Chapter 13. Fourier and Spectral Applications

13.0 Introduction

Fourier methods have revolutionized fields of science and engineering, from radio 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 stand aside: If you speed up *any* nontrivial algorithm by a factor of a million or so, the world will beat a path towards 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 something in-between. Section 13.10 is an introduction to this subject. Finally §13.11 is an excursion into numerical use of the Fourier sampling theorem.

13.1 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 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.

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:

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

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

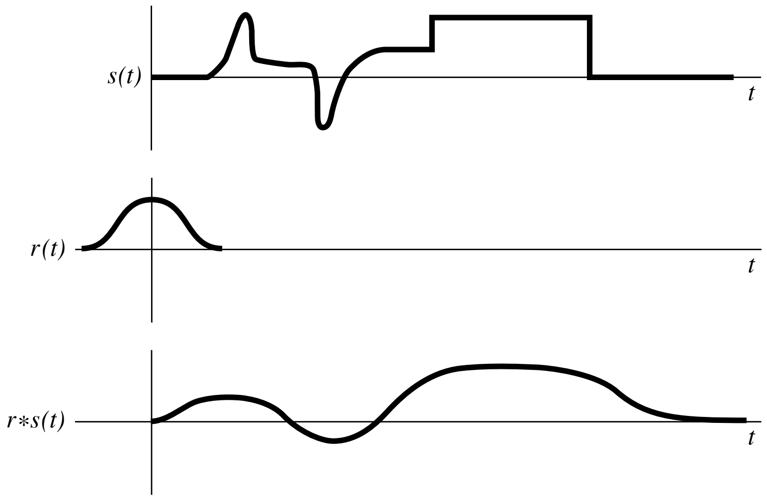


Figure 13.1.1. Example of the convolution of two functions. A signal $s(t)$ is convolved with a response function $r(t)$. Since the response function is broader than some features in the original signal, these are "washed out" in the convolution. In the absence of any additional noise, the process can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

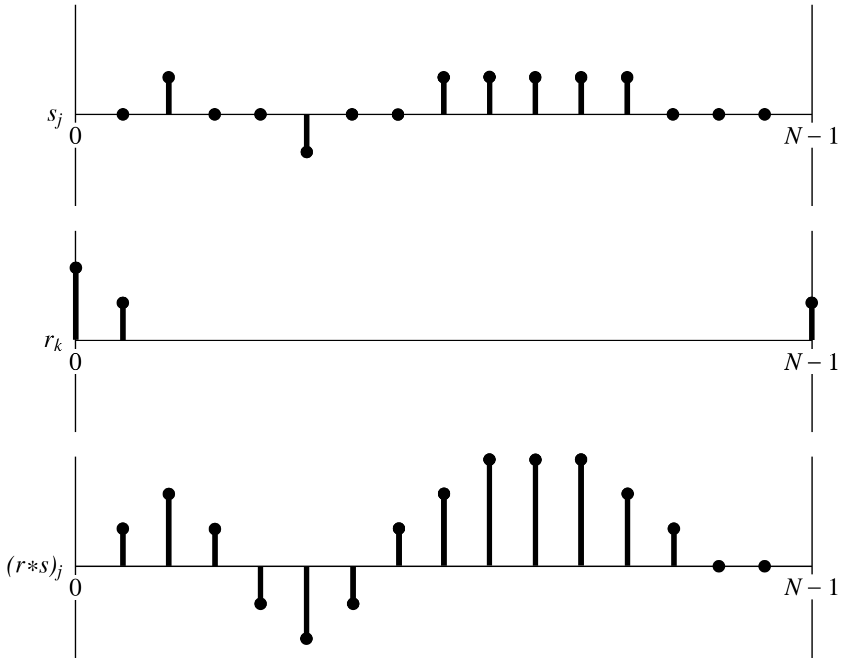


Figure 13.1.2. Convolution of discretely sampled functions. Note how the response function for negative times is wrapped around and stored at the extreme right end of the array r_k.

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 ones as for the range $k = -N/2+1, \dots, N/2$, 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 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 $-N/2+1 \le k \le -M/2+1$. 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 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 wrap-around 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 wrap-around 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

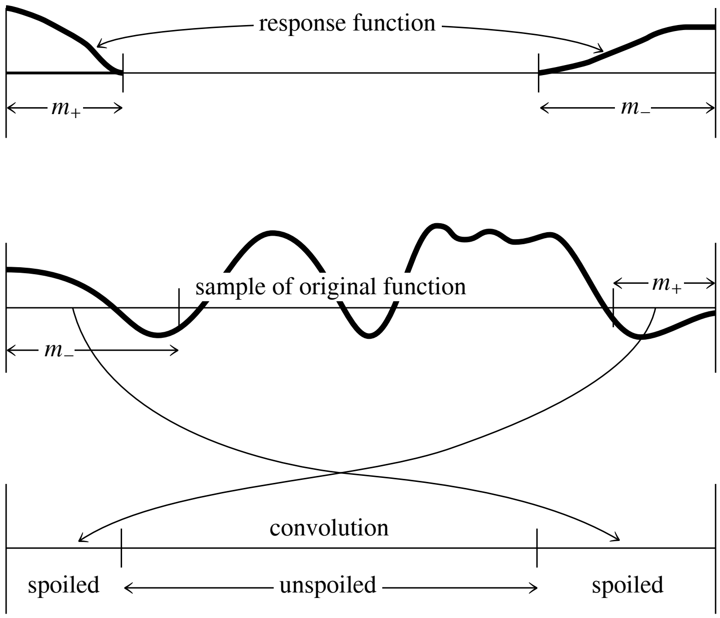


Figure 13.1.3. The wrap-around problem in convolving finite segments of a function. Not only must the response function wrap be viewed as cyclic, but so must the sampled original function. Therefore a portion at each end of the original function is erroneously wrapped around by convolution with the response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

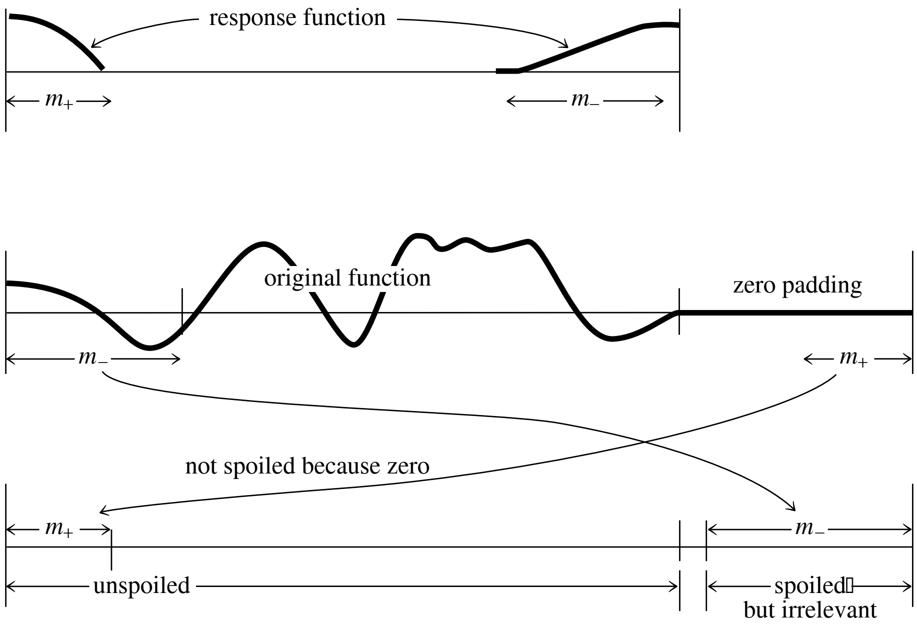


Figure 13.1.4. Zero padding as solution to the wrap-around problem. The original function is extended by zeros, serving a dual purpose: When the zeros wrap around, they do not disturb the true convolution; and while the original function wraps around onto the zero region, that region can be discarded.

padding of the response r_k described above, we effectively insulate the data from artifacts of undesired periodicity. Figure 13.1.4 illustrates matters.

Use of FFT for Convolution

The data, complete with zero padding, are now a set of real numbers s_j, $j = 0, \ldots, N-1$, 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 $r*s$.

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 `four1` of §12.2. Since the data and response functions are real, not complex, both of their transforms can be taken simultaneously using `twofft`. Note, however, that two calls to `realft` should be substituted if `data` and `respns` have very different magnitudes, to minimize roundoff. 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 wrap-around 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`.

```
#include "nrutil.h"

void convlv(float data[], unsigned long n, float respns[], unsigned long m,
    int isign, float ans[])
```
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 $\leq$ `n`. Wrap-around order means that 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]`. On input `isign` is $+1$ 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 `[1..2*n]`, for consistency with `twofft`. `n` MUST be an integer power of two.

```
{
    void realft(float data[], unsigned long n, int isign);
    void twofft(float data1[], float data2[], float fft1[], float fft2[],
        unsigned long n);
    unsigned long i,no2;
    float dum,mag2,*fft;

    fft=vector(1,n<<1);
    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>>1;
    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])) == 0.0)
                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;              to deconvolve.
        } else nrerror("No meaning for isign in convlv");
    }
    ans[2]=ans[n+1];                    Pack last element with first for realft.
    realft(ans,n,-1);                   Inverse transform back to time domain.
    free_vector(fft,1,n<<1);
}
```

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 wrap-around 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

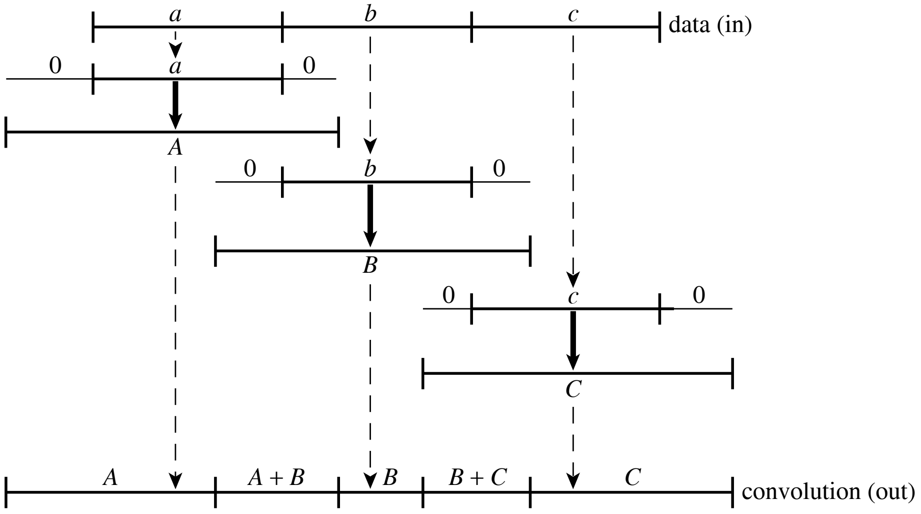


Figure 13.1.5. The overlap-add method for convolving a response with a very long signal. The signal data is broken up into smaller pieces. Each is zero padded at both ends and convolved (denoted by bold arrows in the figure). Finally the pieces are added back together, including the overlapping regions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

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 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 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.

CITED REFERENCES AND FURTHER READING:

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

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

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall), 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 (12.0.10) the correlation between two continuous functions $g(t)$ and $h(t)$, which is denoted Corr(g, h), and is a function of *lag* t. We will occasionally show this time dependence explicitly, with the rather awkward notation 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

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

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 wrap-around 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:

```
#include "nrutil.h"

void correl(float data1[], float data2[], unsigned long n, float ans[])
```
Computes the correlation of two real data sets `data1[1..n]` and `data2[1..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 wrap-around 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 `ans[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.
```
{
    void realft(float data[], unsigned long n, int isign);
    void twofft(float data1[], float data2[], float fft1[], float fft2[],
        unsigned long n);
    unsigned long no2,i;
    float dum,*fft;

    fft=vector(1,n<<1);
    twofft(data1,data2,fft,ans,n);          Transform both data vectors at once.
    no2=n>>1;                               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
        ans[i]=(fft[i]*dum-fft[i-1]*ans[i])/no2;                  FFT of their cor-
    }                                                             relation.
    ans[2]=ans[n+1];                        Pack first and last into one element.
    realft(ans,n,-1);                       Inverse transform gives correlation.
    free_vector(fft,1,n<<1);
}
```

As in `convlv`, it would be better to substitute two calls to `realft` for the one call to `twofft`, if `data1` and `data2` have very different magnitudes, to minimize roundoff error.

