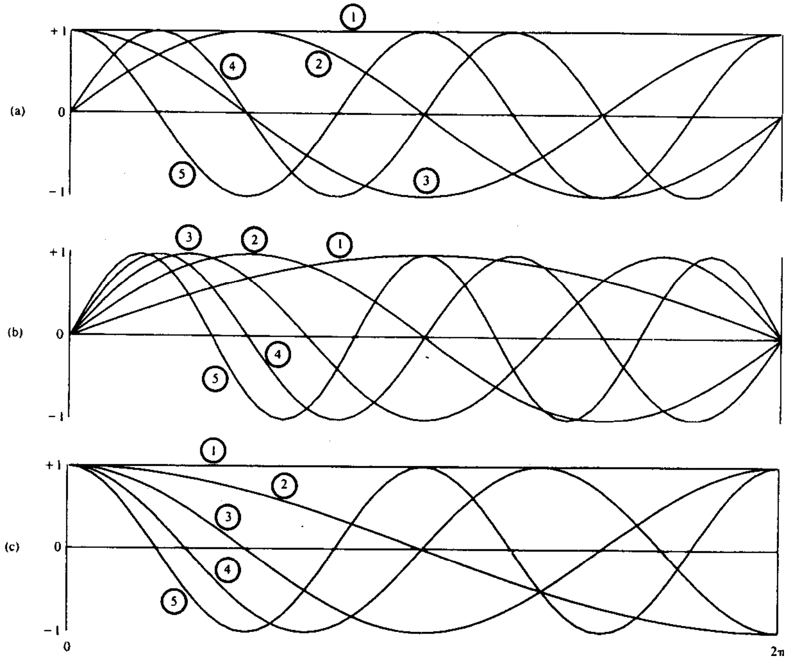

Figure 12.3.1. Basis functions used by the Fourier transform (a), sine transform (b), and cosine transform (c), are plotted. The first five basis functions are shown in each case. (For the Fourier transform, the real and imaginary parts of the basis functions are both shown.) While some basis functions 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.

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

$$
\begin{aligned}
\sum_{j=N}^{2N-1} f_j e^{2\pi ijk/(2N)} &= \sum_{j'=1}^{N} f_{2N-j'} e^{2\pi i(2N-j')k/(2N)}\\
&= -\sum_{j'=0}^{N-1} f_{j'} e^{-2\pi ij'k/(2N)}
\end{aligned}
\tag{12.3.10}
$$

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. The same procedure applies to the cosine transform with the exception that the data are extended as an even function.

In both cases, however, 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 the sine 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 we are inspired to eliminate the inefficiency.

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, $j = 1, \ldots, N$ the auxiliary array is

$$\begin{aligned} y_0 &= 0 \\ y_j &= \sin(j\pi/N)(f_j + f_{N-j}) + \frac{1}{2}(f_j - f_{N-j}) \qquad (12.3.12) \\ j &= 1, \ldots, N-1 \end{aligned}$$

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 part8 I_k given by

$$\begin{aligned} R_k &= \sum_{j=0}^{N-1} y_j \cos(2\pi jk/N) \\ &= \sum_{j=1}^{N-1} (f_j + f_{N-j}) \sin(j\pi/N) \cos(2\pi jk/N) \\ &= \sum_{j=0}^{N-1} 2f_j \sin(j\pi/N) \cos(2\pi jk/N) \end{aligned}$$

$$\begin{aligned} &= \sum_{j=0}^{N-1} f_j \left\{ \sin(\frac{2k+1}{N} j\pi) - \sin(\frac{2k-1}{N} j\pi) \right\} \\ &= F_{2k+1} - F_{2k-1} \qquad (12.3.13) \end{aligned}$$

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

Therefore F_k can be determined as follows,

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

The even terms of F_k are thus determined very directly. The odd terms require a recursion, the starting point of which is

$$F_1 = \sum_{j=0}^{N-1} f_j \sin(j\pi/N) \qquad (12.3.16)$$

The implementing program is

```
#include <math.h>

void sinft(y,n)
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.
{
    int j,m=n/2,n2=n+2;
    float sum,y1,y2;
    double theta,wi=0.0,wr=1.0,wpi,wpr,wtemp;      Double precision in the trigonometric
    void realf t ();                                    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<=m+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,m,1);                           Transform the auxiliary array.
    y[1]*=0.5;
    sum=y[2]=0.0;                            Initialize the sum used for odd terms below.
    for (j=1;j<=n-1;j+=2) {
        sum += y[j];
        y[j]=y[j+1];                         Even terms in the transform are determined directly.
        y[j+1]=sum;                          Odd terms are 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 cosine transform is slightly more difficult, but the idea is the same. Now, the auxiliary function is

$$y_0 = f_0 \qquad y_j = \frac{1}{2}(f_j + f_{N-j}) - \sin(j\pi/N)(f_j - f_{N-j}) \tag{12.3.17}$$

and the same analysis leads to

$$F_{2k} = R_k \qquad F_{2k+1} = F_{2k-1} + I_k \qquad k = 0, \ldots, (N/2 - 1) \tag{12.3.18}$$

The starting value for the recursion in this case is

$$F_1 = \sum_{j=0}^{N-1} f_j \cos(j\pi/N) \tag{12.3.19}$$

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_i.

An additional complication is that the cosine transform is not its own inverse. To derive the inverse of an array F_k, we first compute an array $\widetilde{f}_l$ as if the transform were its own inverse,

$$\widetilde{f}_l = \sum_{k=0}^{N-1} F_k \cos(\pi k l/N) \tag{12.3.20}$$

One easily verifies the relations between $\widetilde{f}_l$ and the desired inverse f_l,

$$\begin{aligned} \widetilde{f}_0 &= N f_0 + \sum_{j \text{ odd}} f_j \\ \widetilde{f}_l &= \frac{N}{2} f_l + \sum_{j \text{ even}} f_j \qquad \text{for 1 odd} \\ \widetilde{f}_l &= \frac{N}{2} f_l + \sum_{j \text{ odd}} f_j \qquad \text{for 1 even, } l \neq 0 \end{aligned} \tag{12.3.21}$$

The unknown sums on the right of these equations are determined as follows

$$\begin{aligned} C_1 &\equiv \sum_{1 \text{ even}} \widetilde{f}_l = \frac{N}{2}\left(f_0 + \sum_{j=0}^{N-1} f_j\right) \\ C_2 &\equiv \sum_{l \text{ odd}} \widetilde{f}_l = \frac{N}{2} \sum_{j=0}^{N-1} f_j \end{aligned} \tag{12.3.22}$$

It follows that

$$\frac{N}{2} f_0 = C_1 - C_2 \tag{12.3.23}$$

$$\sum_{j \text{ odd}} f_j = \widetilde{f}_0 - 2(C_1 - C_2) \qquad \sum_{j \text{ even}} f_j = \frac{2}{N} C_2 - \sum_{j \text{ odd}} f_j \tag{12.3.24}$$

Knowing these sums, the desired f_j are now easily recovered from equation (12.3.21). This is implemented in the following routine.

```
#include<math.h>

void cosft(y,n,isign)
float y[];
int n,isign;
Calculates the 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.
{
    int j,m,n2;
    float enf0,even,odd,sum,sume,sumo,y1,y2;
    double theta,wi=0.0,wr=1.0,wpi,wpr,wtemp;      Double precision for the trigonomet-
    void realft();                                     ric recurrences.

