Fast Fourier Transform

CHAPTER 12

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.

Historically, Fourier and spectral methods have been considered a part of "signal processing," rather than "numerical analysis" proper. There is really no justification for such a distinction. Fourier methods are commonplace in research and we will not treat them as specialized or arcane. However, we realize that many users have had relatively less experience with this field than with, say, differential equations or numerical integration. Therefore our summary of analytical results will be more complete. Numerical algorithms, per se, begin in §12.2. Various applications of Fourier transform methods are discussed in Chapter 13.

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

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

If t is measured in seconds, then f in equation (12.0.1) is in cycles per second, or Hertz (the unit of frequency). However, the equations work with other units, too. If h is a function of position x (in meters), H will be a function of inverse wavelength

(cycles per meter), and so on. If you are trained as a physicist or mathematician, you are probably more used to using *angular frequency* ω, which is given in *radians* per second. The relation between ω and f, $H(\omega)$ and $H(f)$, is

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

and equation (12.0.1) looks like this:

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

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

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

In the time domain, the 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

$$h(t) \Longleftrightarrow H(f) \tag{12.0.4}$$

is such a pair, then other transform pairs are

$$h(at) \Longleftrightarrow \frac{1}{|a|} H(\frac{f}{a}) \qquad \text{time scaling} \tag{12.0.5}$$

$$\frac{1}{|b|} h(\frac{t}{b}) \Longleftrightarrow H(bf) \qquad \text{frequency scaling} \tag{12.0.6}$$

$$h(t - t_0) \Longleftrightarrow H(f)\, e^{2\pi i f t_0} \qquad \text{time shifting} \tag{12.0.7}$$

$$h(t)\, e^{-2\pi i f_0 t} \Longleftrightarrow H(f - f_0) \qquad \text{frequency shifting} \tag{12.0.8}$$

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

$$g * h \equiv \int_{-\infty}^{\infty} g(\tau) h(t - \tau)\, d\tau \tag{12.0.9}$$

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,

$$g * h \Longleftrightarrow G(f) H(f) \qquad \text{convolution theorem} \tag{12.0.10}$$

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

$$\text{Corr}(g, h) \equiv \int_{-\infty}^{\infty} g(\tau + t) h(\tau)\, d\tau \tag{12.0.11}$$

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:

$$\text{Corr}(g, h) \Longleftrightarrow G(f) H^*(f) \qquad \text{correlation theorem} \tag{12.0.12}$$

[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 (12.0.12) becomes the transform pair

$$\text{Corr}(g, g) \Longleftrightarrow |G(f)|^2 \qquad \text{Wiener-Khinchin theorem} \tag{12.0.13}$$

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*:

$$\text{total power} \equiv \int_{-\infty}^{\infty} |h(t)|^2\, dt = \int_{-\infty}^{\infty} |H(f)|^2\, df \tag{12.0.14}$$

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

$$P_h(f) \equiv |H(f)|^2 + |H(-f)|^2 \qquad 0 \le f < \infty \tag{12.0.15}$$

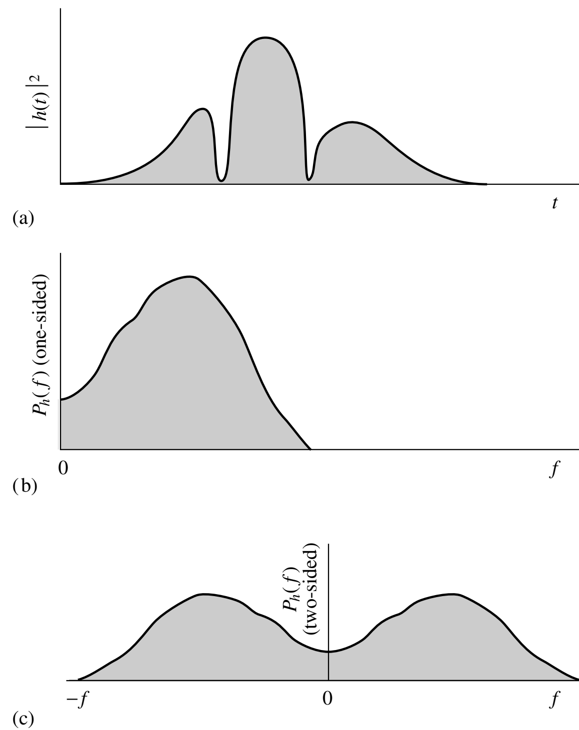

Figure 12.0.1. Normalizations of one- and two-sided power spectra. The area under the square of the function, (a), equals the area under its one-sided power spectrum at positive frequencies, (b), and also equals the area under its two-sided power spectrum at positive and negative frequencies, (c).

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, the two terms in (12.0.15) 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 (12.0.15). Figure 12.0.1 contrasts the two conventions.