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, routine `realft` can, of course, be used to transform the `data` vector instead.

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 $s(t)$,

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

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 $n(t)$,

$$c(t) = s(t) + n(t) \tag{13.3.2}$$

We already know how to deconvolve the effects of the response function r in the absence of any noise (§13.1); 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 $\widetilde{u}(t)$ or $\widetilde{U}(f)$ 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

$$\widetilde{U}(f) = \frac{C(f)\Phi(f)}{R(f)} \tag{13.3.3}$$

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

$$\int_{-\infty}^{\infty} |\widetilde{u}(t) - u(t)|^2 \, dt = \int_{-\infty}^{\infty} \left|\widetilde{U}(f) - U(f)\right|^2 df \quad \text{is minimized.} \tag{13.3.4}$$

Substituting equations (13.3.3) and (13.3.2), the right-hand side of (13.3.4) becomes

$$\begin{aligned} &\int_{-\infty}^{\infty} \left| \frac{[S(f)+N(f)]\Phi(f)}{R(f)} - \frac{S(f)}{R(f)} \right|^2 df \\ &= \int_{-\infty}^{\infty} |R(f)|^{-2} \left\{ |S(f)|^2 \, |1-\Phi(f)|^2 + |N(f)|^2 \, |\Phi(f)|^2 \right\} df \end{aligned} \tag{13.3.5}$$

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 (13.3.5) 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

$$\Phi(f) = \frac{|S(f)|^2}{|S(f)|^2 + |N(f)|^2} \tag{13.3.6}$$

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

Notice that equation (13.3.6) involves S, the smeared signal, and N, the noise. The two of these add up to be C, the measured signal. Equation (13.3.6) 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 (13.3.6) 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 using equations (12.0.14), (12.1.8), and (12.1.5). This quantity is proportional to the sum $|S|^2 + |N|^2$, so we have

$$|S(f)|^2 + |N(f)|^2 \approx P_c(f) = |C(f)|^2 \qquad 0 \le f < f_c \tag{13.3.7}$$

(More sophisticated methods of estimating the power spectral density will be discussed in §13.4 and §13.7, but the estimation above is almost always good enough for the optimal filter problem.) The resulting plot (see Figure 13.3.1) 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 f 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 (13.3.6) just turns out to be the optimal way of going in between these two extremes.

Because the optimal filter results from a minimization 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 $\widetilde{U}(f) = \Phi(f)C(f)/R(f)$, 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.

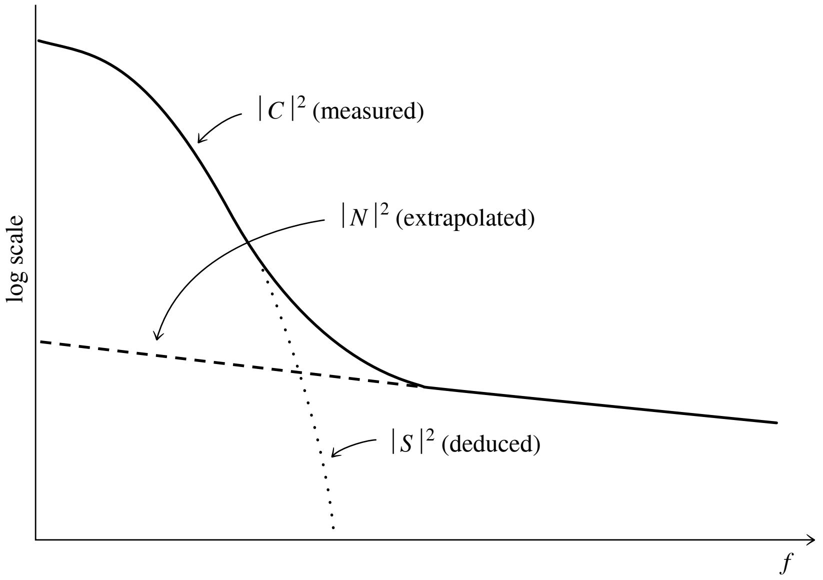


Figure 13.3.1. Optimal (Wiener) filtering. The power spectrum of signal plus noise shows a signal peak added to a noise tail. The tail is extrapolated back into the signal region as a "noise model." Subtracting gives the "signal model." The models need not be accurate for the method to be useful. A simple algebraic combination of the models gives the optimal filter (see text).

Don't waste your time on this line of thought. The scheme converges to a signal of $S(f) = 0$. Converging iterative methods do exist; this just isn't one of them.

You can use the routine `four1` (§12.2) or `realft` (§12.3) 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 §13.1. 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.

CITED REFERENCES AND FURTHER READING:

Rabiner, L.R., and Gold, B. 1975, *Theory and Application of Digital Signal Processing* (Englewood Cliffs, NJ: 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).

13.4 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 $c_0 \dots c_{N-1}$, and that these points span a range of time T, that is $T = (N-1)\Delta$, where Δ is the sampling interval. Then here are several different descriptions of the total power:

$$\sum_{j=0}^{N-1} |c_j|^2 \equiv \text{“sum squared amplitude”} \tag{13.4.1}$$

$$\frac{1}{T}\int_0^T |c(t)|^2 \, dt \approx \frac{1}{N}\sum_{j=0}^{N-1} |c_j|^2 \equiv \text{“mean squared amplitude”} \tag{13.4.2}$$

$$\int_0^T |c(t)|^2 \, dt \approx \Delta \sum_{j=0}^{N-1} |c_j|^2 \equiv \text{“time-integral squared amplitude”} \tag{13.4.3}$$

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., 13.4.2), 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

$$C_k = \sum_{j=0}^{N-1} c_j \, e^{2\pi ijk/N} \qquad k = 0, \ldots, N-1 \tag{13.4.4}$$

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

$$\begin{aligned} P(0) = P(f_0) &= \frac{1}{N^2} |C_0|^2 \\ P(f_k) &= \frac{1}{N^2} \left[|C_k|^2 + |C_{N-k}|^2 \right] \qquad k = 1, 2, \ldots, \left(\frac{N}{2} - 1 \right) \\ P(f_c) = P(f_{N/2}) &= \frac{1}{N^2} \left| C_{N/2} \right|^2 \end{aligned} \tag{13.4.5}$$

where f_k is defined only for the zero and positive frequencies

$$f_k \equiv \frac{k}{N\Delta} = 2f_c \frac{k}{N} \qquad k = 0, 1, \ldots, \frac{N}{2} \tag{13.4.6}$$

By Parseval's theorem, equation (12.1.10), we see immediately that equation (13.4.5) 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 (13.4.5) 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., [1] 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 (13.4.6) that window function, as a function of s the frequency offset *in bins*, is

$$W(s) = \frac{1}{N^2} \left[\frac{\sin(\pi s)}{\sin(\pi s/N)} \right]^2 \tag{13.4.7}$$

Notice that $W(s)$ has oscillatory lobes but, apart from these, falls off only about as $W(s) \approx (\pi s)^{-2}$. 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 $W(s)$ happens to be zero for s 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 (13.4.5) 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 13.4.5 with $N \equiv 2M$). 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 (13.4.7) above.

Data Windowing

The purpose of data windowing is to modify equation (13.4.7), 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 (13.4.7). 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 that is zero except during the total sampling time $N\Delta$, 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 (13.4.7) as nothing more than the square of the discrete Fourier transform of the unity window function.

$$W(s) = \frac{1}{N^2}\left[\frac{\sin(\pi s)}{\sin(\pi s/N)}\right]^2 = \frac{1}{N^2}\left|\sum_{k=0}^{N-1} e^{2\pi i s k/N}\right|^2 \tag{13.4.8}$$

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 $c_j,\ j = 0, \dots, N-1$ 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. In this case, the equations for the periodogram estimator (13.4.4–13.4.5) become

$$D_k \equiv \sum_{j=0}^{N-1} c_j w_j\, e^{2\pi i j k/N} \qquad k = 0, \dots, N-1 \tag{13.4.9}$$

$$\begin{aligned}
P(0) &= P(f_0) = \frac{1}{W_{ss}}\,|D_0|^2 \\
P(f_k) &= \frac{1}{W_{ss}}\left[|D_k|^2 + |D_{N-k}|^2\right] \qquad k = 1, 2, \dots, \left(\frac{N}{2} - 1\right) \\
P(f_c) &= P(f_{N/2}) = \frac{1}{W_{ss}}\,\left|D_{N/2}\right|^2
\end{aligned} \tag{13.4.10}$$

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

$$W_{ss} \equiv N \sum_{j=0}^{N-1} w_j^2 \tag{13.4.11}$$

and f_k is given by (13.4.6). The more general form of (13.4.7) can now be written in terms of the window function w_j as

$$\begin{aligned} W(s) &= \frac{1}{W_{ss}} \left| \sum_{k=0}^{N-1} e^{2\pi i s k/N} w_k \right|^2 \\ &\approx \frac{1}{W_{ss}} \left| \int_{-N/2}^{N/2} \cos(2\pi s k/N) w(k - N/2)\, dk \right|^2 \end{aligned} \tag{13.4.12}$$

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 that 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 13.4.1) are:

$$w_j = 1 - \left| \frac{j - \frac{1}{2}N}{\frac{1}{2}N} \right| \equiv \text{“Bartlett window”} \tag{13.4.13}$$

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

$$w_j = \frac{1}{2}\left[1 - \cos\left(\frac{2\pi j}{N}\right)\right] \equiv \text{“Hann window”} \tag{13.4.14}$$

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

$$w_j = 1 - \left(\frac{j - \frac{1}{2}N}{\frac{1}{2}N} \right)^2 \equiv \text{“Welch window”} \tag{13.4.15}$$

We are inclined to follow Welch in recommending that you use either (13.4.13) or (13.4.15) 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 trade-offs among the various figures of merit that can be used to describe the narrowness or peakedness of the spectral leakage functions computed by (13.4.12). 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 trade-off 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.) [2]. Figure 13.4.2 plots the leakage amplitudes for several windows already discussed.

There is particularly a lore about window functions that 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

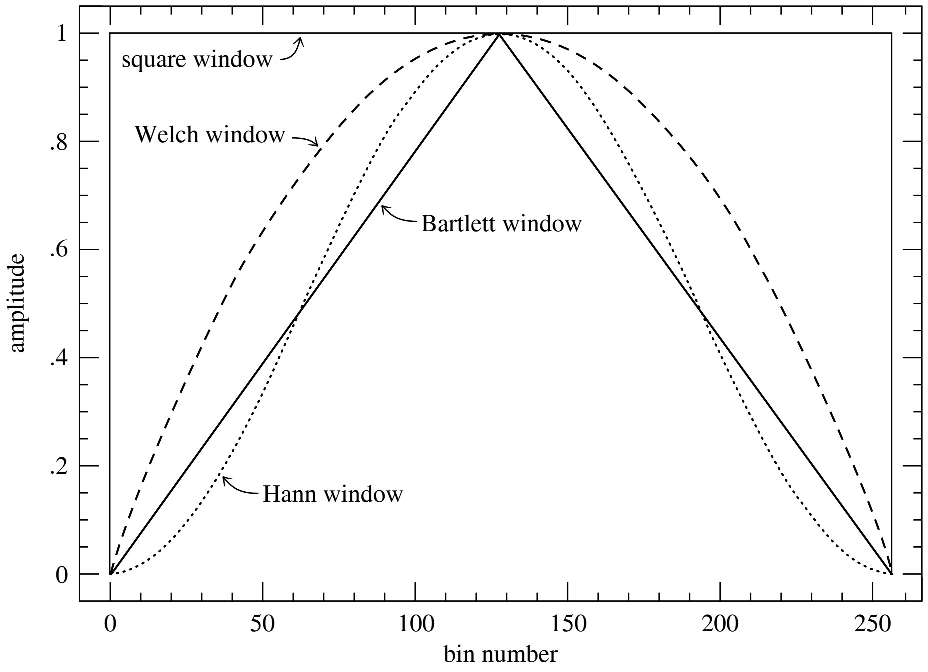


Figure 13.4.1. Window functions commonly used in FFT power spectral estimation. The data segment, here of length 256, is multiplied (bin by bin) by the window function before the FFT is computed. The square window, which is equivalent to no windowing, is least recommended. The Welch and Bartlett windows are good choices.