    theta=3.14159265358979/(double) n;             Initialize the recurrence.
```

```
	wtemp=sin(0.5*theta);
	wpr = -2.0*wtemp*wtemp;
	wpi=sin(theta);
	sum=y[1];
	m=n >> 1;
	n2=n+2;
	for (j-2; j<=m; j++) {             j=m+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]);              Calculates 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 unfolding the transform.
	}
	realft(y,m,1);                      Calculate the transform of the auxiliary function.
	y[2]=sum;                           sum is the value in equation (12.3.19).
	for (j=4;j<=n;j+=2) {
		sum += y[j];                    Equation (12.3.18).
		y[j]=sum;
	}
	if (isign == -1) {                  This code applies only to the inverse transform.
		even=y[1];
		odd=y[2];
		for (j=3;j<=n-1;j+=2) {
			even += y[j];               Sum up the even and odd transform values as in equation (12.3.22).
			odd += y[j+1];
		}
		enf0=2.0*(even-odd);
		sumo=y[1]-enf0;                 Next. implement equation (12.3.24).
		sume=(2.0*odd/n)-sumo;
		y[1]=0.5*enf0;
		y[2] -= sume;
		for (j=3;j<=n-1;j+=2) {
			y[j] -= sumo;               Finally, equation (12.3.21) gives us the true inverse cosine trans-
			y[j+1] -= sume;                 form (excepting the factor 2/N).
		}
	}
}
```

REFERENCES AND FURTHER READING:

Brigham, E. Oran. 1974, *The Fasr Fourier Transform* (Englewood Cliffs, N.J.: Prentice-Hall), §10-10.

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.

12.4 Convolution and Deconvolution Using the FFT

We have defined the ***convolution*** of two functions for the continuous case in equation (12.0.8), and have given the ***convolution theorem*** as equation (12.0.9). The theorem says that the Fourier transform of the convolution of two functions is equal to the product of their individual Fourier transforms. Now, we want to deal with the discrete case. We will mention first the context in which convolution is a useful procedure, and then discuss how to compute it efficiently using the FFT.

The convolution of two functions $r(t)$ and $s(t)$, denoted $r * s$, is mathematically equal to their convolution in the opposite order, $s * r$. Nevertheless, in most applications the two functions have quite different meanings and characters. One of the functions, say s, is typically a signal or data stream, which goes on indefinitely in time (or in whatever the appropriate independent variable may be). The other function r is a "response function," typically a peaked function that falls to zero in both directions from its maximum. The effect of convolution is to smear the signal $s(t)$ in time according to the recipe provided by the response function $r(t)$, as shown in Figure 12.4.1. In particular, a spike or delta-function of unit area in s which occurs at some time t_0 is supposed to be smeared into the shape of the response function itself, but translated from time 0 to time t_0 as $r(t - t_0)$.

In the discrete case, the signal $s(t)$ is represented by its sampled values at equal time intervals s_j. The response function is also a discrete set of numbers r_k, with the following interpretation: r_0 tells what multiple of the input signal in one channel (one particular value of j) is copied into the identical output channel (same value of j); r_1 tells what multiple of input signal in channel j is additionally copied into output channel $j + 1$; r_{-1} tells the multiple that is copied into channel $j - 1$; and so on for both positive and negative values of k in r_k. Figure 12.4.2 illustrates the situation.

Example: a response function with $r_0 = 1$ and all other r_k's equal to zero is just the identity filter: convolution of a signal with this response function gives identically the signal. Another example is the response function with $r_{14} = 1.5$ and all other r_k's equal to zero. This produces convolved output which is the input signal multiplied by 1.5 and delayed by 14 sample intervals.

Evidently, we have just described in words the following definition of discrete convolution with a response function of finite duration M:

$$(r * s)_j \equiv \sum_{k=-M/2+1}^{M/2} s_{j-k}\, r_k \qquad (12.4.1)$$

If a discrete response function is nonzero only in some range $-M/2 < k \leq M/2$, where M is a sufficiently large even integer, then the response function is called a ***finite impulse response (FIR),*** and its ***duration*** is M. (Notice that we are defining M as the number of nonzero values of r_k; these values span

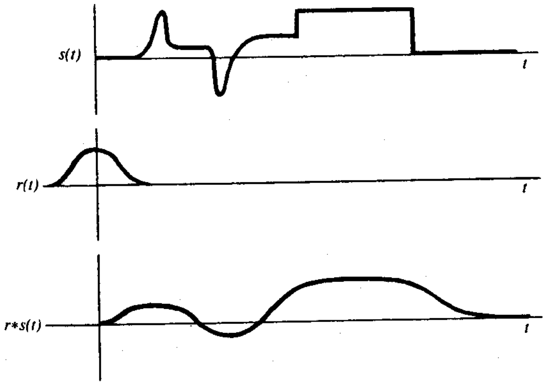

Figure 12.4.1. Example of the convolution of two functions. A signal s(t) is convolved with a response function r(t). Since the response function is broader than some features in the original signal, these are "washed out" in the convolution. In the absence of any additional noise, the process can be reversed by deconvolution.

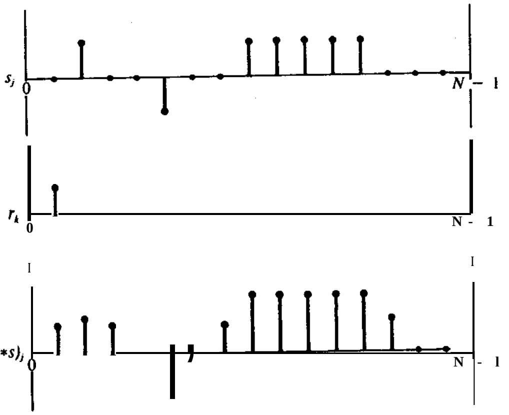

Figure 12.4.2. Convolution of discretely sampled functions. Note how the response function for negative times is wrapped around and stored at the extreme right end of the array r_k.

a time interval of $M - 1$ sampling times.) In most practical circumstances the case of finite M is the case of interest, either because the response really has a finite duration, or because we choose to truncate it at some point and approximate it by a finite-duration response function.

The ***discrete*** convolution ***theorem*** is this: If a signal s_j is ***periodic*** with period N, so that it is completely determined by the N values so,. . . , s_{N-1}, then its discrete convolution with a response function *of* finite ***duration N*** is a member of the discrete Fourier transform pair,

$$\sum_{k=-N/2+1}^{N/2} s_{j-k}\; r_k \iff S_n R_n \qquad (12.4.2)$$

Here S_n, (n = 0,. . . , N − 1) is the discrete Fourier transform of the values s_j, $(j = 0, \ldots, N - 1)$, while R_n, $(n = 0, \ldots, N - 1)$ is the discrete Fourier transform of the values r_k, $(k = 0, \ldots, N - 1)$. These values of r_k are the same ones as for the range $k = -N/2 + 1, \ldots, N/2$, but in wrap-around order, exactly as was described at the end of §12.2.

Treatment of End Effects by Zero Padding

The discrete convolution theorem presumes a set of two circumstances which are not universal. First, it assumes that the input signal is periodic, whereas real data often either go forever without repetition or else consist of one non-periodic stretch of finite length. Second, the convolution theorem takes the duration of the response to be the same as the period of the data; they are both N. We need to work around these two constraints.