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

You might well worry about how the PSD-per-unit-time, which is a function of frequency f, converges as one evaluates it using longer and longer stretches of data. This interesting question is the content of the subject of "power spectrum estimation" and will be considered below in §13.4 – §13.7. A crude answer for now is, the PSD-per-unit-time converges to finite values at all frequencies *except* those where $h(t)$ has a discrete sine-wave (or cosine-wave) component of finite amplitude. At

those frequencies, it becomes a delta-function, i.e., a sharp spike, whose width gets narrower and narrower, but whose area converges to be the mean square amplitude of the discrete sine or cosine component at that frequency.

We have by now stated all of the analytical formalism that we will need in this chapter, with one exception: In computational work, especially with experimental data, we are almost never given a continuous function $h(t)$ to work with, but are given, rather, a list of measurements of $h(t_i)$ for a discrete set of t_i's. The profound implications of this seemingly trivial fact are the subject of §12.1.

12.0.1 Higher-Order Statistics

The Wiener-Khinchin theorem, equation (12.0.13), along with the definition (12.0.11), shows that the power spectrum of a function is fully equivalent to the function's *two-point statistic*, that is, the expectation value of the product of the function at two different points separated by t. One can correspondingly define *higher-order statistics* in both the time and Fourier domains. For example, a function's *three-point correlation* is

$$\text{Corr3}(g, g, g) \equiv \int_{-\infty}^{\infty} g(\tau)g(\tau + t_1)g(\tau + t_2)\, d\tau \tag{12.0.16}$$

a function of the two variables t_1 and t_2. The two-dimensional Fourier transform (§12.5) of equation (12.0.16) over t_1 and t_2 is called the *bispectrum*, a function of two frequencies f_1 and f_2.

Higher-order statistics, including the bispectrum, can make visible non-Gaussian and nonlinear phenomena to which two-point statistics (and thus power spectra) are blind. However, they have the disadvantages of being often difficult to interpret and, because of the high powers of the signal that enter, highly susceptible to noise. On these grounds, we advise caution. Useful, if sometimes overly enthusiastic, references are [1,2,3].

CITED REFERENCES AND FURTHER READING:

Bracewell, R.N. 1999, *The Fourier Transform and Its Applications*, 3rd ed. (New York: McGraw-Hill)

Folland, G.B. 1992, *Fourier Analysis and Its Applications* (Pacific Grove, CA: Wadsworth & Brooks).

James, J.F. 2002, *A Student's Guide to Fourier Transforms*, 2nd ed. (Cambridge, UK: Cambridge University Press)

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

Brillinger, D., and Rosenblatt, M. 1967, "Computation and Intepretation of kth Order Spectra," in B. Harris, ed., *Spectral Analysis of Time Signals* (New York: Wiley).[1]

Mendel, J.M. 1991, "Tutorial on Higher-Order Statistics (Spectra) in Signal Processing and System Theory: Theoretical Results and Some Applications," *Proceedings of the IEEE*, vol. 79, pp. 278–305.[2]

Nikias, C.L., and Petropulu, A.P. 1993, *Higher-Order Spectra Analysis* (New Jersey: Prentice-Hall).[3]

12.1 Fourier Transform of Discretely Sampled Data

In the most common situations, function $h(t)$ is sampled (that is, 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.

12.1.1 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 *bandwidth limited* to frequencies smaller in magnitude than f_c, i.e., if $H(f) = 0$ for all $|f| \geq f_c$, then the function $h(t)$ is *completely determined* by its samples h_n. In fact, $h(t)$ is given explicitly by the formula

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

This is a remarkable theorem for many reasons, among them that it shows that the "information content" of a bandwidth limited function is, in some sense, infinitely smaller than that of a general continuous function. Fairly often, one is dealing with a signal that is known on physical grounds to be bandwidth limited (or at least approximately bandwidth limited). For example, the signal may have passed through a physical component 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. equation 12.1.2).

Now the bad news. The bad news concerns the effect of sampling a continuous function that is *not* bandwidth limited to less than the Nyquist critical frequency. In that case, it turns out that all of the power spectral density that lies outside of