much as a factor of two, however), but trade this off by widening the leakage tail by a significant factor (e.g., the reciprocal 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

$$(\text{FWHM in bins}) \approx \frac{N}{(\text{window width})} \tag{13.4.16}$$

$$(\text{leakage width in bins}) \approx \frac{N}{(\text{window rise/fall time})} \tag{13.4.17}$$

For the windows given above in (13.4.13)–(13.4.15), 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

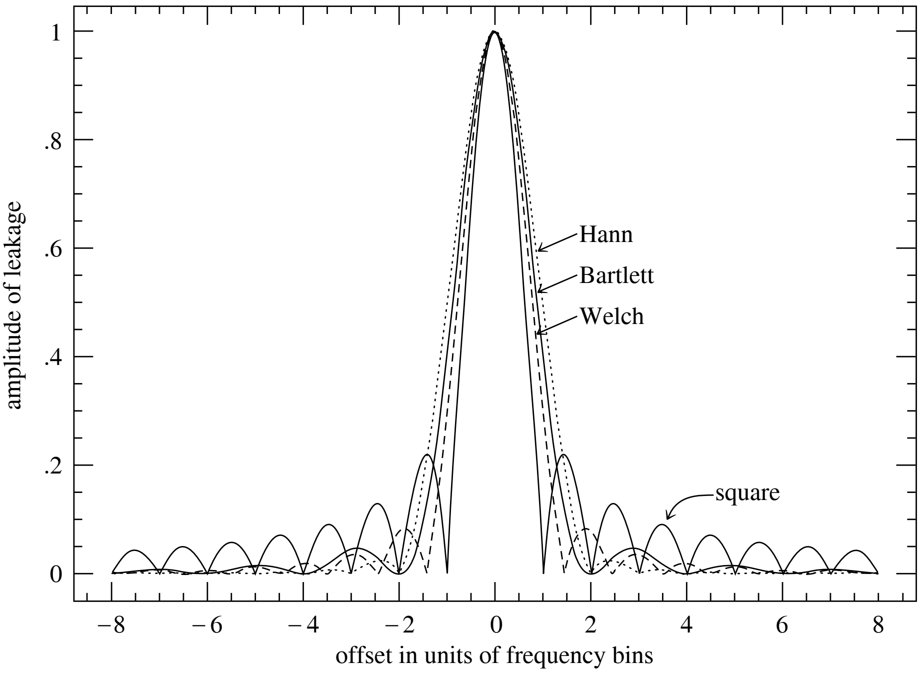


Figure 13.4.2. Leakage functions for the window functions of Figure 13.4.1. A signal whose frequency is actually located at zero offset "leaks" into neighboring bins with the amplitude shown. The purpose of windowing is to reduce the leakage at large offsets, where square (no) windowing has large sidelobes. Offset can have a fractional value, since the actual signal frequency can be located between two frequency bins of the FFT.

PSD estimate at $M + 1$ frequency values from 0 to 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 Kth 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 [3]). This is, however, significantly better than the reduction of about $K/2$ that 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. Note that `spctrm` returns the power at M, not $M + 1$, frequencies, omitting the component $P(f_c)$ at the Nyquist frequency. It would also be straightforward to include that component.

```
#include <math.h>
#include <stdio.h>
#include "nrutil.h"
#define WINDOW(j,a,b) (1.0-fabs((((j)-1)-(a))*(b))) /* Bartlett */
/* #define WINDOW(j,a,b) 1.0 */ /* Square */
/* #define WINDOW(j,a,b) (1.0-SQR((((j)-1)-(a))*(b))) */ /* Welch */

void spctrm(FILE *fp, float p[], int m, int k, int 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.
{
    void four1(float data[], unsigned long nn, int isign);
    int mm,m44,m43,m4,kk,joffn,joff,j2,j;
    float w,facp,facm,*w1,*w2,sumw=0.0,den=0.0;

    mm=m+m;                                     Useful factors.
    m43=(m4=mm+mm)+3;
    m44=m43+1;
    w1=vector(1,m4);
    w2=vector(1,m);
    facm=m;
    facp=1.0/m;
    for (j=1;j<=mm;j++) sumw += SQR(WINDOW(j,facm,facp));
    Accumulate the squared sum of the weights.
    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];
            } else {
                for (j=joff+2;j<=m4;j+=2)
                    fscanf(fp,"%f",&w1[j]);
            }
        }
        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;
        }
        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]));
        }
        den += sumw;
    }
    den *= m4;                                  Correct normalization.
    for (j=1;j<=m;j++) p[j] /= den;             Normalize the output.
    free_vector(w2,1,m);
    free_vector(w1,1,m4);
}
```

CITED REFERENCES AND FURTHER READING:

Oppenheim, A.V., and Schafer, R.W. 1989, *Discrete-Time Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Harris, F.J. 1978, *Proceedings of the IEEE*, vol. 66, pp. 51–83. [2]

Childers, D.G. (ed.) 1978, *Modern Spectrum Analysis* (New York: IEEE Press), paper by P.D. Welch. [3]

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, NJ: Prentice-Hall).

13.5 Digital Filtering in the Time Domain

Suppose that you have a signal that you want to filter digitally. For example, perhaps you want to apply *high-pass* or *low-pass* filtering, to eliminate noise at low or high frequencies respectively; or perhaps the interesting part of your signal lies only in a certain frequency band, so that you need a *bandpass* filter. Or, if your measurements are contaminated by 60 Hz power-line interference, you may need a *notch filter* to remove only a narrow band around that frequency. This section speaks particularly about the case in which you have chosen to do such filtering in the time domain.

Before continuing, we hope you will reconsider this choice. Remember how convenient it is to filter in the Fourier domain. You just take your whole data record, FFT it, multiply the FFT output by a filter function $\mathcal{H}(f)$, and then do an inverse FFT to get back a filtered data set in time domain. Here is some additional background on the Fourier technique that you will want to take into account.

- Remember that you must define your filter function $\mathcal{H}(f)$ for both positive and negative frequencies, and that the magnitude of the frequency extremes is always the Nyquist frequency $1/(2\Delta)$, where Δ is the sampling interval. The magnitude of the smallest nonzero frequencies in the FFT is $\pm 1/(N\Delta)$, where N is the number of (complex) points in the FFT. The positive and negative frequencies to which this filter are applied are arranged in wrap-around order.
- If the measured data are real, and you want the filtered output also to be real, then your arbitrary filter function should obey $\mathcal{H}(-f) = \mathcal{H}(f)^*$. You can arrange this most easily by picking an $\mathcal{H}$ that is real and even in f.

- If your chosen $\mathcal{H}(f)$ has sharp vertical edges in it, then the *impulse response* of your filter (the output arising from a short impulse as input) will have damped "ringing" at frequencies corresponding to these edges. There is nothing wrong with this, but if you don't like it, then pick a smoother $\mathcal{H}(f)$. To get a first-hand look at the impulse response of your filter, just take the inverse FFT of your $\mathcal{H}(f)$. If you smooth all edges of the filter function over some number k of points, then the impulse response function of your filter will have a span on the order of a fraction $1/k$ of the whole data record.
- If your data set is too long to FFT all at once, then break it up into segments of any convenient size, as long as they are much longer than the impulse response function of the filter. Use zero-padding, if necessary.
- You should probably remove any trend from the data, by subtracting from it a straight line through the first and last points (i.e., make the first and last points equal to zero). If you are segmenting the data, then you can pick overlapping segments and use only the middle section of each, comfortably distant from edge effects.
- A digital filter is said to be *causal* or *physically realizable* if its output for a particular time-step depends only on inputs at that particular time-step or earlier. It is said to be *acausal* if its output can depend on both earlier and later inputs. Filtering in the Fourier domain is, in general, acausal, since the data are processed "in a batch," without regard to time ordering. Don't let this bother you! Acausal filters can generally give superior performance (e.g., less dispersion of phases, sharper edges, less asymmetric impulse response functions). People use causal filters not because they are better, but because some situations just don't allow access to out-of-time-order data. Time domain filters can, in principle, be either causal or acausal, but they are most often used in applications where physical realizability is a constraint. For this reason we will restrict ourselves to the causal case in what follows.

If you are still favoring time-domain filtering after all we have said, it is probably because you have a real-time application, for which you must process a continuous data stream and wish to output filtered values at the same rate as you receive raw data. Otherwise, it may be that the quantity of data to be processed is so large that you can afford only a very small number of floating operations on each data point and cannot afford even a modest-sized FFT (with a number of floating operations per data point several times the logarithm of the number of points in the data set or segment).

Linear Filters

The most general linear filter takes a sequence x_k of input points and produces a sequence y_n of output points by the formula

$$y_n = \sum_{k=0}^{M} c_k \; x_{n-k} + \sum_{j=1}^{N} d_j \; y_{n-j} \tag{13.5.1}$$

Here the $M+1$ coefficients c_k and the N coefficients d_j are fixed and define the filter response. The filter (13.5.1) produces each new output value from the current and M previous input values, and from its own N previous output values. If $N = 0$, so that there is no second sum in (13.5.1), then the filter is called *nonrecursive* or *finite impulse response (FIR)*. If $N \neq 0$, then it is called *recursive* or *infinite impulse response (IIR)*. (The term "IIR" connotes only that such filters are *capable* of having infinitely long impulse responses, not that their impulse response is necessarily long in a particular application. Typically the response of an IIR filter will drop off exponentially at late times, rapidly becoming negligible.)

The relation between the c_k's and d_j's and the filter response function $\mathcal{H}(f)$ is

$$\mathcal{H}(f) = \frac{\sum\limits_{k=0}^{M} c_k e^{-2\pi i k (f\Delta)}}{1 - \sum\limits_{j=1}^{N} d_j e^{-2\pi i j (f\Delta)}} \tag{13.5.2}$$

where Δ is, as usual, the sampling interval. The Nyquist interval corresponds to $f\Delta$ between $-1/2$ and $1/2$. For FIR filters the denominator of (13.5.2) is just unity.

Equation (13.5.2) tells how to determine $\mathcal{H}(f)$ from the c's and d's. To design a filter, though, we need a way of doing the inverse, getting a suitable set of c's and d's — as small a set as possible, to minimize the computational burden — from a desired $\mathcal{H}(f)$. Entire books are devoted to this issue. Like many other "inverse problems," it has no all-purpose solution. One clearly has to make compromises, since $\mathcal{H}(f)$ is a full continuous function, while the short list of c's and d's represents only a few adjustable parameters. The subject of digital filter design concerns itself with the various ways of making these compromises. We cannot hope to give any sort of complete treatment of the subject. We can, however, sketch a couple of basic techniques to get you started. For further details, you will have to consult some specialized books (see references).

FIR (Nonrecursive) Filters

When the denominator in (13.5.2) is unity, the right-hand side is just a discrete Fourier transform. The transform is easily invertible, giving the desired small number of c_k coefficients in terms of the same small number of values of $\mathcal{H}(f_i)$ at some discrete frequencies f_i. This fact, however, is not very useful. The reason is that, for values of c_k computed in this way, $\mathcal{H}(f)$ will tend to oscillate wildly in between the discrete frequencies where it is pinned down to specific values.

A better strategy, and one which is the basis of several formal methods in the literature, is this: Start by pretending that you are willing to have a relatively large number of filter coefficients, that is, a relatively large value of M. Then $\mathcal{H}(f)$ can be fixed to desired values on a relatively fine mesh, and the M coefficients $c_k,\ k = 0,\dots,M-1$ can be found by an FFT. Next, truncate (set to zero) most of the c_k's, leaving nonzero only the first, say, K, $(c_0, c_1, \dots, c_{K-1})$ and last $K-1$, $(c_{M-K+1},\dots,c_{M-1})$. The last few c_k's are filter coefficients at *negative lag*, because of the wrap-around property of the FFT. But we don't want coefficients at negative lag. Therefore we cyclically shift the array of c_k's, to bring everything to positive lag. (This corresponds to introducing a time-delay into the filter.) Do this by copying the c_k's into a new array of length M in the following order:

$$(c_{M-K+1},\dots,c_{M-1},\ c_0,\ c_1,\dots,c_{K-1},\ 0,\ 0,\dots,0) \qquad (13.5.3)$$

To see if your truncation is acceptable, take the FFT of the array (13.5.3), giving an approximation to your original $\mathcal{H}(f)$. You will generally want to compare the *modulus* $|\mathcal{H}(f)|$ to your original function, since the time-delay will have introduced complex phases into the filter response.