The second is very straightforward. Almost always, one is interested in a response function whose duration M is much shorter than the length of the data set N. In this case, one simply extends the response function to length N by padding it with zeros, i.e. defining $r_k = 0$ for $M/2 \le k \le N/2$ and also for $-N/2 + 1 \le k \le -M/2 + 1$. Dealing with the first constraint is more challenging. Since the convolution theorem rashly assumes that the data are periodic, it will falsely "pollute" the first output channel $(r * s)_0$ with some wrapped-around data from the far end of the data stream s_{N-1}, s_{N-2}, etc. (See Figure 12.4.3.) So, we need to set up a buffer zone of zero-padded values at the end of the s_j vector, in order to make this pollution zero. How many zero values do we need in this buffer? Exactly as many as the most negative index for which the response function is nonzero. For example, if r-s is nonzero, while r-4, r-s,. . . are all zero, then we need three zero pads at the end of the data: $s_{N-3} = s_{N-2} = s_{N-1} = 0$. These zeros will protect the first output channel $(r * s)_0$ from wraparound pollution. It should be obvious that the second output channel $(r * s)_1$ and subsequent ones will also be protected

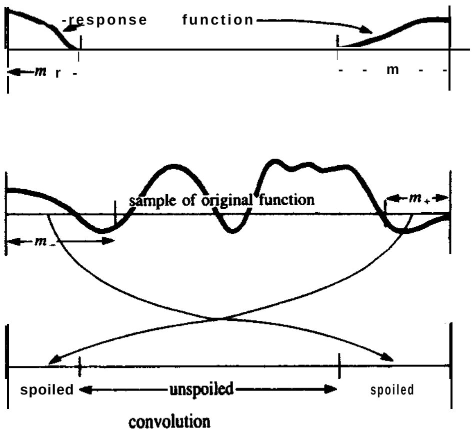

Figure 12.4.3. The wraparound problem in convolving finite segments of a function. Not only must the response function wrap be viewed as cyclic, but so must the sampled original function. Therefore a portion at each end of the original function is erroneously wrapped around by convolution with the response function.

by these same zeros. Let K denote the number of padding zeros, so that the last actual input data point is s_{N-K-1}.

What now about pollution of the very last output channel? Since the data now end with s_{N-K-1}, the last output channel of interest is $(r * s)_{N-K-1}$. This channel can be polluted by wrap around from input channel so unless the number K is also large enough to take care of the most positive index k for which the response function r_k is nonzero. For example, if r_0 through r_6 are nonzero, while $r_7, r_8 \dots$ are all zero, then we need at least $K = 6$ padding zeros at the end of the data: $s_{N-6} = \dots = s_{N-1} = 0$.

To summarize – we need to pad the data with a number of zeros on one *end* equal to the maximum positive duration or maximum negative duration of the response function, whichever is larger. (For a symmetric response function of duration M, you will need only $M/2$ zero pads.) Combining this operation with the padding of the response r_k described above, we effectively insulate the data from artifacts of undesired periodicity. Figure 12.4.4 illustrates matters.

Use of FFT for Convolution

The data, complete with zero padding, are now a set of real numbers s_j, $j = 0, \dots, N-1$, and the response function is zero padded out to duration N and arranged in wrap-around order. (Generally this means that a large contiguous section of the r_k's, in the middle of that array, is zero, with nonzero values clustered at the two extreme ends of the array.) You now compute the discrete convolution as follows: Use the FFT algorithm to compute the discrete Fourier transform of s and of r. Multiply the two transforms together component by component, remembering that the transforms consist of complex numbers. Then use the FFT algorithm to take the inverse discrete Fourier transform of the products. The answer is the convolution $r * s$.

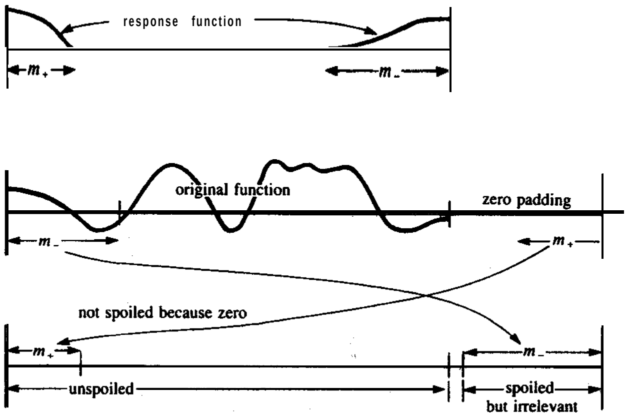

Figure 12.4.4. Zero padding as solution to the wraparound problem. The original function is extended by zeros, serving a dual purpose: when the zeros wrap around, they do not disturb the true convolution; and while the original function wraps around onto the zero region, that region can be discarded.

What about ***deconvolution?*** Deconvolution is the process of ***undoing*** the smearing in a data set which has occurred under the influence of a known response function, for example, due to the known effect of a less-than-perfect measuring apparatus. The defining equation of deconvolution is the same as that for convolution, namely (12.4.1), except now the left-hand side is taken to be known, and (12.4.1) is to be considered as a set of N linear equations for the unknown quantities s_j. Solving these simultaneous linear equations in the time domain of (12.4.1) is unrealistic in most cases, but the FFT renders the problem almost trivial. Instead of multiplying the transform of the signal and response to get the transform of the convolution, we just divide the transform of the (known) convolution by the transform of the response to get the transform of the deconvolved signal.

This procedure can go wrong *mathematically* if the transform of the response function is exactly zero for some value R_n, so that we can't divide by it. This indicates that the original convolution has truly lost all information at that one frequency, so that a reconstruction of that frequency component is not possible. You should be aware, however, that apart from mathematical problems, the process of deconvolution has other practical shortcomings. The process is generally quite sensitive to noise in the input data, and to the accuracy to which the response function r_k is known. Perfectly reasonable attempts at deconvolution can sometimes produce nonsense for these reasons. In such cases you may want to make use of the additional process of ***optimal filtering,*** which is discussed in §12.6.

Here is our routine for convolution and deconvolution, using the FFT as implemented in four1 of §12.2. Since the data and response functions are real, not complex, both of their transforms can be taken simultaneously by the technique described in S12.3. The routine thus makes just one call to compute an FFT and one call to compute an inverse FFT. The data are assumed to be stored in a float array data [1. . n], with n an integer power of two. The response function is assumed to be stored in wraparound order in a sub-array respns [1. . m] of the array respns [1. . n]. The value of m can be any *odd* integer less than or equal to n, since the first thing the program does is to recopy the response function into the appropriate wrap around order in respns [1. . n] . The answer is provided in ans.