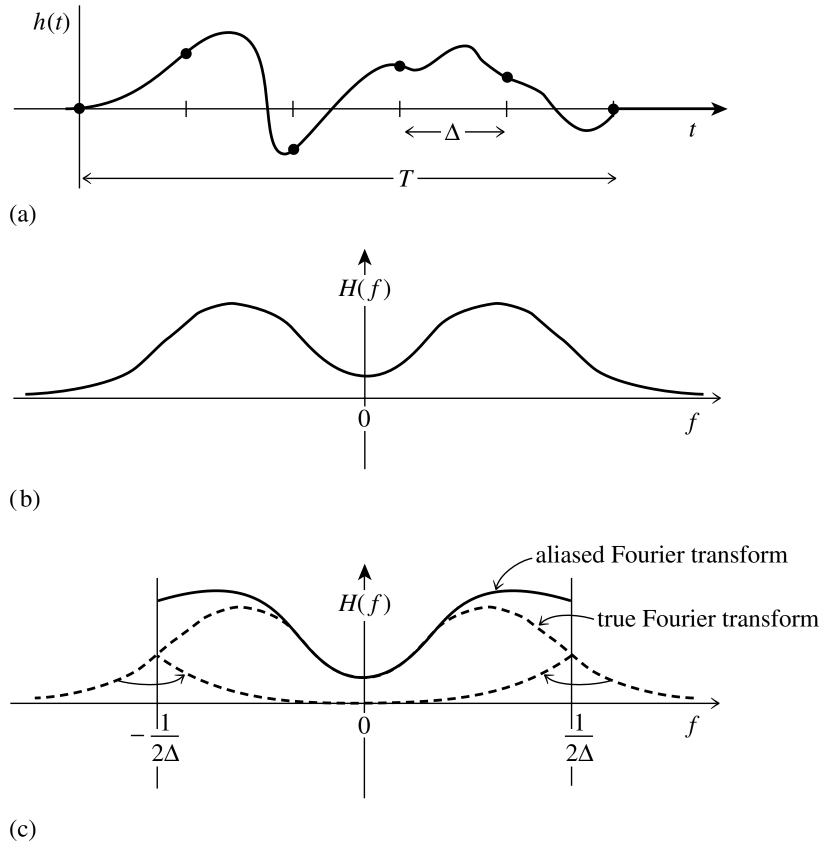

Figure 12.1.1. The continuous function shown in (a) is nonzero only for a finite interval of time T. It follows that its Fourier transform, whose modulus is shown schematically in (b), is not bandwidth limited but has finite amplitude for all frequencies. If the original function is sampled with a sampling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

the frequency range $-f_c < f < f_c$ is spuriously moved into that range. This phenomenon is called *aliasing*. Any frequency component outside of the frequency range $(-f_c, f_c)$ is *aliased* (falsely translated) into that range by the very act of discrete sampling. You can readily convince yourself that two waves $\exp(2\pi i f_1 t)$ and $\exp(2\pi i f_2 t)$ give the same samples at an interval Δ if and only if f_1 and f_2 differ by a multiple of $1/\Delta$, which is just the width in frequency of the range $(-f_c, f_c)$. There is little that you can do to remove aliased power once you have discretely sampled a signal. The way to overcome aliasing is to (i) know the natural bandwidth limit of the signal — or else enforce a known limit by analog filtering of the continuous signal, and then (ii) sample at a rate sufficiently rapid to give at least two points per cycle of the highest frequency present. Figure 12.1.1 illustrates these considerations.

To put the best face on this, we can take the alternative point of view: If a continuous function has been competently sampled, then, when we come to estimate its Fourier transform from the discrete samples, we can *assume* (or rather we *might as well* assume) that its Fourier transform is equal to zero outside of the frequency range

in between $-f_c$ and f_c. Then we look to the Fourier transform to tell whether the continuous function *has* been competently sampled (aliasing effects minimized). We do this by looking to see whether the Fourier transform is already approaching zero as the frequency approaches f_c from below or $-f_c$ from above. If, on the contrary, the transform is going toward some finite value, then chances are that components outside of the range have been folded back over onto the critical range.

12.1.2 Discrete Fourier Transform

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

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

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

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

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

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:

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

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,

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

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

$$H(f_n) \approx \Delta H_n \tag{12.1.8}$$

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 $n = 0$ and positive frequencies $0 < f < f_c$ correspond to values $1 \le n \le N/2 - 1$, while negative frequencies $-f_c < f < 0$ correspond to $N/2 + 1 \le n \le N - 1$. The value $n = N/2$ corresponds to *both* $f = f_c$ and $f = -f_c$.

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

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

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

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

The discrete form of Parseval's theorem is

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

There are also discrete analogs to the convolution and correlation theorems (equations 12.0.10 and 12.0.12), but we shall defer them to §13.1 and §13.2, respectively.

CITED REFERENCES AND FURTHER READING:

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

James, J.F. 2002, *A Student's Guide to Fourier Transforms*, 2nd ed. (Cambridge, UK: Cambridge University Press)

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

12.2 Fast Fourier Transform (FFT)

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

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

Then (12.1.7) can be written as

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

In other words, the vector of h_k's is multiplied by a matrix whose (n,k)th element is the constant W to the power $n \times k$. The matrix multiplication produces a vector result whose components are the H_n's. This matrix multiplication evidently requires N^2 complex multiplications, plus a smaller number of operations to generate the required powers of W. So, the discrete Fourier transform appears to be an $O(N^2)$ process. These appearances are deceiving! The discrete Fourier transform can, in fact, be computed in $O(N \log_2 N)$ operations with an algorithm called the *fast Fourier transform*, or *FFT*. The difference between $N \log_2 N$ and N^2 is immense. With $N = 10^8$, for example, it is a factor of several million, comparable to the ratio of one second to one month. The existence of an FFT algorithm became generally known only in the mid-1960s, from the work of J.W. Cooley and J.W. Tukey. Retrospectively, we now know (see [1]) that efficient methods for computing the DFT had been independently discovered, and in some cases implemented, by as many as a dozen individuals, starting with Gauss in 1805!