If the new filter function is acceptable, then you are done and have a set of $2K-1$ filter coefficients. If it is not acceptable, then you can either (i) increase K and try again, or (ii) do something fancier to improve the acceptability for the same K. An example of something fancier is to modify the magnitudes (but not the phases) of the unacceptable $\mathcal{H}(f)$ to bring it more in line with your ideal, and then to FFT to get new c_k's. Once again set to zero all but the first $2K-1$ values of these (no need to cyclically shift since you have preserved the time-delaying phases), then inverse transform to get a new $\mathcal{H}(f)$, which will often be more acceptable. You can iterate this procedure. Note, however, that the procedure will not converge if your requirements for acceptability are more stringent than your $2K-1$ coefficients can handle.

The key idea, in other words, is to iterate between the space of coefficients and the space of functions $\mathcal{H}(f)$, until a Fourier conjugate pair that satisfies the imposed constraints *in both spaces* is found. A more formal technique for this kind of iteration is the *Remes Exchange Algorithm* which produces the best Chebyshev approximation to a given desired frequency response with a fixed number of filter coefficients (cf. §5.13).

IIR (Recursive) Filters

Recursive filters, whose output at a given time depends both on the current and previous inputs and on previous outputs, can generally have performance that is superior to nonrecursive filters with the same total number of coefficients (or same number of floating operations per input point). The reason is fairly clear by inspection of (13.5.2): A nonrecursive filter has a frequency response that is a polynomial in the variable $1/z$, where

$$z \equiv e^{2\pi i(f\Delta)} \tag{13.5.4}$$

By contrast, a recursive filter's frequency response is a *rational function* in $1/z$. The class of rational functions is especially good at fitting functions with sharp edges or narrow features, and most desired filter functions are in this category.

Nonrecursive filters are always stable. If you turn off the sequence of incoming x_i's, then after no more than M steps the sequence of y_j's produced by (13.5.1) will also turn off. Recursive filters, feeding as they do on their own output, are not necessarily stable. If the coefficients d_j are badly chosen, a recursive filter can have exponentially growing, so-called *homogeneous*, modes, which become huge even after the input sequence has been turned off. This is not good. The problem of designing recursive filters, therefore, is not just an inverse problem; it is an inverse problem with an additional stability constraint.

How do you tell if the filter (13.5.1) is stable for a given set of c_k and d_j coefficients? Stability depends only on the d_j's. The filter is stable if and only if all N complex roots of the *characteristic polynomial* equation

$$z^N - \sum_{j=1}^{N} d_j z^{N-j} = 0 \tag{13.5.5}$$

are inside the unit circle, i.e., satisfy

$$|z| \le 1 \tag{13.5.6}$$

The various methods for constructing stable recursive filters again form a subject area for which you will need more specialized books. One very useful technique, however, is the *bilinear transformation method*. For this topic we define a new variable w that reparametrizes the frequency f,

$$w \equiv \tan[\pi(f\Delta)] = i\left(\frac{1-e^{2\pi i(f\Delta)}}{1+e^{2\pi i(f\Delta)}}\right) = i\left(\frac{1-z}{1+z}\right) \tag{13.5.7}$$

Don't be fooled by the i's in (13.5.7). This equation maps real frequencies f into real values of w. In fact, it maps the Nyquist interval $-\frac{1}{2} < f\Delta < \frac{1}{2}$ onto the real w axis $-\infty < w < +\infty$. The inverse equation to (13.5.7) is

$$z = e^{2\pi i(f\Delta)} = \frac{1+iw}{1-iw} \tag{13.5.8}$$

In reparametrizing f, w also reparametrizes z, of course. Therefore, the condition for stability (13.5.5)–(13.5.6) can be rephrased in terms of w: If the filter response $\mathcal{H}(f)$ is written as a function of w, then the filter is stable if and only if the poles of the filter function (zeros of its denominator) are all in the upper half complex plane,

$$\text{Im}(w) \ge 0 \tag{13.5.9}$$

The idea of the bilinear transformation method is that instead of specifying your desired $\mathcal{H}(f)$, you specify only its desired modulus square, $|\mathcal{H}(f)|^2 = \mathcal{H}(f)\mathcal{H}(f)^* = \mathcal{H}(f)\mathcal{H}(-f)$. Pick this to be approximated by some rational function in w^2. Then find all the poles of this function in the w complex plane. Every pole in the lower half-plane will have a corresponding pole in the upper half-plane, by symmetry. The idea is to form a product only of the factors with good poles, ones in the upper half-plane. This product is your *stably realizable* $\mathcal{H}(f)$. Now substitute equation (13.5.7) to write the function as a rational function in z, and compare with equation (13.5.2) to read off the c's and d's.

The procedure becomes clearer when we go through an example. Suppose we want to design a simple bandpass filter, whose lower cutoff frequency corresponds to a value $w = a$, and whose upper cutoff frequency corresponds to a value $w = b$, with a and b both positive numbers. A simple rational function that accomplishes this is

$$|\mathcal{H}(f)|^2 = \left(\frac{w^2}{w^2 + a^2}\right)\left(\frac{b^2}{w^2 + b^2}\right) \tag{13.5.10}$$

This function does not have a very sharp cutoff, but it is illustrative of the more general case. To obtain sharper edges, one could take the function (13.5.10) to some positive integer power, or, equivalently, run the data sequentially through some number of copies of the filter that we will obtain from (13.5.10).

The poles of (13.5.10) are evidently at $w = \pm ia$ and $w = \pm ib$. Therefore the stably realizable $\mathcal{H}(f)$ is

$$\mathcal{H}(f) = \left(\frac{w}{w - ia}\right)\left(\frac{ib}{w - ib}\right) = \frac{\left(\frac{1-z}{1+z}\right) b}{\left[\left(\frac{1-z}{1+z}\right) - a\right]\left[\left(\frac{1-z}{1+z}\right) - b\right]} \tag{13.5.11}$$

We put the i in the numerator of the second factor in order to end up with real-valued coefficients. If we multiply out all the denominators, (13.5.11) can be rewritten in the form

$$\mathcal{H}(f) = \frac{-\frac{b}{(1+a)(1+b)} + \frac{b}{(1+a)(1+b)} z^{-2}}{1 - \frac{(1+a)(1-b)+(1-a)(1+b)}{(1+a)(1+b)} z^{-1} + \frac{(1-a)(1-b)}{(1+a)(1+b)} z^{-2}} \tag{13.5.12}$$

from which one reads off the filter coefficients for equation (13.5.1),

$$\begin{aligned}
c_0 &= -\frac{b}{(1+a)(1+b)} \\
c_1 &= 0 \\
c_2 &= \frac{b}{(1+a)(1+b)} \\
d_1 &= \frac{(1+a)(1-b) + (1-a)(1+b)}{(1+a)(1+b)} \\
d_2 &= -\frac{(1-a)(1-b)}{(1+a)(1+b)}
\end{aligned} \tag{13.5.13}$$

This completes the design of the bandpass filter.

Sometimes you can figure out how to construct directly a rational function in w for $\mathcal{H}(f)$, rather than having to start with its modulus square. The function that you construct has to have its poles only in the upper half-plane, for stability. It should also have the property of going into its own complex conjugate if you substitute $-w$ for w, so that the filter coefficients will be real.

For example, here is a function for a notch filter, designed to remove only a narrow frequency band around some fiducial frequency $w = w_0$, where w_0 is a positive number,

$$\begin{aligned}
\mathcal{H}(f) &= \left(\frac{w - w_0}{w - w_0 - i\epsilon w_0}\right)\left(\frac{w + w_0}{w + w_0 - i\epsilon w_0}\right) \\
&= \frac{w^2 - w_0^2}{(w - i\epsilon w_0)^2 - w_0^2}
\end{aligned} \tag{13.5.14}$$

In (13.5.14) the parameter ϵ is a small positive number that is the desired width of the notch, as a

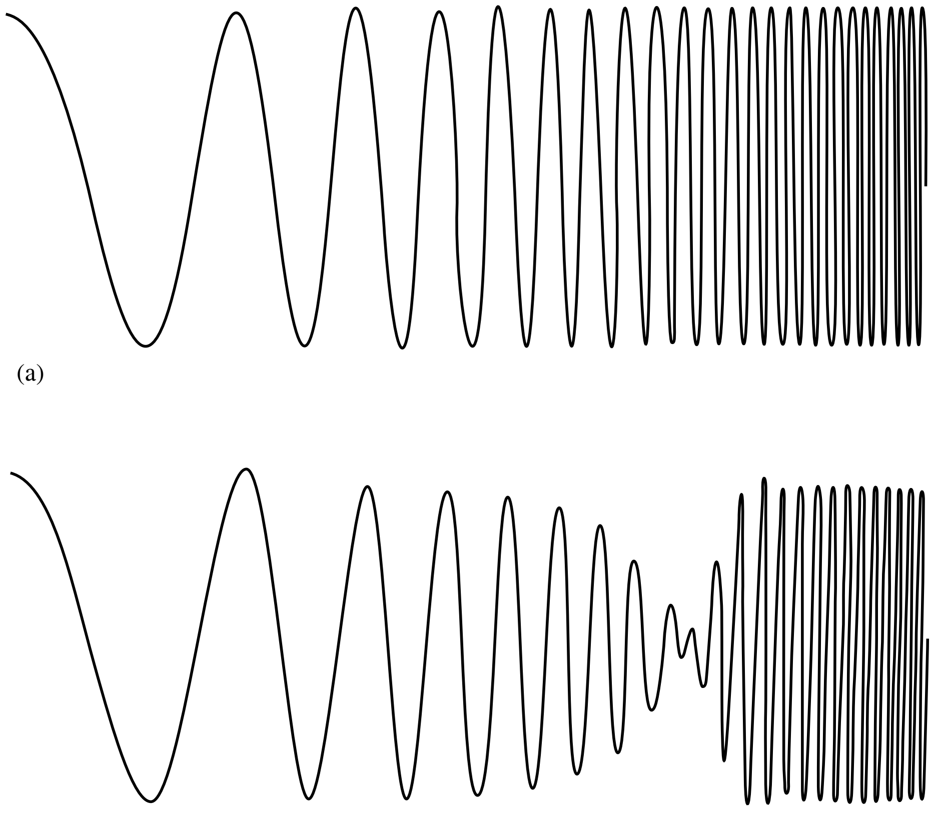


Figure 13.5.1. (a) A "chirp," or signal whose frequency increases continuously with time. (b) Same signal after it has passed through the notch filter (13.5.15). The parameter ϵ is here 0.2.

fraction of w_0. Going through the arithmetic of substituting z for w gives the filter coefficients

$$\begin{aligned}
c_0 &= \frac{1+w_0^2}{(1+\epsilon w_0)^2+w_0^2} \\
c_1 &= -2\frac{1-w_0^2}{(1+\epsilon w_0)^2+w_0^2} \\
c_2 &= \frac{1+w_0^2}{(1+\epsilon w_0)^2+w_0^2} \\
d_1 &= 2\frac{1-\epsilon^2 w_0^2-w_0^2}{(1+\epsilon w_0)^2+w_0^2} \\
d_2 &= -\frac{(1-\epsilon w_0)^2+w_0^2}{(1+\epsilon w_0)^2+w_0^2}
\end{aligned} \tag{13.5.15}$$

Figure 13.5.1 shows the results of using a filter of the form (13.5.15) on a "chirp" input signal, one that glides upwards in frequency, crossing the notch frequency along the way.

While the bilinear transformation may seem very general, its applications are limited by some features of the resulting filters. The method is good at getting the general shape of the desired filter, and good where "flatness" is a desired goal. However, the nonlinear mapping between w and f makes it difficult to design to a desired shape for a cutoff, and may move cutoff frequencies (defined by a certain number of dB) from their desired places. Consequently, practitioners of the art of digital filter design reserve the bilinear transformation for specific situations, and arm themselves with a variety of other tricks. We suggest that you do likewise, as your projects demand.

CITED REFERENCES AND FURTHER READING:

Hamming, R.W. 1983, *Digital Filters*, 2nd ed. (Englewood Cliffs, NJ: Prentice-Hall).

Antoniou, A. 1979, *Digital Filters: Analysis and Design* (New York: McGraw-Hill).

Parks, T.W., and Burrus, C.S. 1987, *Digital Filter Design* (New York: Wiley).

Oppenheim, A.V., and Schafer, R.W. 1989, *Discrete-Time Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall).

Rice, J.R. 1964, *The Approximation of Functions* (Reading, MA: Addison-Wesley); also 1969, *op. cit.*, Vol. 2.