```
static float sqrarg;
#define SQR(a) (sqrarg=(a),sqrarg*sqrarg)

void convlv(data,n,respns,m,isign,ans)
float data[],respns[],ans[];
int n,m,isign;
Convolves or deconvolves a real data set data[1. .n] (including any user-supplied zero padding)
with a response function respns [1..n] The response function must be stored in wrap around
order in the first m elements of respns, where m is an odd integer ≤ n. Wrap around order
means that the first half of the array respns contains the impulse response function at pos-
itive times, while the second half of the array contains the impulse response function at
negative times, counting down from the highest element respns[m]. On input isign is i-l
for convolution, -1 for deconvolution. The answer is returned in the first n components of
ans. However, ans must be supplied in the calling program with dimensions Cl. .2*n], for
consistency with twofft. n MUST be an integer power of two.
{
    int i,no2;
    float dum,mag2,*fft,*vector();
    void twofft(),realft(),nrerror(),free_vector();

    fft=vector(1,2*n);
    for (i=1;i<=(m-1)/2;i++)                                 Put respns in array of length n.
        respns[n+1-i]=respns[m+1-i];
    for (i=(m+3)/2;i<=n-(m-1)/2;i++)                         Pad with zeros.
        respns[i]=0.0;
    twofft(data,respns,fft,ans,n);                           FFT both at once.
    no2=n/2;
    for (i=2;i<=n+2;i+=2) {
        if (isign == 1) {
            ans[i-1]=(fft[i-1]*(dum=ans[i-1])-fft[i]*ans[i])/no2;        Multiply FFTs
            ans[i]=(fft[i]*dum+fft[i-1]*ans[i])/no2;                     to convolve.
        } else if (isign == -1) {
         if ((mag2=SQR(ans[i-1])+SQR(ans[i])) == 0.01
                nrerror("Deconvolving at response zero in CONVLV");
            ans[i-1]=(fft[i-1]*(dum=ans[i-1])+fft[i]*ans[i])/mag2/no2;   Divide FFTs
            ans[i]=(fft[i]*dum-fft[i-1]*ans[i])/mag2/no2;                tOdeconvolve.
        } else nrerror("No meaning for ISICN in CONVLV");
    }
    ans[2]=ans[n+1];                                         Pack last element with first for realft.
    realft(ans,no2,-1);                                      Inverse transform back to time domain.
    free_vector(fft,1,2*n);
}
```

Convolving or Deconvolving Very Large Data Sets

If your data set is so long that you do not want to fit it into memory all at once, then you must break it up into sections and convolve each section

separately. Now, however, the treatment of end effects is a bit different. You have to worry not only about spurious wraparound effects, but also about the fact that the ends of each section of data ***should*** have been influenced by data at the nearby ends of the immediately preceding and following sections of data, but were not so influenced since only one section of data is in the machine at a time.

There are two, related, standard solutions to this problem. Both are fairly obvious, so with a few words of description here, you ought to be able to implement them for yourself. The first solution is called the overlap-save ***method.*** In this technique you pad only the very beginning of the data with enough zeros to avoid wrap around pollution. After this initial padding, you forget about zero padding altogether. Bring in a section of data and convolve or deconvolve it. Then throw out the points at each end that are polluted by wrap around end effects. Output only, the remaining good points in the middle. Now bring in the next section of data, but not all new data. The first points the next section are to overlap points from the preceding section of data. The sections are to be overlapped sufficiently so that the polluted output points at the end of one section are recomputed as the first of the unpolluted output points from the subsequent section. With a bit of thought you can easily determine how many points to overlap and save.

The second solution, called the ***overlap-add*** method, is illustrated in Figure 12.4.5. Here you ***don't*** overlap the input data. Each section of data is disjoint from the others and is used exactly once. However, you carefully zero-pad it at both ends so that there is no wrap-around ambiguity in the output convolution or deconvolution. Now you overlap ***and add*** these sections of output. Thus, an output point near the end of one section will have the response due to the input points at the beginning of the next section of data properly added in to it, and likewise for an output point near the beginning of a section, ***mutatis*** **mutandis.**

Even when computer memory is available, there is some slight gain in computing speed in segmenting a long data set, since the FFTs' $N \log_2 N$ is slightly slower than linear in N. However, the log term is so slowly varying that you will often be much happier to avoid the bookkeeping complexities of the overlap-add or overlap-save methods: if it is practical to do so, just cram the whole data set into memory and FFT away. Then you will have more time for the finer things in life, some of which are described in succeeding sections of this chapter.

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. Oran. 1974, *The Fast Fourier Transform* (Englewood Cliffs, N.J.: Prentice-Hall), Chapter 13.

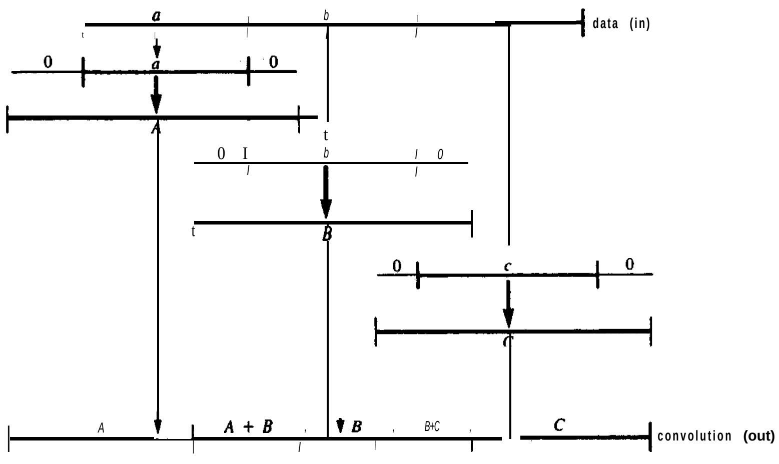

Figure 12.4.5. The overlapadd method for convolving a response with a very long signal. The signal data is broken up into smaller pieces. Each is zero padded at both ends and convolved (denoted by bold arrows in the figure). Finally the pieces are added back together, including the overlapping regions formed by the zero pads.

12.5 Correlation and Autocorrelation Using the FFT

Correlation is the close mathematical cousin of convolution. It is in some ways simpler, however, because the two functions that go into a correlation are not as conceptually distinct as were the data and response functions which entered into convolution. Rather, in correlation, the functions are represented by different, but generally similar, data sets. We investigate their "correlation," by comparing them both directly superposed, and with one of them shifted left or right.

We have already defined in equation (12.0.10) the correlation between two continuous functions $g(t)$ and $h(t)$, which is denoted $\text{Corr}(g, h)$, and is a function of lag t. We will occasionally show this time dependence explicitly, with the rather awkward notation $\text{Corr}(g, h)(t)$. The correlation will be large at some value of t if the first function (g) is a close copy of the second (h) but lags it in time by t, i.e., if the first function is shifted to the right of the second. Likewise, the correlation will be large for some negative value of t if the first function *leads* the second, i.e., is shifted to the left of the second. The relation that holds when the two functions are interchanged is

$$\text{Corr}(g, h)(t) = \text{Corr}(h, g)(-t) \tag{12.5.1}$$

The discrete correlation of two sampled functions g_k and h_k, each periodic

with period N, is defined by

$$\text{Corr}(g, h)_j \equiv \sum_{k=0}^{N-1} g_{j+k} h_k \tag{12.5.2}$$

The *discrete correlation theorem* says that this discrete correlation of two real functions g and h is one member of the discrete Fourier transform pair

$$\text{Corr}(g, h)_j \Longleftrightarrow G_k H_k{}^* \tag{12.5.3}$$

where G_k and H_k are the discrete Fourier transforms of g_j and h_j, and asterisk denotes complex conjugation. This theorem makes the same presumptions about the functions as those encountered for the discrete convolution theorem.

We can compute correlations using the FFT as follows: FFT the two data sets, multiply one resulting transform by the complex conjugate of the other, and inverse transform the product. The result (call it r_k) will formally be a complex vector of length N. However, it will turn out to have all its imaginary parts zero since the original data sets were both real. The components of r_k are the values of the correlation at different lags, with positive and negative lags stored in the by now familiar wraparound order: The correlation at zero lag is in r_0, the first component; the correlation at lag 1 is in r_1, the second component; the correlation at lag -1 is in r_{N-1}, the last component; etc.

Just as in the case of convolution we have to consider end effects, since our data will not, in general, be periodic as intended by the correlation theorem. Here again, we can use zero padding. If you are interested in the correlation for lags as large as $\pm K$, then you must append a buffer zone of K zeros at the end of both input data sets. If you want all possible lags from N data points (not a usual thing), then you will need to pad the data with an equal number of zeros; this is the extreme case. So here is the program:

```
void correl(data1,data2,n,ans)
float data1 [], data2 [], ans [];
int n;
Computes the correlation of two real data sets data1 Cl. .n] and data2[1..n], each of length
n (including any user-supplied zero padding). n MUST be an integer power of two. The answer
is returned as the first n points in ans[1..2*n] stored in wraparound order, i.e. correlations
at increasingly negative lags are in ans[n] on down to ans [n/2+1], while correlations at
increasingly positive lags are in ens [1] (zero lag) on up to ans [n/2]. Note that ans must
be supplied in the calling program with length at least 2*n, since it is also used as working
space. Sign convention of this routine: if data1 lags data2, i.e. is shifted to the right of it,
then ans will show a peak at positive lags.
{
    int no2.i;
    float dum,*fft,*vector();
    void twofft(),realft(),free_vector();