One "rediscovery" of the FFT, that of Danielson and Lanczos in 1942, provides one of the clearest derivations of the algorithm. Danielson and Lanczos showed that a discrete Fourier transform of length N can be rewritten as the sum of two discrete Fourier transforms, each of length $N/2$. One of the two is formed from the even-numbered points of the original N, the other from the odd-numbered points. The proof is simply this:

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

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

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

points that 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 2. If the length of your data set is not a power of 2, pad it with zeros up to the next power of 2. (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 one. What is the Fourier transform of length one? It is just the identity operation that copies its one input number into its one output slot! In other words, for every pattern of $\log_2 N$ e's and o's, there is a one-point transform that is just one of the input numbers f_n,

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

(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 that, 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 four-point transforms, and so on, until the first and second halves of the whole data set are combined into the final transform. Each combination takes of order N operations, and there are evidently $\log_2 N$ combinations, so the whole algorithm is of order $N \log_2 N$ (assuming, as is the case, that the process of sorting into bit-reversed order is no greater in order than $N \log_2 N$).

This, then, is the structure of an FFT algorithm: It has two sections. The first section sorts the data into bit-reversed order. Luckily this takes no additional storage, since it involves only swapping pairs of elements. (If k_1 is the bit reverse of k_2, then k_2 is the bit reverse of k_1.) The second section has an outer loop that is executed $\log_2 N$ times and calculates, in turn, transforms of length $2, 4, 8, \ldots, N$. This series of operations is often called a *butterfly*. For each stage of the process, two nested inner loops range over the subtransforms already computed and the elements of each transform, implementing the Danielson-Lanczos lemma. The operation is made more efficient by restricting external calls for trigonometric sines and cosines to the outer loop, where they are made only $\log_2 N$ times. Computation of the sines and cosines of multiple angles is through simple recurrence relations in the inner loops (cf. 5.4.6).

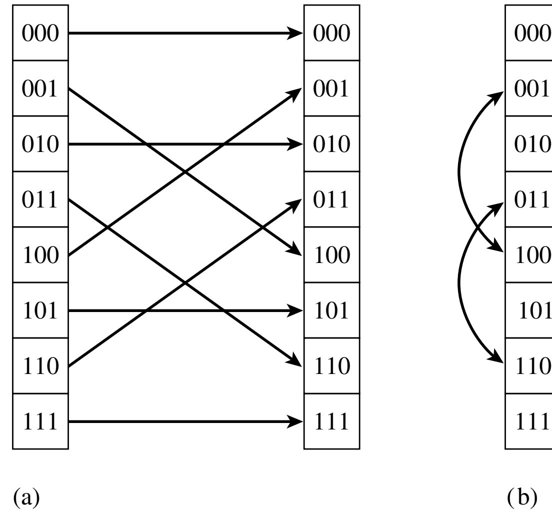

Figure 12.2.1. Reordering an array (here of length 8) by bit reversal, (a) between two arrays, versus (b) in place. Bit-reversal reordering is a necessary part of the fast Fourier transform (FFT) algorithm.

12.2.1 Bare FFT Routine and Helper Interfaces

Experience convinces us that a good way to package the FFT is as (i) a bare routine with a minimal interface, plus also (ii) a small set of interface objects that make it easier to get data in and out of the bare routine. The bare FFT routine given below is based on one originally written by N.M. Brenner. The input quantities are the number of complex data points `n` (=N), a pointer to the data array (`data[0..2*n-1]`), and `isign`, which is 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 do that yourself. We test to be sure that `n` is a power of 2 by the C++ idiom `n&(n-1)`, which is zero only if `n` is, in binary, 1 followed by any number of zeros.