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

13.6 Linear Prediction and Linear Predictive Coding

We begin with a very general formulation that will allow us to make connections to various special cases. Let $\{y'_\alpha\}$ be a set of measured values for some underlying set of true values of a quantity y, denoted $\{y_\alpha\}$, related to these true values by the addition of random noise,

$$y'_\alpha = y_\alpha + n_\alpha \tag{13.6.1}$$

(compare equation 13.3.2, with a somewhat different notation). Our use of a Greek subscript to index the members of the set is meant to indicate that the data points are not necessarily equally spaced along a line, or even ordered: they might be "random" points in three-dimensional space, for example. Now, suppose we want to construct the "best" estimate of the true value of some particular point $y_\star$ as a linear combination of the known, noisy, values. Writing

$$y_\star = \sum_\alpha d_{\star\alpha} y'_\alpha + x_\star \tag{13.6.2}$$

we want to find coefficients $d_{\star\alpha}$ that minimize, in some way, the *discrepancy* $x_\star$. The coefficients $d_{\star\alpha}$ have a "star" subscript to indicate that they depend on the choice of point $y_\star$. Later, we might want to let $y_\star$ be one of the existing y_α's. In that case, our problem becomes one of optimal filtering or estimation, closely related to the discussion in §13.3. On the other hand, we might want $y_\star$ to be a completely new point. In that case, our problem will be one of *linear prediction*.

A natural way to minimize the discrepancy $x_\star$ is in the statistical mean square sense. If angle brackets denote statistical averages, then we seek $d_{\star\alpha}$'s that minimize

$$\begin{aligned}\left\langle x_\star^2 \right\rangle &= \left\langle \left[\sum_\alpha d_{\star\alpha}(y_\alpha + n_\alpha) - y_\star \right]^2 \right\rangle \\ &= \sum_{\alpha\beta} (\langle y_\alpha y_\beta \rangle + \langle n_\alpha n_\beta \rangle) d_{\star\alpha} d_{\star\beta} - 2 \sum_\alpha \langle y_\star y_\alpha \rangle d_{\star\alpha} + \left\langle y_\star^2 \right\rangle\end{aligned} \tag{13.6.3}$$

Here we have used the fact that noise is uncorrelated with signal, e.g., $\langle n_\alpha y_\beta \rangle = 0$. The quantities $\langle y_\alpha y_\beta \rangle$ and $\langle y_\star y_\alpha \rangle$ describe the autocorrelation structure of the underlying data. We have already seen an analogous expression, (13.2.2), for the case of equally spaced data points on a line; we will meet correlation several times again in its statistical sense in Chapters 14 and 15. The quantities $\langle n_\alpha n_\beta \rangle$ describe the autocorrelation properties of the noise. Often, for point-to-point uncorrelated noise, we have $\langle n_\alpha n_\beta \rangle = \left\langle n_\alpha^2 \right\rangle \delta_{\alpha\beta}$. It is convenient to think of the various correlation quantities as comprising matrices and vectors,

$$\phi_{\alpha\beta} \equiv \langle y_\alpha y_\beta \rangle \qquad \phi_{\star\alpha} \equiv \langle y_\star y_\alpha \rangle \qquad \eta_{\alpha\beta} \equiv \langle n_\alpha n_\beta \rangle \;\text{ or }\; \left\langle n_\alpha^2 \right\rangle \delta_{\alpha\beta} \tag{13.6.4}$$

Setting the derivative of equation (13.6.3) with respect to the $d_{\star\alpha}$'s equal to zero, one readily obtains the set of linear equations,

$$\sum_\beta \left[\phi_{\alpha\beta} + \eta_{\alpha\beta}\right] d_{\star\beta} = \phi_{\star\alpha} \tag{13.6.5}$$

If we write the solution as a matrix inverse, then the estimation equation (13.6.2) becomes, omitting the minimized discrepancy $x_\star$,

$$y_\star \approx \sum_{\alpha\beta} \phi_{\star\alpha} \left[\phi_{\mu\nu} + \eta_{\mu\nu}\right]^{-1}_{\alpha\beta} y'_\beta \tag{13.6.6}$$

From equations (13.6.3) and (13.6.5) one can also calculate the expected mean square value of the discrepancy at its minimum, denoted $\left\langle x_\star^2 \right\rangle_0$,

$$\left\langle x_\star^2 \right\rangle_0 = \left\langle y_\star^2 \right\rangle - \sum_\beta d_{\star\beta}\phi_{\star\beta} = \left\langle y_\star^2 \right\rangle - \sum_{\alpha\beta} \phi_{\star\alpha} \left[\phi_{\mu\nu} + \eta_{\mu\nu}\right]^{-1}_{\alpha\beta} \phi_{\star\beta} \tag{13.6.7}$$

A final general result tells how much the mean square discrepancy $\left\langle x_\star^2 \right\rangle$ is increased if we use the estimation equation (13.6.2) not with the best values $d_{\star\beta}$, but with some other values $\widehat{d}_{\star\beta}$. The above equations then imply

$$\left\langle x_\star^2 \right\rangle = \left\langle x_\star^2 \right\rangle_0 + \sum_{\alpha\beta} (\widehat{d}_{\star\alpha} - d_{\star\alpha}) \left[\phi_{\alpha\beta} + \eta_{\alpha\beta}\right] (\widehat{d}_{\star\beta} - d_{\star\beta}) \tag{13.6.8}$$

Since the second term is a pure quadratic form, we see that the increase in the discrepancy is only second order in any error made in estimating the $d_{\star\beta}$'s.

Connection to Optimal Filtering

If we change "star" to a Greek index, say γ, then the above formulas describe optimal filtering, generalizing the discussion of §13.3. One sees, for example, that if the noise amplitudes n_α go to zero, so likewise do the noise autocorrelations $\eta_{\alpha\beta}$, and, canceling a matrix times its inverse, equation (13.6.6) simply becomes $y_\gamma = y'_\gamma$. Another special case occurs if the matrices $\phi_{\alpha\beta}$ and $\eta_{\alpha\beta}$ are diagonal. In that case, equation (13.6.6) becomes

$$y_\gamma = \frac{\phi_{\gamma\gamma}}{\phi_{\gamma\gamma} + \eta_{\gamma\gamma}} y'_\gamma \tag{13.6.9}$$

which is readily recognizable as equation (13.3.6) with $S^2 \to \phi_{\gamma\gamma}$, $N^2 \to \eta_{\gamma\gamma}$. What is going on is this: For the case of equally spaced data points, and in the Fourier domain, autocorrelations become simply squares of Fourier amplitudes (Wiener-Khinchin theorem, equation 12.0.12), and the optimal filter can be constructed algebraically, as equation (13.6.9), without inverting any matrix.

More generally, in the time domain, or any other domain, an optimal filter (one that minimizes the square of the discrepancy from the underlying true value in the presence of measurement noise) can be constructed by estimating the autocorrelation matrices $\phi_{\alpha\beta}$ and $\eta_{\alpha\beta}$, and applying equation (13.6.6) with $\star \to \gamma$. (Equation 13.6.8 is in fact the basis for the §13.3's statement that even crude optimal filtering can be quite effective.)

Linear Prediction

Classical *linear prediction* specializes to the case where the data points y_β are equally spaced along a line, y_i, $i = 1, 2, \ldots, N$, and we want to use M consecutive values of y_i to predict an $M + 1$st. Stationarity is assumed. That is, the autocorrelation $\langle y_j y_k \rangle$ is assumed to depend only on the difference $|j - k|$, and not on j or k individually, so that the autocorrelation ϕ has only a single index,

$$\phi_j \equiv \langle y_i y_{i+j} \rangle \approx \frac{1}{N-j} \sum_{i=1}^{N-j} y_i y_{i+j} \qquad (13.6.10)$$

Here, the approximate equality shows one way to use the actual data set values to estimate the autocorrelation components. (In fact, there is a better way to make these estimates; see below.) In the situation described, the estimation equation (13.6.2) is

$$y_n = \sum_{j=1}^{M} d_j y_{n-j} + x_n \qquad (13.6.11)$$

(compare equation 13.5.1) and equation (13.6.5) becomes the set of M equations for the M unknown d_j's, now called the *linear prediction (LP) coefficients*,

$$\sum_{j=1}^{M} \phi_{|j-k|}\, d_j = \phi_k \qquad (k = 1, \ldots, M) \qquad (13.6.12)$$

Notice that while noise is not explicitly included in the equations, it is properly accounted for, *if* it is point-to-point uncorrelated: ϕ_0, as estimated by equation (13.6.10) using *measured* values y'_i, actually estimates the diagonal part of $\phi_{\alpha\alpha} + \eta_{\alpha\alpha}$, above. The mean square discrepancy $\langle x_n^2 \rangle$ is estimated by equation (13.6.7) as

$$\langle x_n^2 \rangle = \phi_0 - \phi_1 d_1 - \phi_2 d_2 - \cdots - \phi_M d_M \qquad (13.6.13)$$

To use linear prediction, we first compute the d_j's, using equations (13.6.10) and (13.6.12). We then calculate equation (13.6.13) or, more concretely, apply (13.6.11) to the known record to get an idea of how large are the discrepancies x_i. If the discrepancies are small, then we can continue applying (13.6.11) right on into the future, imagining the unknown "future" discrepancies x_i to be zero. In this application, (13.6.11) is a kind of extrapolation formula. In many situations, this extrapolation turns out to be vastly more powerful than any kind of simple polynomial extrapolation. (By the way, you should not confuse the terms "linear prediction" and "linear extrapolation"; the general functional form used by linear prediction is *much* more complex than a straight line, or even a low-order polynomial!)

However, to achieve its full usefulness, linear prediction must be constrained in one additional respect: One must take additional measures to guarantee its *stability*. Equation (13.6.11) is a special case of the general linear filter (13.5.1). The condition that (13.6.11) be stable as a linear predictor is precisely that given in equations (13.5.5) and (13.5.6), namely that the characteristic polynomial

$$z^N - \sum_{j=1}^{N} d_j z^{N-j} = 0 \qquad (13.6.14)$$

have all N of its roots inside the unit circle,

$$|z| \le 1 \qquad (13.6.15)$$

There is no guarantee that the coefficients produced by equation (13.6.12) will have this property. If the data contain many oscillations without any particular trend towards increasing or decreasing amplitude, then the complex roots of (13.6.14) will generally all be rather close to the unit circle. The finite length of the data set will cause some of these roots to be inside the unit circle, others outside. In some applications, where the resulting instabilities are slowly growing and the linear prediction is not pushed too far, it is best to use the "unmassaged" LP coefficients that come directly out of (13.6.12). For example, one might be extrapolating to fill a short gap in a data set; then one might extrapolate both forwards across the gap and backwards from the data beyond the gap. If the two extrapolations agree tolerably well, then instability is not a problem.

When instability *is* a problem, you have to "massage" the LP coefficients. You do this by (i) solving (numerically) equation (13.6.14) for its N complex roots; (ii) moving the roots to where you think they ought to be inside or on the unit circle; (iii) reconstituting the now-modified LP coefficients. You may think that step (ii) sounds a little vague. It is. There is no "best" procedure. If you think that your signal is truly a sum of undamped sine and cosine waves (perhaps with incommensurate periods), then you will want simply to move each root z_i onto the unit circle,

$$z_i \;\rightarrow\; z_i / |z_i| \qquad (13.6.16)$$

In other circumstances it may seem appropriate to reflect a bad root across the unit circle

$$z_i \;\rightarrow\; 1/{z_i}^* \qquad (13.6.17)$$

This alternative has the property that it preserves the amplitude of the output of (13.6.11) when it is driven by a sinusoidal set of x_i's. It assumes that (13.6.12) has correctly identified the spectral width of a resonance, but only slipped up on identifying its time sense so that signals that should be damped as time proceeds end up growing in amplitude. The choice between (13.6.16) and (13.6.17) sometimes might as well be based on voodoo. We prefer (13.6.17).

Also magical is the choice of M, the number of LP coefficients to use. You should choose M to be as small as works for you, that is, you should choose it by experimenting with your data. Try $M = 5, 10, 20, 40$. If you need larger M's than this, be aware that the procedure of "massaging" all those complex roots is quite sensitive to roundoff error. Use double precision.