    fft=vector(1,2*n);
    tnofft(datai.data2.fft.ans.n);          Transform both data vectors at once.
    no2=n/2;                                Normalization for inverse FFT.
    for (i=2;i<=n+2;i+=2) {
        ans[i-1]=(fft[i-1]*(dum=ans[i-1])+fft[i]*ans[i])/no2;     Multiply to find FFT
                                                                  of their correlation.
        ans[i]=(fft[i]*dum-fft[i-1]*ans[i])/no2;
    }
    ans[2]=ans[n+1];                          Pack first and last into one element
    realft(ans,no2,-1);                       inverse transform gives correlation.
    free_vector(fft,1,2*n);
}
```

The ***discrete autocorrelation*** of a sampled function g_j is just the discrete correlation of the function with itself. Obviously this is always symmetric with respect to positive and negative lags. Feel free to use the above routine corre1 to obtain autocorrelations, simply calling it with the same data vector in both arguments. If the inefficiency bothers you, routine realft can, of course, be used to transform the data vector instead.

REFERENCES AND FURTHER READING:

Brigham, E. Oran. *1974, The Fasf Fourier Transform* **(Englewood Cliffs, N.J.: Prentice-Hall), §13-2.**

12.6 Optimal (Wiener) Filtering with the FFT

There are a number of other tasks in numerical processing which are routinely handled with Fourier techniques. One of these is filtering for the removal of noise from a “corrupted” signal. The particular situation we consider is this: There is some underlying, uncorrupted signal $u(t)$ that we want to measure. The measurement process is imperfect, however, and what comes out of our measurement device is a corrupted signal $c(t)$. The signal $c(t)$ may be less than perfect in either or both of two respects. First, the apparatus may not have a perfect “delta-function” response, so that the true signal $u(t)$ is convolved with (smeared out by) some known response function $r(t)$ to give a smeared signal $s(t)$,

$$s(t) = \int_{-\infty}^{\infty} r(\tau)u(t - \tau)\, d\tau \quad \text{or} \quad S(f) = R(f)U(f) \tag{12.6.1}$$

where S, R, U are the Fourier transforms of s, r, u respectively. Second, the measured signal $c(t)$ may contain an additional component of noise $n(t)$,

$$c(t) = s(t) + n(t) \tag{12.6.2}$$

We already know how to deconvolve the effects of the response function r in the absence of any noise (S12.4); we just divide C(f) by $R(f)$ to get a deconvolved signal. We now want to treat the analogous problem when noise is present. Our task is to find the ***optimal filter,*** $\phi(t)$ or $\Phi(f)$ which, when

applied to the measured signal $c(t)$ or C(f), and then deconvolved by $r(t)$ or $R(f)$, produces a signal $\tilde{u}(t)$ or $\tilde{U}(f)$ that is as close as possible to the uncorrupted signal $u(t)$ or $U(f)$. In other words we will estimate the true signal U by

$$\tilde{U}(f) = \frac{C(f)\Phi(f)}{R(f)} \tag{12.6.3}$$

In what sense is $\tilde{U}$ to be close to U? We ask that they be close ***in the*** least-square sense

$$\int_{-\infty}^{\infty} |\tilde{u}(t) - u(t)|^2\, dt = \int_{-\infty}^{\infty} \left|\tilde{U}(f) - U(f)\right|^2 df \quad \text{is minimized.} \tag{12.6.4}$$

Substituting equations (12.6.3) and (12.6.2), the right-hand side of (12.6.4) becomes

$$\begin{aligned} &\int_{-\infty}^{\infty} \left|\frac{[S(f) + N(f)]\Phi(f)}{R(f)} - \frac{S(f)}{R(f)}\right|^2 df \\ &= \int_{-\infty}^{\infty} |R(f)|^{-2} \left\{|S(f)|^2\, |1 - \Phi(f)|^2 + |N(f)|^2\, |\Phi(f)|^2\right\} df \end{aligned} \tag{12.6.5}$$

The signal S and the noise N are ***uncorrelated,*** so their cross product, when integrated over frequency f gave zero. (This is practically the ***definition*** of what we mean by noise!). Obviously (12.6.5) will be a minimum if and only if the integrand is minimized with respect to $\Phi(f)$ at every value of f. Let us search for such a solution where $\Phi(f)$ is a real function. Differentiating with respect to Φ, and setting the result equal to zero gives

$$\Phi(f) = \frac{|S(f)|^2}{|S(f)|^2 + |N(f)|^2} \tag{12.6.6}$$

This is the formula for the optimal filter $\Phi(f)$.

Notice that equation (12.6.6) involves S, the smeared signal, and N, the noise. The two of these add up to be C, the measured signal. Equation (12.6.6) does not contain U the “true” signal. This makes for an important simplification: The optimal filter can be determined independently of the determination of the deconvolution function that relates S and U.

To determine the optimal filter from equation (12.6.6) we need some way of separately estimating $|S|^2$ and $|N|^2$. There is no way to do this from the measured signal C alone without some other information, or some assumption or guess. Luckily, the extra information is often easy to obtain. For example, we can sample a long stretch of data $c(t)$ and plot its power spectral density

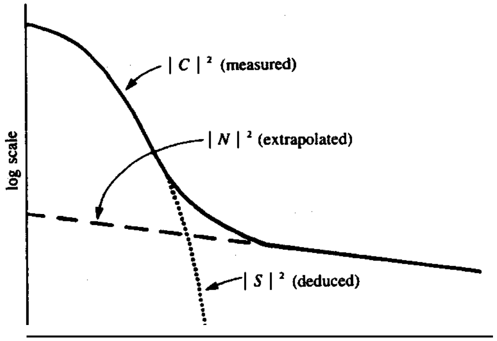

Figure 12.6.1. Optimal (Wiener) filtering. The power spectrum of signal plus noise shows a signal peak added to a noise tail. The tail is extrapolated back into the signal region as a "noise model." Subtracting gives the "signal model." The models need not be accurate for the method to be useful. A simple algebraic combination of the models gives the optimal filter (see text).

using equations (12.0.14), (12.16) and (12.1.5). This quantity is proportional to the sum $|S|^2 + |N|^2$, so we have

$$|S(f)|^2 + |N(f)|^2 \approx P_c(f) = |C(f)|^2 \qquad 0 \le f < f_c \qquad (12.6.7)$$

(More sophisticated methods of estimating the power spectral density will be discussed in §§**12.7** and 12.8, but the estimation above is almost always good enough for the optimal filter problem.) The resulting plot (see Figure 12.6.1) will often immediately show the spectral signature of a signal sticking up above a continuous noise spectrum. The noise spectrum may be flat, or tilted, or smoothly varying; it doesn't matter, as long as we can guess a reasonable hypothesis as to what it is. Draw a smooth curve through the noise spectrum, extrapolating it into the region dominated by the signal as well. Now draw a smooth curve through the signal plus noise power. The difference between these two curves is your smooth "model" of the signal power. The quotient of your model of signal power to your model of signal plus noise power is the optimal filter $\Phi(f)$. [Extend it to negative values of j by the formula $\Phi(-f) = \Phi(f)$.] Notice that $\Phi(f)$ will be close to unity where the noise is negligible, and close to zero where the noise is dominant. That is how it does its job! The intermediate dependence given by equation (12.6.6) just turns out to be the optimal way of going in between these two extremes.

Because the optimal filter results from a miniiization problem, the quality of the results obtained by optimal filtering differs from the true optimum by an amount that is ***second order*** in the precision to which the optimal filter is determined. In other words, even a fairly crudely determined optimal filter (sloppy, say, at the 10 percent level) can give excellent results when it is applied to data. That is why the separation of the measured signal C into signal and noise components S and N can usefully be done "by eye" from a crude plot of power spectral density. All of this may give you thoughts about iterating the procedure we have just described. For example, after designing a filter with response $\Phi(f)$ and using it to make a respectable guess at the signal $\widetilde{U}(f) = \Phi(f)C(f)/R($ j), you might turn about and regard $\widetilde{U}(f)$ as a fresh new signal which you could improve even further with the same filtering technique. Don't waste your time on this line of thought. The scheme converges to a signal of S(j) = 0. Converging iterative methods do exist; this just isn't one of them.

You can use the routine four1 ($12.2) or realft (512.3) to FFT your data when you are constructing an optimal filter. To apply the filter to your data, you can use the methods described in **§12.4.** The specific routine convlv is not needed for optimal filtering, since your filter is constructed in the frequency domain to begin with. If you are also deconvolving your data with a known response function, however, you can modify convlv to multiply by your optimal filter just before it takes the inverse Fourier transform.

REFERENCES AND FURTHER READING:

Rabiner, L.R., and Gold ***B. 1975, Theory and Application of Digital Signal Processing*** **(Englewood Cliffs, N.J.: Prentice-Hall).**

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).**

12.7 Power Spectrum Estimation Using the FFT

In the previous section we "informally" estimated the power spectral density of a function c(t) by taking the modulus-squared of the discrete Fourier transform of some finite, sampled stretch of it. In this section we'll do roughly the same thing, but with considerably greater attention to details. Our attention will uncover some surprises.

The first detail is power spectrum (also called a power spectral density or PSD) normalization. In general there is some relation of proportionality between a measure of the squared amplitude of the function and a measure of the amplitude of the PSD. Unfortunately there are several different conventions for describing the normalization in each domain, and many opportunities for getting wrong the relationship between the two domains. Suppose that our function $c(t)$ is sampled at N points to produce values $c_0 \ldots c_{N-1}$, and that these points span a range of time T, that is $T = (N-1)\Delta$, where A is the sampling interval. Then here are several different descriptions of the total power:

$$\sum_{j=0}^{N-1} |c_j|^2 \equiv \text{“sum squared amplitude”} \qquad (12.7.1)$$

$$\frac{1}{T}\int_0^T |c(t)|^2\, dt \approx \frac{1}{N}\sum_{j=0}^{N-1} |c_j|^2 \equiv \text{“mean squared amplitude”} \qquad (12.7.2)$$

$$\int_0^T |c(t)|^2\, dt \approx \text{A} \sum_{\text{j=o}}^{N-1} |c_j|^2 \equiv \text{“time-integral squared amplitude”} \qquad (12.7.3)$$

PSD estimators, as we shall see, have an even greater variety. In this section, we consider a class of them that give estimates at discrete values of frequency f_i, where i will range over integer values. In the next section, we will learn about a different class of estimators that produce estimates that are continuous functions of frequency f. Even if it is agreed always to relate the PSD normalization to a particular description of the function normalization (e.g. 12.7.2), there are at least the following possibilities: The PSD is

- defined for discrete positive, zero, and negative frequencies, and its sum over these is the function mean squared amplitude
- defined for zero and discrete positive frequencies only, and its sum over these is the function mean squared amplitude
- defined in the Nyquist interval from $-f_c$ to f_c, and its integral over this range is the function mean squared amplitude
- defined from 0 to f_c, and its integral over this range is the function mean squared amplitude

It never makes sense to integrate the PSD of a sampled function outside of the Nyquist interval $-f_c$ and f_c since, according to the sampling theorem, power there will have been aliased into the Nyquist interval.

It is hopeless to define enough notation to distinguish all possible combinations of normalizations. In what follows, we use the notation *P(f)* to mean any of the above PSDs, stating in each instance how the particular *P(f)* is normalized. Beware the inconsistent notation in the literature.

The method of power spectrum estimation used in the previous section is a simple version of an estimator called, historically, the *periodogram.* If we take an N-point sample of the function *c(t)* at equal intervals and use the FFT to compute its discrete Fourier transform

$$C_k = \sum_{j=0}^{N-1} c_j\, e^{2\pi ijk/N} \qquad \text{k} = 0, \ldots, \text{N} - 1 \qquad (12.7.4)$$

then the periodogram estimate of the power spectrum is defined at N/2 + 1 frequencies as

$$\begin{aligned} P(0) &= P(f_0) = \frac{1}{N^2}|C_0|^2 \\ P(f_k) &= \frac{1}{N^2}\left[|C_k|^2 + |C_{N-k}|^2\right] \qquad k = 1, 2, \ldots, \left(\frac{N}{2} - 1\right) \qquad (127.5) \\ P(f_c) &= P(f_{N/2}) = \frac{1}{N^2}|C_{N/2}|^2 \end{aligned}$$

where f_k is defined only for the zero and positive frequencies

$$f_k \equiv \frac{k}{N\Delta} = 2f_c\frac{k}{N} \qquad k = 0, 1, \ldots, \frac{N}{2} \qquad \mathbf{(12.7.6)}$$

By Parseval's theorem, equation (12.1.10), we see immediately that equation (12.7.5) is normalized so that the sum of the N/2 + 1 values of P is equal to the mean squared amplitude of the function c_j.

We must now ask this question. In what sense is the periodogram estimate (12.7.5) a "true" estimator of the power spectrum of the underlying function *c(t)?* You can find the answer treated in considerable detail in the literature cited (see, e.g., Oppenheim and Schafer for an introduction). Here is a summary.

First, is the *expectation* value of the periodogram estimate equal to the power spectrum, i.e., is the estimator correct on average? Well, yes and no. We wouldn't really expect one of the $P(f_k)$'s to equal the continuous *P(f)* at *exactly* f_k, since f_k is supposed to be representative of a whole frequency "bin" extending from halfway from the preceding discrete frequency to halfway to the next one. We *should* be expecting the $P(f_k)$ to be some kind of average of P(f) over a narrow window function centered on its f_k. For the periodogram estimate (12.7.6) that window function, as a function of *a* the frequency offset *in bins,* is

$$W(s) = \frac{1}{N^2}\left[\frac{\sin(\pi s)}{\sin(\pi s/N)}\right]^2 \qquad (12.7.7)$$

Notice that *W(a)* has oscillatory lobes but, apart from these, falls off only about as *W(a)* $\approx (\pi s)^{-2}$. This is not a very rapid fall-off, and it results in significant *leakage* (that is the technical term) from one frequency to another in the periodogram estimate. Notice also that *W(a)* happens to be zero for *a* equal to a nonzero integer. This means that if the function *c(t)* is a pure sine wave of frequency exactly equal to one of the f_k's, then there will be no leakage to adjacent f_k's. But this is not the characteristic case! If the frequency is, say, one-third of the way between two adjacent f_k's, then the leakage will extend *well* beyond those two adjacent bins. The solution to the problem of leakage is called *data* windowing, and we will discuss it below.

Turn now to another question about the periodogram estimate. What is the variance of that estimate as N goes to infinity? In other words, as we take more sampled points from the original function (either sampling a longer stretch of data at the same sampling rate, or else by resampling the same stretch of data with a faster sampling rate), then how much more accurate do the estimates P_k become? The unpleasant answer is that the periodogram estimates do *not become more accurate at all!* In fact, the variance of the periodogram estimate at a frequency f_k is always equal to the square of its expectation value at that frequency. In other words, the standard deviation is always 100 percent of the value, independent of N! How can this be? Where did all the information go as we added points? It all went into producing estimates at a greater number of discrete frequencies f_k. If we sample a longer run of data using the same sampling rate, then the Nyquist critical frequency f_c is unchanged, but we now have finer frequency resolution (more f_k's) within the Nyquist frequency interval; alternatively, if we sample the same length of data with a finer sampling interval, then our frequency resolution is unchanged, but the Nyquist range now extends up to a higher frequency. In neither case do the additional samples reduce the variance of any one particular frequency's estimated PSD.

You don't have to live with PSD estimates with 100 percent standard deviations, however. You simply have to know some techniques for reducing the variance of the estimates. Here are two techniques that are very nearly identical mathematically, though different in implementation. The first is to compute a periodogram estimate with finer discrete frequency spacing than you really need, and then to sum the periodogram estimates at K consecutive discrete frequencies to get one "smoother" estimate at the mid frequency of those K. The variance of that summed estimate will be smaller than the estimate itself by a factor of exactly $1/K$, i.e. the standard deviation will be smaller than 100 percent by a factor $1/\sqrt{K}$. Thus, to estimate the power spectrum at $M + 1$ discrete frequencies between 0 and f_c inclusive, you begin by taking the FFT of $2MK$ points (which number had better be an integer power of two!). You then take the modulus square of the resulting coefficients, add positive and negative frequency pairs and divide by $(2MK)^2$, all according to equation (12.7.5) with $N = 2MK$. Finally, you "bin" the results into summed (not averaged) groups of K. This procedure is very easy to program, so we will not bother to give a routine for it. The reason that you sum, rather than average, K consecutive points is so that your final PSD estimate will preserve the normalization property that the sum of its $M + 1$ values equals the mean square value of the function.

A second technique for estimating the PSD at $M + 1$ discrete frequencies in the range 0 to f_c is to partition the original sampled data into K segments each of $2M$ consecutive sampled points. Each segment is separately **FFT'd** to produce a periodogram estimate (equation 12.7.5 with $N = 2M$). Finally, the K periodogram estimates are averaged at each frequency. It is this final averaging that reduces the variance of the estimate by a factor K (standard deviation by $\sqrt{K}$). This second technique is computationally more efficient than the first technique above by a modest factor, since it is logarithmically more efficient to take many shorter FFTs than one longer one. The principal advantage of the second technique, however, is that only $2M$ data points are manipulated at a single time, not $2KM$ as in the first technique. This means that the second technique is the natural choice for processing long runs of data, as from a magnetic tape or other data record. We will give a routine later for implementing this second technique, but we need first to return to the matters of leakage and data windowing which were brought up after equation (12.7.7) above.

Data Windowing

The purpose of data windowing is to modify equation (12.7.7), which expresses the relation between the spectral estimate P_k at a discrete frequency and the actual underlying continuous spectrum P(f) at nearby frequencies. In general, the spectral power in one "bin" k contains leakage from frequency components that are actually s bins away, where s is the independent variable in equation (12.7.7). There is, as we pointed out, quite substantial leakage even from moderately large values of s.