Notice that the argument `n` is the number of *complex* data points. The actual length of the `Doub` array (`data[0..2*n-1]`) is 2n, with each complex value occupying two consecutive locations. In other words, `data[0]` is the real part of f_0, `data[1]` is the imaginary part of f_0, and so on up to `data[2*n-2]`, which is the real part of f_{N-1}, and `data[2*n-1]`, 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 `n` complex numbers. The real and imaginary parts of the zero frequency component F_0 are in `data[0]` and `data[1]`; the smallest nonzero positive frequency has real and imaginary parts in `data[2]` and `data[3]`; the smallest (in magnitude) nonzero negative frequency has real and imaginary parts in `data[2*n-2]` and `data[2*n-1]`. Positive frequencies increasing in magnitude are stored in the real-imaginary pairs `data[4]`, `data[5]` up to `data[n-2]`, `data[n-1]`. Negative frequencies of increasing magnitude are stored in `data[2*n-4]`, `data[2*n-3]` down to `data[n+2]`, `data[n+3]`. Finally, the pair `data[n]`, `data[n+1]` contains the real and imaginary parts of the one aliased point that contains the most positive and the most negative frequencies. You should try to develop a familiarity with this storage arrangement of

CHAPTER 13

Fourier and Spectral Applications

13.0 Introduction

Fourier methods have revolutionized fields of science and engineering, from astronomy to medical imaging, from seismology to spectroscopy. In this chapter, we present some of the basic applications of Fourier and spectral methods that have made these revolutions possible.

Say the word "Fourier" to a numericist, and the response, as if by Pavlovian conditioning, will likely be "FFT." Indeed, the wide application of Fourier methods must be credited principally to the existence of the fast Fourier transform. Better mousetraps move over: If you speed up *any* nontrivial algorithm by a factor of a million or so, the world will beat a path toward finding useful applications for it. The most direct applications of the FFT are to the convolution or deconvolution of data (§13.1), correlation and autocorrelation (§13.2), optimal filtering (§13.3), power spectrum estimation (§13.4), and the computation of Fourier integrals (§13.9).

As important as they are, however, FFT methods are not the be-all and end-all of spectral analysis. Section 13.5 is a brief introduction to the field of time-domain digital filters. In the spectral domain, one limitation of the FFT is that it always represents a function's Fourier transform as a polynomial in $z = \exp(2\pi i f \Delta)$ (cf. equation 12.1.7). Sometimes, processes have spectra whose shapes are not well represented by this form. An alternative form, which allows the spectrum to have poles in z, is used in the techniques of linear prediction (§13.6) and maximum entropy spectral estimation (§13.7).

Another significant limitation of all FFT methods is that they require the input data to be sampled at evenly spaced intervals. For irregularly or incompletely sampled data, other (albeit slower) methods are available, as discussed in §13.8.

So-called wavelet methods inhabit a representation of function space that is neither in the temporal nor in the spectral domain, but rather somewhere in-between. Section 13.10 is an introduction to this subject. Finally, §13.11 is an excursion into the numerical use of the Fourier sampling theorem.

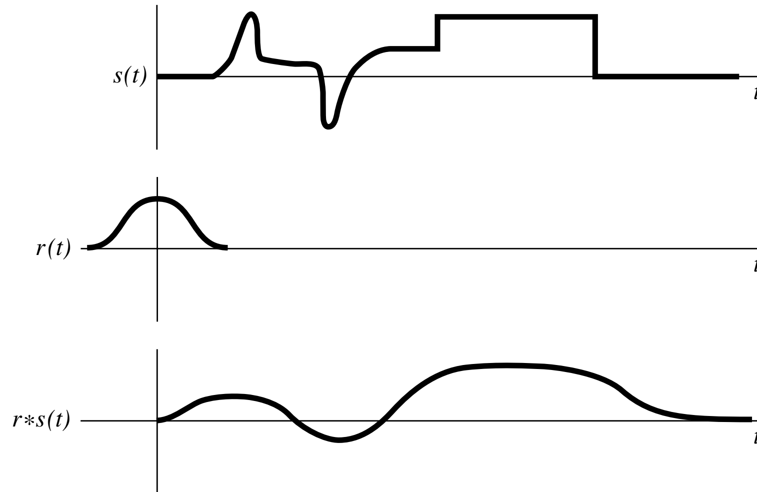

Figure 13.1.1. Example of the convolution of two functions. A signal $s(t)$ is convolved with a response function $r(t)$. Since the response function is broader than some features in the original signal, these are "washed out" in the convolution. In the absence of any additional noise, the process can be reversed by deconvolution.

13.1 Convolution and Deconvolution Using the FFT

We have defined the *convolution* of two functions for the continuous case in equation (12.0.9), and have given the *convolution theorem* as equation (12.0.10). 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 13.1.1. 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 13.1.2 illustrates the situation.