Linear prediction is especially successful at extrapolating signals that are smooth and oscillatory, though not necessarily periodic. In such cases, linear prediction often extrapolates accurately through *many cycles* of the signal. By contrast, polynomial extrapolation in general becomes seriously inaccurate after at most a cycle or two. A prototypical example of a signal that can successfully be linearly predicted is the height of ocean tides, for which the fundamental 12-hour period is modulated in phase and amplitude over the course of the month and year, and for which local hydrodynamic effects may make even one cycle of the curve look rather different in shape from a sine wave.

We already remarked that equation (13.6.10) is not necessarily the best way to estimate the covariances ϕ_k from the data set. In fact, results obtained from linear prediction are remarkably sensitive to exactly how the ϕ_k's are estimated. One particularly good method is due to Burg [1], and involves a recursive procedure for increasing the order M by one unit at a time, at each stage re-estimating the coefficients d_j, $j = 1, \ldots, M$ so as to minimize the residual in equation (13.6.13). Although further discussion of the Burg method is beyond our scope here, the method is implemented in the following routine [1,2] for estimating the LP coefficients d_j of a data set.

```
#include <math.h>
#include "nrutil.h"

void memcof(float data[], int n, int m, float *xms, float d[])
```
Given a real vector of `data[1..n]`, and given `m`, this routine returns `m` linear prediction coefficients as `d[1..m]`, and returns the mean square discrepancy as `xms`.
```
{
    int k,j,i;
    float p=0.0,*wk1,*wk2,*wkm;

    wk1=vector(1,n);
    wk2=vector(1,n);
    wkm=vector(1,m);
    for (j=1;j<=n;j++) p += SQR(data[j]);
    *xms=p/n;
    wk1[1]=data[1];
    wk2[n-1]=data[n];
    for (j=2;j<=n-1;j++) {
        wk1[j]=data[j];
        wk2[j-1]=data[j];
    }
    for (k=1;k<=m;k++) {
        float num=0.0,denom=0.0;
        for (j=1;j<=(n-k);j++) {
            num += wk1[j]*wk2[j];
            denom += SQR(wk1[j])+SQR(wk2[j]);
        }
        d[k]=2.0*num/denom;
        *xms *= (1.0-SQR(d[k]));
        for (i=1;i<=(k-1);i++)
            d[i]=wkm[i]-d[k]*wkm[k-i];
            The algorithm is recursive, building up the answer for larger and larger values of m
            until the desired value is reached. At this point in the algorithm, one could return
            the vector d and scalar xms for a set of LP coefficients with k (rather than m)
            terms.
        if (k == m) {
            free_vector(wkm,1,m);
            free_vector(wk2,1,n);
            free_vector(wk1,1,n);
            return;
        }
        for (i=1;i<=k;i++) wkm[i]=d[i];
        for (j=1;j<=(n-k-1);j++) {
            wk1[j] -= wkm[k]*wk2[j];
            wk2[j]=wk2[j+1]-wkm[k]*wk1[j+1];
        }
    }
    nrerror("never get here in memcof.");
}
```

Here are procedures for rendering the LP coefficients stable (if you choose to do so), and for extrapolating a data set by linear prediction, using the original or massaged LP coefficients. The routine zroots (§9.5) is used to find all complex roots of a polynomial.

```
#include <math.h>
#include "complex.h"
#define NMAX 100                            Largest expected value of m.
#define ZERO Complex(0.0,0.0)
#define ONE Complex(1.0,0.0)

void fixrts(float d[], int m)
Given the LP coefficients d[1..m], this routine finds all roots of the characteristic polynomial
(13.6.14), reflects any roots that are outside the unit circle back inside, and then returns a
modified set of coefficients d[1..m].
{
    void zroots(fcomplex a[], int m, fcomplex roots[], int polish);
    int i,j,polish;
    fcomplex a[NMAX],roots[NMAX];

    a[m]=ONE;
    for (j=m-1;j>=0;j--)                    Set up complex coefficients for polynomial root
        a[j]=Complex(-d[m-j],0.0);              finder.
    polish=1;
    zroots(a,m,roots,polish);               Find all the roots.
    for (j=1;j<=m;j++)                      Look for a...
        if (Cabs(roots[j]) > 1.0)           root outside the unit circle,
            roots[j]=Cdiv(ONE,Conjg(roots[j]));   and reflect it back inside.
    a[0]=Csub(ZERO,roots[1]);               Now reconstruct the polynomial coefficients,
    a[1]=ONE;
    for (j=2;j<=m;j++) {                    by looping over the roots
        a[j]=ONE;
        for (i=j;i>=2;i--)                  and synthetically multiplying.
            a[i-1]=Csub(a[i-2],Cmul(roots[j],a[i-1]));
        a[0]=Csub(ZERO,Cmul(roots[j],a[0]));
    }
    for (j=0;j<=m-1;j++)                    The polynomial coefficients are guaranteed to be
        d[m-j] = -a[j].r;                       real, so we need only return the real part as
}                                               new LP coefficients.
```

```
#include "nrutil.h"

void predic(float data[], int ndata, float d[], int m, float future[],
    int nfut)
Given data[1..ndata], and given the data's LP coefficients d[1..m], this routine ap-
plies equation (13.6.11) to predict the next nfut data points, which it returns in the array
future[1..nfut]. Note that the routine references only the last m values of data, as initial
values for the prediction.
{
    int k,j;
    float sum,discrp,*reg;

    reg=vector(1,m);
    for (j=1;j<=m;j++) reg[j]=data[ndata+1-j];
    for (j=1;j<=nfut;j++) {
        discrp=0.0;
        This is where you would put in a known discrepancy if you were reconstructing a
        function by linear predictive coding rather than extrapolating a function by linear pre-
        diction. See text.
        sum=discrp;
        for (k=1;k<=m;k++) sum += d[k]*reg[k];
        for (k=m;k>=2;k--) reg[k]=reg[k-1];       [If you want to implement circular
        future[j]=reg[1]=sum;                         arrays, you can avoid this shift-
    }                                                 ing of coefficients.]
    free_vector(reg,1,m);
}
```

Removing the Bias in Linear Prediction

You might expect that the sum of the d_j's in equation (13.6.11) (or, more generally, in equation 13.6.2) should be 1, so that (e.g.) adding a constant to all the data points y_i yields a prediction that is increased by the same constant. However, the d_j's do not sum to 1 but, in general, to a value slightly less than one. This fact reveals a subtle point, that the estimator of classical linear prediction is not *unbiased*, even though it does minimize the mean square discrepancy. At any place where the measured autocorrelation does not imply a better estimate, the equations of linear prediction tend to predict a value that tends towards zero.

Sometimes, that is just what you want. If the process that generates the y_i's in fact has zero mean, then zero is the best guess absent other information. At other times, however, this behavior is unwarranted. If you have data that show only small variations around a positive value, you don't want linear predictions that droop towards zero.

Often it is a workable approximation to subtract the mean off your data set, perform the linear prediction, and then add the mean back. This procedure contains the germ of the correct solution; but the simple arithmetic mean is not quite the correct constant to subtract. In fact, an unbiased estimator is obtained by subtracting from every data point an autocorrelation-weighted mean defined by [3,4]

$$\overline{y} \equiv \sum_{\beta} [\phi_{\mu\nu} + \eta_{\mu\nu}]^{-1}_{\alpha\beta}\, y_\beta \bigg/ \sum_{\alpha\beta} [\phi_{\mu\nu} + \eta_{\mu\nu}]^{-1}_{\alpha\beta} \tag{13.6.18}$$

With this subtraction, the sum of the LP coefficients should be unity, up to roundoff and differences in how the ϕ_k's are estimated.

Linear Predictive Coding (LPC)

A different, though related, method to which the formalism above can be applied is the "compression" of a sampled signal so that it can be stored more compactly. The original form should be *exactly* recoverable from the compressed version. Obviously, compression can be accomplished only if there is redundancy in the signal. Equation (13.6.11) describes one kind of redundancy: It says that the signal, except for a small discrepancy, is predictable from its previous values and from a small number of LP coefficients. Compression of a signal by the use of (13.6.11) is thus called *linear predictive coding*, or *LPC*.

The basic idea of LPC (in its simplest form) is to record as a compressed file (i) the number of LP coefficients M, (ii) their M values, e.g., as obtained by `memcof`, (iii) the first M data points, and then (iv) for each subsequent data point only its residual discrepancy x_i (equation 13.6.1). When you are creating the compressed file, you find the residual by applying (13.6.1) to the previous M points, subtracting the sum from the actual value of the current point. When you are reconstructing the original file, you add the residual back in, at the point indicated in the routine `predic`.

It may not be obvious why there is any compression at all in this scheme. After all, we are storing one value of residual per data point! Why not just store the original data point? The answer depends on the relative sizes of the numbers involved. The residual is obtained by subtracting two very nearly equal numbers (the data and the linear prediction). Therefore, the discrepancy typically has only a very small number of nonzero bits. These can be stored in a compressed file. How do you do it in a high-level language? Here is one way: Scale your data to have integer values, say between $+1000000$ and -1000000 (supposing that you need six significant figures). Modify equation (13.6.1) by enclosing the sum term in an "integer part of" operator. The discrepancy will now, by definition, be an integer. Experiment with different values of M, to find LP coefficients that make the range of the discrepancy as small as you can. If you can get to within a range of ± 127 (and in our experience this is not at all difficult) then you can write it to a file as a single byte. This is a compression factor of 4, compared to 4-byte integer or floating formats.

Notice that the LP coefficients are computed using the *quantized* data, and that the discrepancy is also quantized, i.e., quantization is done both outside and inside the LPC loop. If you are careful in following this prescription, then, apart from the initial quantization of the data, you will not introduce even a single bit of roundoff error into the compression-reconstruction process: While the evaluation of the sum in (13.6.11) may have roundoff errors, the residual that you store is the value which, when added back to the sum, gives *exactly* the original (quantized) data value. Notice also that you do not need to massage the LP coefficients for stability; by adding the residual back in to each point, you never depart from the original data, so instabilities cannot grow. There is therefore no need for `fixrts`, above.

Look at §20.4 to learn about *Huffman coding*, which will further compress the residuals by taking advantage of the fact that smaller values of discrepancy will occur more often than larger values. A very primitive version of Huffman coding would be this: If most of the discrepancies are in the range ± 127, but an occasional one is outside, then reserve the value 127 to mean "out of range," and then record on the file (immediately following the 127) a full-word value of the out-of-range discrepancy. §20.4 explains how to do much better.

There are many variant procedures that all fall under the rubric of LPC.

- If the spectral character of the data is time-variable, then it is best not to use a single set of LP coefficients for the whole data set, but rather to partition the data into segments, computing and storing different LP coefficients for each segment.
- If the data are really well characterized by their LP coefficients, and you can tolerate some small amount of error, then don't bother storing all of the residuals. Just do linear prediction until you are outside of tolerances, then reinitialize (using M sequential stored residuals) and continue predicting.
- In some applications, most notably speech synthesis, one cares only about the spectral content of the reconstructed signal, not the relative phases. In this case, one need not store any starting values at all, but only the LP coefficients for each segment of the data. The output is reconstructed by driving these coefficients with initial conditions consisting of all zeros except for one nonzero spike. A speech synthesizer chip may have of order 10 LP coefficients, which change perhaps 20 to 50 times per second.
- Some people believe that it is interesting to analyze a signal by LPC, even when the residuals x_i are *not* small. The x_i's are then interpreted as the underlying "input signal" which, when filtered through the all-poles filter defined by the LP coefficients (see §13.7), produces the observed "output signal." LPC reveals simultaneously, it is said, the nature of the filter *and* the particular input that is driving it. We are skeptical of these applications; the literature, however, is full of extravagant claims.

CITED REFERENCES AND FURTHER READING:

Childers, D.G. (ed.) 1978, *Modern Spectrum Analysis* (New York: IEEE Press), especially the paper by J. Makhoul (reprinted from *Proceedings of the IEEE*, vol. 63, p. 561, 1975).