When we select a run of N sampled points for periodogram spectral estimation, we are in effect multiplying an infinite run of sampled data c_j by a window function in time, one which is zero except during the total sampling time NA, and is unity during that time. In other words, the data are windowed by a square window function. By the convolution theorem (12.0.9; but interchanging the roles of f and t), the Fourier transform of the product of the data with this square window function is equal to the convolution of the data's Fourier transform with the window's Fourier transform. In fact, we determined equation (12.7.7) as nothing more than the square of the discrete Fourier transform of the unity window function.

$$W(s) = \frac{1}{N^2}\left[\frac{\sin(\pi s)}{\sin(\pi s/N)}\right]^2 = \frac{1}{N^2}\left|\sum_{k=0}^{N-1} e^{2\pi i s k/N}\right|^2 \qquad (12.7.8)$$

The reason for the leakage at large values of s, is that the square window function turns on and off so rapidly. Its Fourier transform has substantial components at high frequencies. To remedy this situation, we can multiply the input data c_j, $j = 0, \ldots, N - 1$ by a window function w_j that changes more gradually from zero to a maximum and then back to zero as j ranges from 0 to $N - 1$. In this case, the equations for the periodogram estimator (12.7.4-12.7.5) become

$$D_k \equiv \sum_{j=0}^{N-1} c_j w_j\, e^{2\pi i j k/N} \qquad k = 0, \ldots, N-1 \qquad (12.7.9)$$

$$P(0) = P(f_0) = \frac{1}{W_{ss}}\left|D_0\right|^2$$

$$P(f_k) = \frac{1}{W_{ss}}\left[\left|D_k\right|^2 + \left|D_{N-k}\right|^2\right] \qquad k = 1, 2, \ldots, \left(\frac{N}{2} - 1\right)$$

$$P(f_c) = P(f_{N/2}) = \frac{1}{W_{ss}}\left|D_{N/2}\right|^2 \tag{12.7.10}$$

where W_{ss} stands for “window squared and summed,”

$$W_{ss} \equiv N\sum_{j=0}^{N} w_j^2 \tag{12.7.11}$$

and f_k is given by (12.7.6). The more general form of (12.7.7) can now be written in terms of the window function w_j as

$$\begin{aligned} W(s) &= \frac{1}{W_{ss}}\left|\sum_{k=0}^{N-1} e^{2\pi i s k/N} w_k\right|^2 \\ &\approx \frac{1}{W_{ss}}\left|\int_{-N/2}^{N/2}\cos(2\pi s k/N)w(k-N/2)\,dk\right|^2 \end{aligned} \tag{12.7.12}$$

Here the approximate equality is useful for practical estimates, and holds for any window that is left-right symmetric (the usual case), and for $s \ll N$ (the case of interest for estimating leakage into nearby bins). The continuous function $w(k-N/2)$ in the integral is meant to be some smooth function that passes through the points w_k.

There is a lot of perhaps unnecessary lore about choice of a window function, and practically every function which rises from zero to a peak and then falls again has been named after someone. A few of the more common (also shown in Figure 12.7.1) are:

$$w_j = 1 - \frac{j-\frac{1}{2}(N-1)}{\frac{1}{2}(N+1)} \equiv \text{"Parzen window"} \tag{12.7.13}$$

(The “Bartlett window” is very similar to this.)

$$w_j = \frac{1}{2}\left[1-\cos\left(\frac{2\pi j}{N-1}\right)\right] \equiv \text{“Hanning window”} \tag{12.7.14}$$

(The “Hamming window” is similar but does not go exactly to zero at the ends.)

$$w_j = 1 - \left(\frac{j-\frac{1}{2}(N-1)}{\frac{1}{2}(N+1)}\right)^2 \equiv \text{“Welch window”} \tag{12.7.15}$$

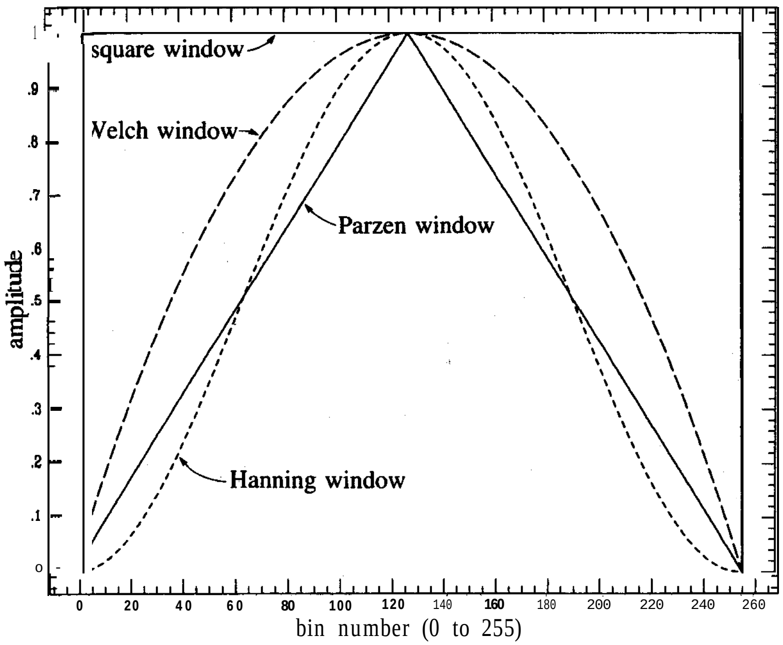

Figure 12.7.1. Window functions commonly used in FFT power spectral estimation. The data segment, here of length 256, is multiplied (bin by bin) by the window function before the FFT is computed. The square window, which is equivalent to no windowing, is least recommended. The Welch and Parzen windows are good choices.

We are inclined to follow Welch in recommending that you use either (12.7.13) or (12.7.15) in practical work. However, at the level of this book, there is effectively no ***difference*** between any of these (or similar) window functions. Their difference lies in subtle tradeoffs among the various figures of merit that can be used to describe the narrowness or peakedness of the spectral leakage functions computed by (12.7.12). These figures of merit have such names ***as highest sidelobe level (db), sidelobe fall-off (db per octave), equivalent noise bandwidth (bins), 3-db bandwidth (bins), scallop loss (db), worst case*** process ***loss (db).*** Roughly speaking, the principal tradeoff is between making the central peak as narrow as possible versus making the tails of the distribution fall off as rapidly as possible. For details, see (e.g.) Harris. Figure 12.7.2 plots the leakage amplitudes for several windows already discussed.

There is particularly a lore about window functions which rise smoothly from zero to unity in the first small fraction (say 10 percent) of the data, then stay at unity until the last small fraction (again say 10 percent) of the data, during which the window function falls smoothly back to zero. These windows will squeeze a little bit of extra narrowness out of the main lobe of the leakage function (never as much as a factor of two, however), but trade this off by widening the leakage tail by a significant factor (e.g., the reciprocal

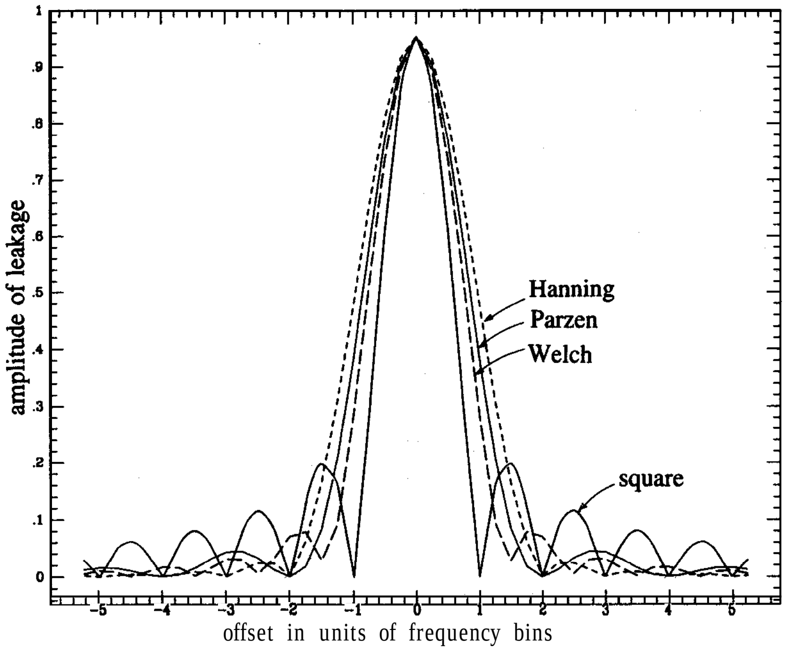

Figure 12.7.2. Leakage functions for the window functions of Figure 12.7.1. A signal whose frequency is actually located at zero offset "leaks" into neighboring bins with the amplitude shown. The purpose of windowing is to reduce the leakage at large offsets, where square (no) windowing has large sidelobes. Offset can have a fractional value, since the actual signal frequency can be located between two frequency bins of the FFT.

of 10 percent, a factor of ten). If we distinguish between the width of a window (number of samples for which it is at its maximum value) and its *rise/fall time* (number of samples during which it rises and falls); and if we distinguish between the ***FWHM*** (full width to half maximum value) of the leakage function's main lobe and the ***leakage width*** (full width that contains half of the spectral power that is not contained in the main lobe); then these quantities are related roughly by

$$\text{(FWHM in bins)} \approx \frac{N}{\text{(window width)}} \tag{12.7.16}$$

$$\text{(leakage width in bins)} \approx \frac{N}{\text{(window rise/fall time)}} \tag{12.7.17}$$

For the windows given above in (12.7.13)-(12.7.15), the effective window widths and the effective window rise/fall times are both of order $\frac{1}{2}N$. Generally speaking, we feel that the advantages of windows whose rise and fall times are only small fractions of the data length are minor or nonexistent, and

we avoid using them. One sometimes hears it said that flat-topped windows "throw away less of the data," but we will now show you a better way of dealing with that problem by use of overlapping data segments.

Let us now suppose that we have chosen a window function, and that we are ready to segment the data into K segments of $N = 2M$ points. Each segment will be FFT'd, and the resulting K periodograms will be averaged together to obtain a PSD estimate at M frequency values between 0 and f_c. We must now distinguish between two possible situations. We might want to obtain the smallest variance from a fixed amount of computation, without regard to the number of data points used. This will generally be the goal when the data are being gathered in real time, with the data-reduction being computer-limited. Alternatively, we might want to obtain the smallest variance from a fixed number of available sampled data points. This will generally be the goal in cases where the data are already recorded and we are analyzing it after the fact.

In the first situation (smallest spectral variance per computer operation), it is best to segment the data without any overlapping. The first $2M$ data points constitute segment number 1; the next $2M$ data points constitute segment number 2; and so on, up to segment number K, for a total of $2KM$ sampled points. The variance in this case, relative to a single segment, is reduced by a factor K.

In the second situation (smallest spectral variance per data point), it turns out to be optimal, or very nearly optimal, to overlap the segments by one half of their length. The first and second sets of M points are segment number 1; the second and third sets of M points are segment number 2; and so on, up to segment number K, which is made of the K^{th} and $K+1^{st}$ sets of M points. The total number of sampled points is therefore $(K+1)M$, just over half as many as with nonoverlapping segments. The reduction in the variance is not a full factor of K, since the segments are not statistically independent. It can be shown that the variance is instead reduced by a factor of about $9K/11$ (see the paper by Welch in Childers). This is, however, significantly better than the reduction of about K/2 which would have resulted if the same number of data points were segmented without overlapping.

We can now codify these ideas into a routine for spectral estimation. While we generally avoid input/output coding, we make an exception here to show how data are read sequentially in one pass through a data file (referenced through the parameter `FILE *fp`). Only a small fraction of the data is in memory at any one time.