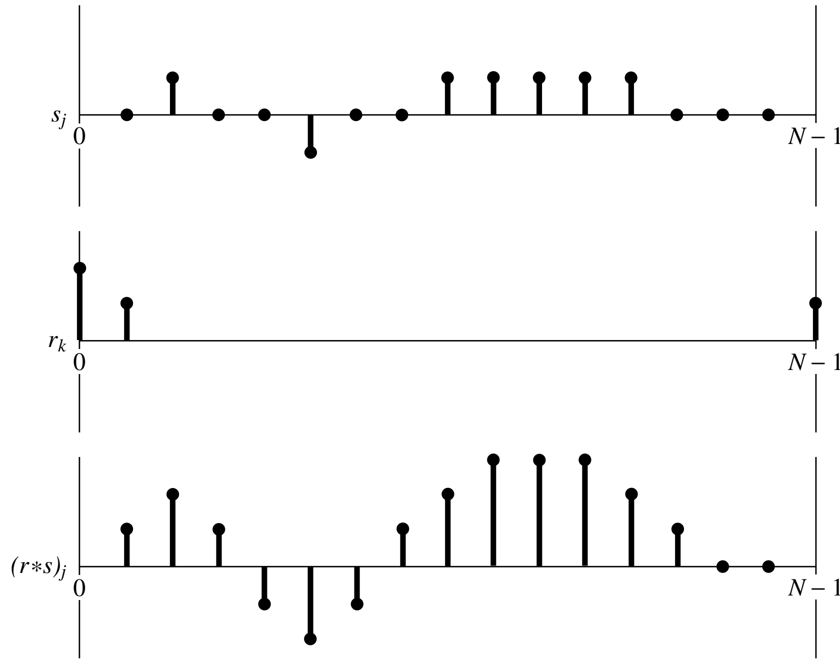

Figure 13.1.2. Convolution of discretely sampled functions. Note how the response function for negative times is wrapped around and stored at the extreme right end of the array r_k.

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 that 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:

$$(r * s)_j \equiv \sum_{k=-M/2+1}^{M/2} s_{j-k}\, r_k \tag{13.1.1}$$

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 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 $s_0, \ldots, s_{N-1}$, then its discrete convolution with a response function *of finite duration* N is a member of the discrete Fourier transform pair,

$$\sum_{k=-N/2+1}^{N/2} s_{j-k}\, r_k \quad \Longleftrightarrow \quad S_n R_n \tag{13.1.2}$$

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

13.1.1 Treatment of End Effects by Zero Padding

The discrete convolution theorem presumes a set of two circumstances that 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 nonperiodic 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, you simply extend the response function to length N by padding it with zeros, i.e., define $r_k = 0$ for $M/2 \le k \le N/2$ and also for $-N/2 + 1 \le\le -M/2 + 1$. 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 13.1.3.) 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_{-3} is nonzero while $r_{-4}, r_{-5}, \dots$ 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 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 wraparound from input channel s_0 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 13.1.4 illustrates matters.

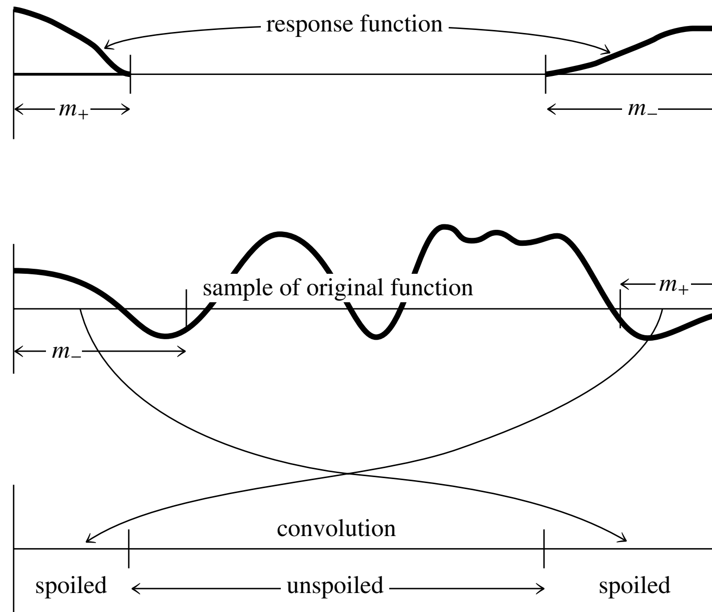

Figure 13.1.3. The wraparound problem in convolving finite segments of a function. Not only must the response function wrap be viewed as cyclic, but so must the sampled original function. Therefore, a portion at each end of the original function is erroneously wrapped around by convolution with the response function.