Burg, J.P. 1968, reprinted in Childers, 1978. [1]

Anderson, N. 1974, reprinted in Childers, 1978. [2]

Cressie, N. 1991, in *Spatial Statistics and Digital Image Analysis* (Washington: National Academy Press). [3]

Press, W.H., and Rybicki, G.B. 1992, *Astrophysical Journal*, vol. 398, pp. 169–176. [4]

13.7 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

$$z \equiv e^{2\pi i f \Delta} \qquad (13.7.1)$$

where Δ 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 (13.7.1) to equations (13.4.4) and (13.4.6), we see that the FFT power spectrum estimate (13.4.5) for any real sampled function $c_k \equiv c(t_k)$ can be written, except for normalization convention, as

$$P(f) = \left| \sum_{k=-N/2}^{N/2-1} c_k z^k \right|^2 \tag{13.7.2}$$

Of course, (13.7.2) 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 (13.7.2), the finite Laurent series offers, in general, only an approximation to a general analytic function of z. In fact, a formal expression for representing "true" power spectra (up to normalization) is

$$P(f) = \left| \sum_{k=-\infty}^{\infty} c_k z^k \right|^2 \tag{13.7.3}$$

This is an infinite Laurent series which depends on an infinite number of values c_k. Equation (13.7.2) is just one kind of analytic approximation to the analytic function of z represented by (13.7.3); the kind, in fact, that is implicit in the use of FFTs to estimate power spectra by periodogram methods. It goes under several names, including *direct method, all-zero model,* and *moving average (MA) model*. The term "all-zero" in particular refers to the fact that the model spectrum can have zeros in the z-plane, but not poles.

If we look at the problem of approximating (13.7.3) more generally it seems clear that we could do a better job with a rational function, one with a series of type (13.7.2) in both the numerator and the denominator. Less obviously, it turns out that there are some advantages in an approximation whose free parameters all lie in the *denominator*, namely,

$$P(f) \approx \frac{1}{\left| \sum_{k=-M/2}^{M/2} b_k z^k \right|^2} = \frac{a_0}{\left| 1 + \sum_{k=1}^{M} a_k z^k \right|^2} \tag{13.7.4}$$

Here the second equality brings in a new set of coefficients a_k's, which can be determined from the b_k's using the fact that z lies on the unit circle. The b_k's can be thought of as being determined by the condition that power series expansion of (13.7.4) agree with the first $M + 1$ terms of (13.7.3). In practice, as we shall see, one determines the b_k's or a_k's by another method.

The differences between the approximations (13.7.2) and (13.7.4) are not just cosmetic. They are approximations with very different character. Most notable is the fact that (13.7.4) can have *poles*, corresponding to infinite power spectral density, on the unit z-circle, i.e., at real frequencies in the Nyquist interval. Such poles can provide an accurate representation for underlying power spectra that have sharp, discrete "lines" or delta-functions. By contrast, (13.7.2) can have only zeros, not poles, at real frequencies in the Nyquist interval, and must thus attempt to fit sharp spectral features with, essentially, a polynomial. The approximation (13.7.4) goes under several names: *all-poles model, maximum entropy method (MEM), autoregressive model (AR)*. We need only find out how to compute the coefficients a_0 and the a_k's from a data set, so that we can actually use (13.7.4) to obtain spectral estimates.

A pleasant surprise is that we already know how! Look at equation (13.6.11) for linear prediction. Compare it with linear filter equations (13.5.1) and (13.5.2), and you will see that, viewed as a filter that takes input x's into output y's, linear prediction has a filter function

$$\mathcal{H}(f) = \frac{1}{1 - \sum_{j=1}^{N} d_j z^{-j}} \tag{13.7.5}$$

Thus, the power spectrum of the y's should be equal to the power spectrum of the x's multiplied by $|\mathcal{H}(f)|^2$. Now let us think about what the spectrum of the input x's is, when they are residual discrepancies from linear prediction. Although we will not prove it formally, it is intuitively believable that the x's are independently random and therefore have a flat (white noise) spectrum. (Roughly speaking, any residual correlations left in the x's would have allowed a more accurate linear prediction, and would have been removed.) The overall normalization of this flat spectrum is just the mean square amplitude of the x's. But this is exactly the quantity computed in equation (13.6.13) and returned by the routine `memcof` as `xms`. Thus, the coefficients a_0 and a_k in equation (13.7.4) are related to the LP coefficients returned by `memcof` simply by

$$a_0 = \texttt{xms} \qquad a_k = -\texttt{d}(k), \quad k = 1, \dots, M \tag{13.7.6}$$

There is also another way to describe the relation between the a_k's and the autocorrelation components ϕ_k. The Wiener-Khinchin theorem (12.0.12) says that the Fourier transform of the autocorrelation is equal to the power spectrum. In z-transform language, this Fourier transform is just a Laurent series in z. The equation that is to be satisfied by the coefficients in equation (13.7.4) is thus

$$\frac{a_0}{\left|1 + \sum\limits_{k=1}^{M} a_k z^k\right|^2} \approx \sum_{j=-M}^{M} \phi_j z^j \tag{13.7.7}$$

The approximately equal sign in (13.7.7) has a somewhat special interpretation. It means that the series expansion of the left-hand side is supposed to agree with the right-hand side term by term from z^{-M} to z^M. Outside this range of terms, the right-hand side is obviously zero, while the left-hand side will still have nonzero terms. Notice that M, the number of coefficients in the approximation on the left-hand side, can be any integer up to N, the total number of autocorrelations available. (In practice, one often chooses M much smaller than N.) M is called the *order* or *number of poles* of the approximation.

Whatever the chosen value of M, the series expansion of the left-hand side of (13.7.7) defines a certain sort of *extrapolation* of the autocorrelation function to lags larger than M, in fact even to lags larger than N, i.e., *larger than the run of data can actually measure*. It turns out that this particular extrapolation can be shown to have, among all possible extrapolations, the maximum *entropy* in a definable information-theoretic sense. Hence the name *maximum entropy method*, or MEM. The maximum entropy property has caused MEM to acquire a certain "cult" popularity; one sometimes hears that it gives an intrinsically "better" estimate than is given by other methods. Don't believe it. MEM has the very cute property of being able to fit sharp spectral features, but there is nothing else magical about its power spectrum estimates.

The operations count in `memcof` scales as the product of N (the number of data points) and M (the desired order of the MEM approximation). If M were chosen to be as large as N, then the method would be much slower than the $N \log N$ FFT methods of the previous section. In practice, however, one usually wants to limit the order (or number of poles) of the MEM approximation to a few times the number of sharp spectral features that one desires it to fit. With this restricted number of poles, the method will smooth the spectrum somewhat, but this is often a desirable property. While exact values depend on the application, one might take $M = 10$ or 20 or 50 for $N = 1000$ or 10000. In that case MEM estimation is not much slower than FFT estimation.

We feel obliged to warn you that `memcof` can be a bit quirky at times. If the number of poles or number of data points is too large, roundoff error can be a problem, even in double precision. With "peaky" data (i.e., data with extremely sharp spectral features), the algorithm may suggest split peaks even at modest orders, and the peaks may shift with the phase of the sine wave. Also, with noisy input functions, if you choose too high an order, you will find spurious peaks galore! Some experts recommend the use of this algorithm in conjunction with more conservative methods, like periodograms, to help choose the correct model order, and to avoid getting too fooled by spurious spectral features. MEM can be finicky, but it can also do remarkable things. We recommend that you try it out, cautiously, on your own problems. We now turn to the evaluation of the MEM spectral estimate from its coefficients.

The MEM estimation (13.7.4) is a function of continuously varying frequency f. There is no special significance to specific equally spaced frequencies as there was in the FFT case.

In fact, since the MEM estimate may have very sharp spectral features, one wants to be able to evaluate it on a very fine mesh near to those features, but perhaps only more coarsely farther away from them. Here is a function which, given the coefficients already computed, evaluates (13.7.4) and returns the estimated power spectrum as a function of $f\Delta$ (the frequency times the sampling interval). Of course, $f\Delta$ should lie in the Nyquist range between $-1/2$ and $1/2$.

```
#include <math.h>

float evlmem(float fdt, float d[], int m, float xms)
Given d[1..m], m, xms as returned by memcof, this function returns the power spectrum
estimate P(f) as a function of fdt = fΔ.
{
    int i;
    float sumr=1.0,sumi=0.0;
    double wr=1.0,wi=0.0,wpr,wpi,wtemp,theta;     Trig. recurrences in double precision.

    theta=6.28318530717959*fdt;
    wpr=cos(theta);                           Set up for recurrence relations.
    wpi=sin(theta);
    for (i=1;i<=m;i++) {                      Loop over the terms in the sum.
        wr=(wtemp=wr)*wpr-wi*wpi;
        wi=wi*wpr+wtemp*wpi;
        sumr -= d[i]*wr;                      These accumulate the denominator of (13.7.4).
        sumi -= d[i]*wi;
    }
    return xms/(sumr*sumr+sumi*sumi);         Equation (13.7.4).
}
```

Be sure to evaluate $P(f)$ on a fine enough grid to *find* any narrow features that may be there! Such narrow features, if present, can contain virtually all of the power in the data. You might also wish to know how the $P(f)$ produced by the routines memcof and evlmem is normalized with respect to the mean square value of the input data vector. The answer is

$$\int_{-1/2}^{1/2} P(f\Delta)d(f\Delta) = 2\int_{0}^{1/2} P(f\Delta)d(f\Delta) = \text{mean square value of data} \qquad (13.7.8)$$

Sample spectra produced by the routines memcof and evlmem are shown in Figure 13.7.1.

CITED REFERENCES AND FURTHER READING:

Childers, D.G. (ed.) 1978, *Modern Spectrum Analysis* (New York: IEEE Press), Chapter II.

Kay, S.M., and Marple, S.L. 1981, *Proceedings of the IEEE*, vol. 69, pp. 1380–1419.

13.8 Spectral Analysis of Unevenly Sampled Data

Thus far, we have been dealing exclusively with evenly sampled data,

$$h_n = h(n\Delta) \qquad n = \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots \qquad (13.8.1)$$

where Δ is the sampling interval, whose reciprocal is the sampling rate. Recall also (§12.1) the significance of the Nyquist critical frequency

$$f_c \equiv \frac{1}{2\Delta} \qquad (13.8.2)$$

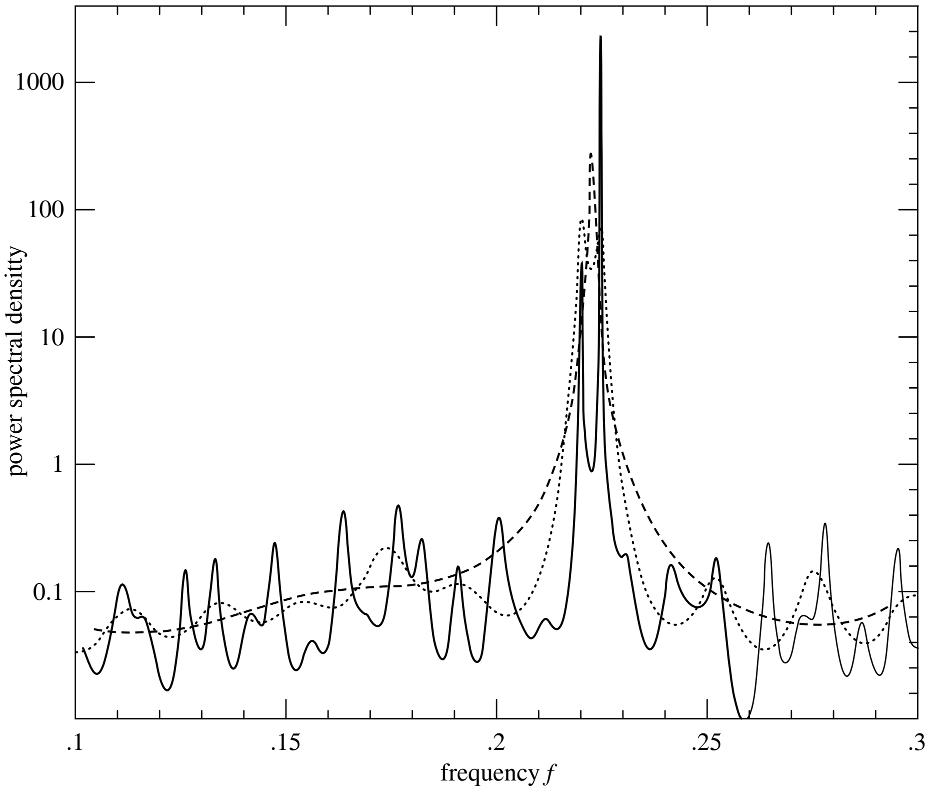


Figure 13.7.1. Sample output of maximum entropy spectral estimation. The input signal consists of 512 samples of the sum of two sinusoids of very nearly the same frequency, plus white noise with about equal power. Shown is an expanded portion of the full Nyquist frequency interval (which would extend from zero to 0.5). The dashed spectral estimate uses 20 poles; the dotted, 40; the solid, 150. With the larger number of poles, the method can resolve the distinct sinusoids; but the flat noise background is beginning to show spurious peaks. (Note logarithmic scale.)

as codified by the sampling theorem: A sampled data set like equation (13.8.1) contains *complete* information about all spectral components in a signal $h(t)$ up to the Nyquist frequency, and scrambled or *aliased* information about any signal components at frequencies larger than the Nyquist frequency. The sampling theorem thus defines both the attractiveness, and the limitation, of any analysis of an evenly spaced data set.