```
#include <math.h>
#include <stdio.h>

static float sqrarg;
#define SQR(a) (sqrarg=(a),sqrarg*sqrarg)

#define WINDOW(j,a,b) (1.0-fabs((((j)-1)-(a))*(b))) /* Parzen */
/* #define WINDOW(j,a,b) 1.0 */ /* Square */
/* #define WINDOW(j,a,b) (1.0-SQR((((j)-1)-(a))*(b))) */ /* Welch */

void spctrm(fp,p,m,k,ovrlap)
FILE *fp;
```

```
float p[];
int m,k,ovrlap;
```

Reads data from input stream specified by file pointer **fp** and returns as **p[j]** the data's power (mean square amplitude) at frequency **(j-1)/(2*m)** `cycles` per gridpoint, for **j=1,2,...,m**, based on **(2*k+1)*m** data points (If `ovrlap` is set **TRUE** `(1)`) or **4*k*m** data points (If `ovrlap` is set **FALSE** (0)). The number of segments of the data Is **2*k** in both cases: the routine calls **four1** k-times, each call with 2 partitions each of **2*m** real data points.

```
{
    int mm,m44,m43,m4,kk,joffn,joff,j2,j;
    float w,facp,facm,*w1,*w2,sumw=0.0,den=0.0;
    float *vector();
    void four1(),free_vector();

    mm=m+m;                                  Useful factors.
    m43=(m4=mm+mm)+3;
    m44=m43+1;
    w1=vector(1,m4);
    w2=vector(1,m);
    facm=m-0.5;
    facp=1.0/(m+0.5);
    for (j=1;j<=mm;j++) sumw += SQR(WINDOW(j,facm,facp));
    for (j=1;j<=m;j++) p[j]=0.0;             Initialize the spectrum to zero.
    if (ovrlap)                              Initialize the "save" half-buffer.
        for (j=1;j<=m;j++) fscanf(fp,"%f",&w2[j]);
    for (kk=1;kk<=k;kk++) {                  Loop over data set segments in groups of two.
        for (joff= -1;joff<=0;joff++) {          Get two complete segments into workspace.
            if (ovrlap) {
                for (j=1;j<=m;j++) w1[joff+j+j]=w2[j];
                for (j=1;j<=m;j++) fscanf(fp,"%f",&w2[j]);
                joffn=joff+mm;
                for (j=1;j<=m;j++) w1[joffn+j+j]=w2[j];
            3 else {
                for (j=joff+2;j<=m4;j+=2)
                    fscanf(fp,"%f",&w1[j]);
            3
        3
        for (j=1; j<=mm; j++) {              Apply the window to the data.
            j2=j+j;
            w=WINDOW(j,facm,facp);
            w1[j2] *= w;
            w1[j2-1] *= w;
        3
        four1(w1,mm,1);                      Fourier transform the windowed data.
        p[1] += (SQR(w1[1])+SQR(w1[2]));         Sum results into previous segments.
        for (j=2;j<=m;j++) {
            j2=j+j;
            p[j] += (SQR(w1[j2])+SQR(w1[j2-1])
                +SQR(w1[m44-j2])+SQR(w1[m43-j2]));
        3
        den += sumw;
    3
    den *= m4;                               Correct normalization.
    for (j=1;j<=m;j++) p[j] /= den;          Normalize the output.
    free-vector(w2,l.m);
    free_vector(w1,1,m4);
3
```

REFERENCES AND FURTHER READING:

Harris, F.J. 1978, *Proceedings of the IEEE, vol. 66, p. 51.*

Childers, Donald G. (ed.). 1978, *Modern Spectrum Analysis (New* York: IEEE Press), paper by P.D. Welch.

Oppenheim, A.V., and Schafer, R.W. 1975, *Digital Signal Processing* (Englewood Cliffs, N.J.: Prentice Hall).

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).

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

Rabiner, L.R., and Gold B. 1975 *Theory and Application of Digital Signal Processing* (Englewood Cliffs, N.J.: Prentice-Hall).

12.8 Power Spectrum Estimation by the Maximum Entropy (All Poles) Method

The FFT is not the only way to estimate the power spectrum of a process, nor is it necessarily the best way for all purposes. To see how one might devise another method, let us enlarge our view for a moment, so that it includes not only real frequencies in the Nyquist interval $-f_c < f < f_c$, but also the entire complex frequency plane. From that vantage point, let us transform the complex f-plane to a new plane, called the z-transform plane or z-plane, by the relation

$$z \equiv e^{2\pi i f\Delta} \qquad (12.8.1)$$

where A is, as usual, the sampling interval in the time domain. Notice that the Nyquist interval on the real axis of the f-plane maps one-to-one onto the unit circle in the complex z-plane.

If we now compare (12.8.1) to equations (12.7.4) and (12.7.6), we see that the FFT power spectrum estimate (12.7.5) for any real sampled function $c_k \equiv c(t_k)$ can be written, except for normalization convention, as

$$P(f) = \left| \sum_{k=-N/2}^{N/2-1} c_k z^k \right|^2 \qquad (12.8.2)$$

Of course, (12.8.2) is not the true power spectrum of the underlying function c(t), but only an estimate. We can see in two related ways why the estimate is not likely to be exact. First, in the time domain, the estimate is based on only a finite range of the function c(t) which may, for all we know, have continued from $t = -\infty$ to ∞. Second, in the z-plane of equation (12.8.2), the finite Laurent series offers, in general, only an approximation to a general