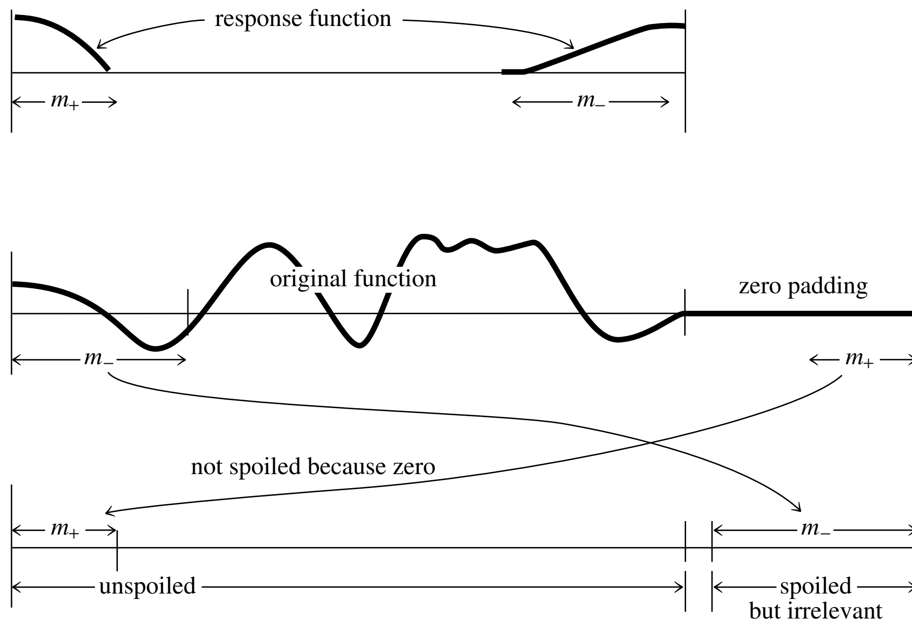

Figure 13.1.4. Zero-padding as solution to the wraparound problem. The original function is extended by zeros, serving a dual purpose: When the zeros wrap around, they do not disturb the true convolution; and while the original function wraps around onto the zero region, that region can be discarded.

13.1.2 Use of FFT for Convolution

The data, complete with zero-padding, are now a set of real numbers s_j, $j = 0, \ldots, N-1$, and the response function is zero-padded out to duration N and arranged in wraparound 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 $r * s$.

What about *deconvolution*? Deconvolution is the process of *undoing* the smearing in a data set that has occurred under the influence of a known response function, for example, because of the known effect of a less-than-perfect measuring apparatus. The defining equation of deconvolution is the same as that for convolution, namely (13.1.1), except now the left-hand side is taken to be known and (13.1.1) 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 (13.1.1) 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 §13.3.

Here is our routine for convolution and deconvolution, using the FFT as implemented in `realft` (§12.3). The data are assumed to be stored in a `VecDoub` array `data[0..n-1]`, with `n` an integer power of 2. The response function is assumed to be stored in wraparound order in a `VecDoub` array `respns[0..m-1]`. 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 wraparound order in an array of length `n`. The answer is provided in `ans`, which is also used as working space.

convlv.h

```
void convlv(VecDoub_I &data, VecDoub_I &respns, const Int isign,
    VecDoub_O &ans) {
```

Convolves or deconvolves a real data set `data[0..n-1]` (including any user-supplied zero padding) with a response function `respns[0..m-1]`, where `m` is an odd integer $\leq$ `n`. The response function must be stored in wraparound order: The first half of the array `respns` contains the impulse response function at positive times, while the second half of the array contains the impulse response function at negative times, counting down from the highest element `respns[m-1]`. On input `isign` is $+1$ for convolution, -1 for deconvolution. The answer is returned in `ans[0..n-1]`. `n` must be an integer power of 2.

```
    Int i,no2,n=data.size(),m=respns.size();
```

```
    Doub mag2,tmp;
    VecDoub temp(n);
    temp[0]=respns[0];
    for (i=1;i<(m+1)/2;i++) {                  Put respns in array of length n.
        temp[i]=respns[i];
        temp[n-i]=respns[m-i];
    }
    for (i=(m+1)/2;i<n-(m-1)/2;i++)            Pad with zeros.
        temp[i]=0.0;
    for (i=0;i<n;i++)
        ans[i]=data[i];
    realft(ans,1);                             FFT both arrays.
    realft(temp,1);
    no2=n>>1;
    if (isign == 1) {
        for (i=2;i<n;i+=2) {                   Multiply FFTs to convolve.
            tmp=ans[i];
            ans[i]=(ans[i]*temp[i]-ans[i+1]*temp[i+1])/no2;
            ans[i+1]=(ans[i+1]*temp[i]+tmp*temp[i+1])/no2;
        }
        ans[0]=ans[0]*temp[0]/no2;
        ans[1]=ans[1]*temp[1]/no2;
    } else if (isign == -1) {
        for (i=2;i<n;i+=2) {                   Divide FFTs to deconvolve.
            if ((mag2=SQR(temp[i])+SQR(temp[i+1])) == 0.0)
                throw("Deconvolving at response zero in convlv");
            tmp=ans[i];
            ans[i]=(ans[i]*temp[i]+ans[i+1]*temp[i+1])/mag2/no2;
            ans[i+1]=(ans[i+1]*temp[i]-tmp*temp[i+1])/mag2/no2;
        }
        if (temp[0] == 0.0 || temp[1] == 0.0)
            throw("Deconvolving at response zero in convlv");
        ans[0]=ans[0]/temp[0]/no2;
        ans[1]=ans[1]/temp[1]/no2;
    } else throw("No meaning for isign in convlv");
    realft(ans,-1);                            Inverse transform back to time domain.
}
```

13.1.3 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 wraparound 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 wraparound 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 in each new section overlap the last points from the preceding section of data. The sections must be overlapped sufficiently so that the polluted output points

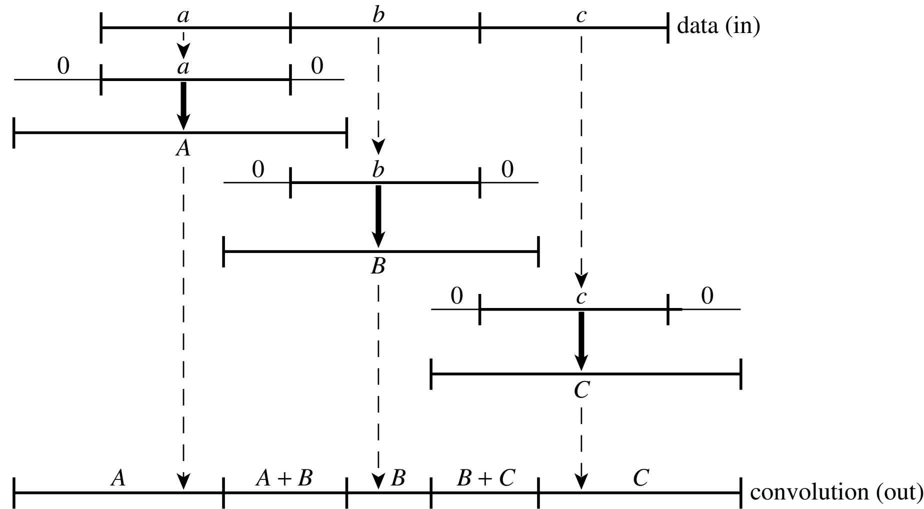

Figure 13.1.5. The overlap-add method for convolving a response with a very long signal. The signal data are broken up into smaller pieces. Each is zero-padded at both ends and convolved (denoted by bold arrows in the figure). Finally the pieces are added back together, including the overlapping regions formed by the zero-pads.

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 13.1.5. 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 wraparound 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.

CITED REFERENCES AND FURTHER READING:

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

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

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 13.

13.2 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 that 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 (12.0.11) 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

$$\text{Corr}(g,h)(t) = \text{Corr}(h,g)(-t) \tag{13.2.1}$$

The discrete correlation of two sampled functions g_k and h_k, each periodic with period N, is defined by

$$\text{Corr}(g,h)_j \equiv \sum_{k=0}^{N-1} g_{j+k} h_k \tag{13.2.2}$$

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

$$\text{Corr}(g,h)_j \iff G_k H_k^* \tag{13.2.3}$$

where G_k and H_k are the discrete Fourier transforms of g_j and h_j, and the 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:

correl.h

```
void correl(VecDoub_I &data1, VecDoub_I &data2, VecDoub_O &ans) {
```
Computes the correlation of two real data sets `data1[0..n-1]` and `data2[0..n-1]` (including any user-supplied zero padding). `n` must be an integer power of 2. The answer is returned in `ans[0..n-1]` stored in wraparound order, i.e., correlations at increasingly negative lags are in `ans[n-1]` on down to `ans[n/2]`, while correlations at increasingly positive lags are in `ans[0]` (zero lag) on up to `ans[n/2-1]`. 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,n=data1.size();
    Doub tmp;
    VecDoub temp(n);
    for (i=0;i<n;i++) {
        ans[i]=data1[i];
        temp[i]=data2[i];
    }
    realft(ans,1);                          Transform both data vectors.
    realft(temp,1);
    no2=n>>1;                               Normalization for inverse FFT.
    for (i=2;i<n;i+=2) {                    Multiply to find FFT of their correlation.
        tmp=ans[i];
        ans[i]=(ans[i]*temp[i]+ans[i+1]*temp[i+1])/no2;
        ans[i+1]=(ans[i+1]*temp[i]-tmp*temp[i+1])/no2;
    }
    ans[0]=ans[0]*temp[0]/no2;
    ans[1]=ans[1]*temp[1]/no2;
    realft(ans,-1);                         Inverse transform gives correlation.
}
```

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 `correl` to obtain autocorrelations, simply calling it with the same `data` vector in both arguments. If the inefficiency bothers you, you can edit the program so that only one call is made to `realft` for the forward transform.

CITED REFERENCES AND FURTHER READING:

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall), §13–2.

13.3 Optimal (Wiener) Filtering with the FFT

There are a number of other tasks in numerical processing that 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 $s(t)$,

$$s(t) = \int_{-\infty}^{\infty} r(t-\tau)u(\tau)\,d\tau \quad \text{or} \quad S(f) = R(f)U(f) \tag{13.3.1}$$