There are situations, however, where evenly spaced data cannot be obtained. A common case is where instrumental drop-outs occur, so that data is obtained only on a (not consecutive integer) subset of equation (13.8.1), the so-called *missing data* problem. Another case, common in observational sciences like astronomy, is that the observer cannot completely control the time of the observations, but must simply accept a certain dictated set of t_i's.

There are some obvious ways to get from unevenly spaced t_i's to evenly spaced ones, as in equation (13.8.1). Interpolation is one way: lay down a grid of evenly spaced times on your data and interpolate values onto that grid; then use FFT methods. In the missing data problem, you only have to interpolate on missing data points. If a lot of consecutive points are missing, you might as well just set them to zero, or perhaps "clamp" the value at the last measured point. However, the experience of practitioners of such interpolation techniques *is not reassuring*. Generally speaking, such techniques perform poorly. Long gaps in the data, for example, often produce a spurious bulge of power at low frequencies (wavelengths comparable to gaps).

A completely different method of spectral analysis for unevenly sampled data, one that mitigates these difficulties and has some other very desirable properties, was developed by Lomb [1], based in part on earlier work by Barning [2] and Vaníček [3], and additionally elaborated by Scargle [4]. The Lomb method (as we will call it) evaluates data, and sines and cosines, only at times t_i that are actually measured. Suppose that there are N data points $h_i \equiv h(t_i)$, $i = 1, \dots, N$. Then first find the mean and variance of the data by the usual formulas,

$$\overline{h} \equiv \frac{1}{N}\sum_1^N h_i \qquad \sigma^2 \equiv \frac{1}{N-1}\sum_1^N (h_i - \overline{h})^2 \tag{13.8.3}$$

Now, the Lomb *normalized periodogram* (spectral power as a function of angular frequency $\omega \equiv 2\pi f > 0$) is defined by

$$P_N(\omega) \equiv \frac{1}{2\sigma^2}\left\{ \frac{\left[\sum_j (h_j - \overline{h})\cos\omega(t_j-\tau)\right]^2}{\sum_j \cos^2\omega(t_j-\tau)} + \frac{\left[\sum_j (h_j - \overline{h})\sin\omega(t_j-\tau)\right]^2}{\sum_j \sin^2\omega(t_j-\tau)} \right\} \tag{13.8.4}$$

Here τ is defined by the relation

$$\tan(2\omega\tau) = \frac{\sum_j \sin 2\omega t_j}{\sum_j \cos 2\omega t_j} \tag{13.8.5}$$

The constant τ is a kind of offset that makes $P_N(\omega)$ completely independent of shifting all the t_i's by any constant. Lomb shows that this particular choice of offset has another, deeper, effect: It makes equation (13.8.4) identical to the equation that one would obtain if one estimated the harmonic content of a data set, at a given frequency ω, by linear least-squares fitting to the model

$$h(t) = A\cos\omega t + B\sin\omega t \tag{13.8.6}$$

This fact gives some insight into why the method can give results superior to FFT methods: It weights the data on a "per point" basis instead of on a "per time interval" basis, when uneven sampling can render the latter seriously in error.

A very common occurrence is that the measured data points h_i are the sum of a periodic signal and independent (white) Gaussian noise. If we are trying to determine the presence or absence of such a periodic signal, we want to be able to give a quantitative answer to the question, "How significant is a peak in the spectrum $P_N(\omega)$?" In this question, the null hypothesis is that the data values are independent Gaussian random values. A very nice property of the Lomb normalized periodogram is that the viability of the null hypothesis can be tested fairly rigorously, as we now discuss.

The word "normalized" refers to the factor σ^2 in the denominator of equation (13.8.4). Scargle [4] shows that with this normalization, at any particular ω and *in the case of the null hypothesis*, $P_N(\omega)$ has an exponential probability distribution with unit mean. In other words, the probability that $P_N(\omega)$ will be between some positive z and $z + dz$ is $\exp(-z)dz$. It readily follows that, if we scan some M *independent* frequencies, the probability that none give values larger than z is $(1 - e^{-z})^M$. So

$$P(>z) \equiv 1 - (1 - e^{-z})^M \tag{13.8.7}$$

is the false-alarm probability of the null hypothesis, that is, the *significance level* of any peak in $P_N(\omega)$ that we do see. A small value for the false-alarm probability indicates a highly significant periodic signal.

To evaluate this significance, we need to know M. After all, the more frequencies we look at, the less significant is some one modest bump in the spectrum. (Look long enough, find anything!) A typical procedure will be to plot $P_N(\omega)$ as a function of many closely spaced frequencies in some large frequency range. How many of these are independent?

Before answering, let us first see how accurately we need to know M. The interesting region is where the significance is a small (significant) number, $\ll 1$. There, equation (13.8.7) can be series expanded to give

$$P(>z) \approx Me^{-z} \tag{13.8.8}$$

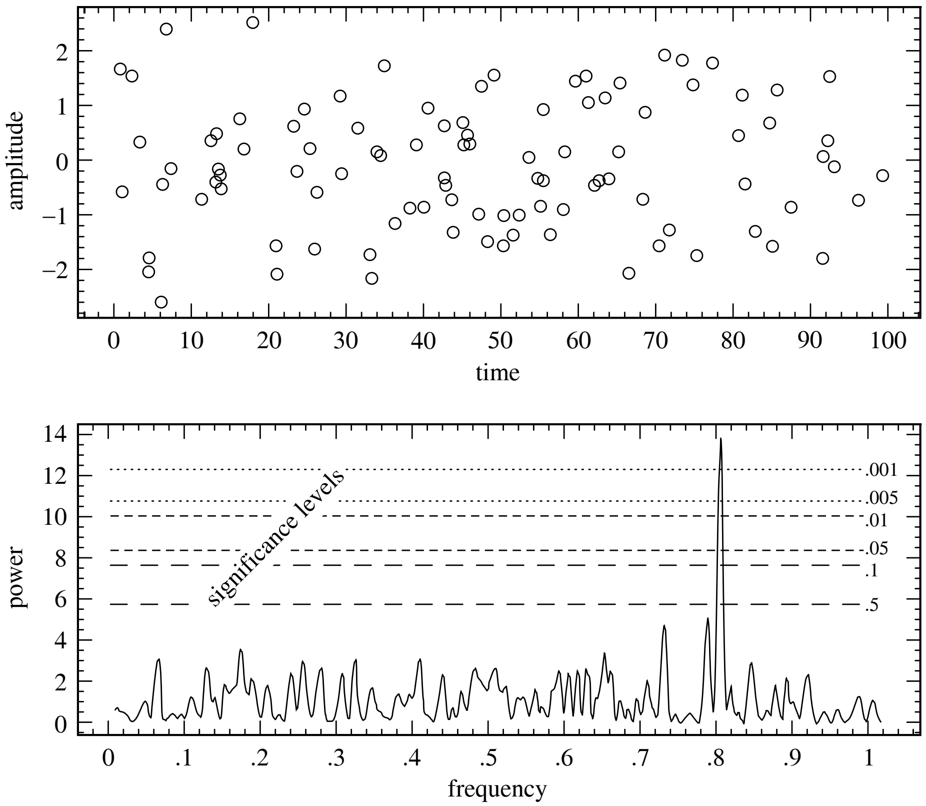


Figure 13.8.1. Example of the Lomb algorithm in action. The 100 data points (upper figure) are at random times between 0 and 100. Their sinusoidal component is readily uncovered (lower figure) by the algorithm, at a significance level better than 0.001. If the 100 data points had been evenly spaced at unit interval, the Nyquist critical frequency would have been 0.5. Note that, for these unevenly spaced points, there is no visible aliasing into the Nyquist range.

We see that the significance scales linearly with M. Practical significance levels are numbers like 0.05, 0.01, 0.001, etc. An error of even $\pm 50\%$ in the estimated significance is often tolerable, since quoted significance levels are typically spaced apart by factors of 5 or 10. So our estimate of M need not be very accurate.

Horne and Baliunas [5] give results from extensive Monte Carlo experiments for determining M in various cases. In general M depends on the number of frequencies sampled, the number of data points N, and their detailed spacing. It turns out that M is very nearly equal to N when the data points are approximately equally spaced, and when the sampled frequencies "fill" (oversample) the frequency range from 0 to the Nyquist frequency f_c (equation 13.8.2). Further, the value of M is not importantly different for random spacing of the data points than for equal spacing. When a larger frequency range than the Nyquist range is sampled, M increases proportionally. About the only case where M differs significantly from the case of evenly spaced points is when the points are closely clumped, say into groups of 3; then (as one would expect) the number of independent frequencies is reduced by a factor of about 3.

The program `period`, below, calculates an effective value for M based on the above rough-and-ready rules and assumes that there is no important clumping. This will be adequate for most purposes. In any particular case, if it really matters, it is not too difficult to compute a better value of M by simple Monte Carlo: Holding fixed the number of data points and their locations t_i, generate synthetic data sets of Gaussian (normal) deviates, find the largest values of $P_N(\omega)$ for each such data set (using the accompanying program), and fit the resulting distribution for M in equation (13.8.7).

Figure 13.8.1 shows the results of applying the method as discussed so far. In the upper figure, the data points are plotted against time. Their number is $N = 100$, and their distribution in t is Poisson random. There is certainly no sinusoidal signal evident to the eye. The lower figure plots $P_N(\omega)$ against frequency $f = \omega/2\pi$. The Nyquist critical frequency that would obtain if the points were evenly spaced is at $f = f_c = 0.5$. Since we have searched up to about twice that frequency, and oversampled the f's to the point where successive values of $P_N(\omega)$ vary smoothly, we take $M = 2N$. The horizontal dashed and dotted lines are (respectively from bottom to top) significance levels 0.5, 0.1, 0.05, 0.01, 0.005, and 0.001. One sees a highly significant peak at a frequency of 0.81. That is in fact the frequency of the sine wave that is present in the data. (You will have to take our word for this!)

Note that two other peaks approach, but do not exceed the 50% significance level; that is about what one might expect by chance. It is also worth commenting on the fact that the significant peak was found (correctly) *above the Nyquist frequency* and without any significant aliasing down into the Nyquist interval! That would not be possible for evenly spaced data. It is possible here because the randomly spaced data has *some* points spaced much closer than the "average" sampling rate, and these remove ambiguity from any aliasing.

Implementation of the normalized periodogram in code is straightforward, with, however, a few points to be kept in mind. We are dealing with a *slow* algorithm. Typically, for N data points, we may wish to examine on the order of $2N$ or $4N$ frequencies. Each combination of frequency and data point has, in equations (13.8.4) and (13.8.5), not just a few adds or multiplies, but four calls to trigonometric functions; the operations count can easily reach several hundred times N^2. It is highly desirable — in fact results in a factor 4 speedup — to replace these trigonometric calls by recurrences. That is possible only if the sequence of frequencies examined is a linear sequence. Since such a sequence is probably what most users would want anyway, we have built this into the implementation.

At the end of this section we describe a way to evaluate equations (13.8.4) and (13.8.5) — approximately, but to any desired degree of approximation — by a fast method [6] whose operation count goes only as $N \log N$. This faster method should be used for long data sets.

The lowest independent frequency f to be examined is the inverse of the span of the input data, $\max_i(t_i) - \min_i(t_i) \equiv T$. This is the frequency such that the data can include one complete cycle. In subtracting off the data's mean, equation (13.8.4) already assumed that you are not interested in the data's zero-frequency piece — which is just that mean value. In an FFT method, higher independent frequencies would be integer multiples of $1/T$. Because we are interested in the statistical significance of any peak that may occur, however, we had better (over-) sample more finely than at interval $1/T$, so that sample points lie close to the top of any peak. Thus, the accompanying program includes an oversampling parameter, called `ofac`; a value `ofac` $\gtrsim 4$ might be typical in use. We also want to specify how high in frequency to go, say f_{hi}. One guide to choosing f_{hi} is to compare it with the Nyquist frequency f_c which would obtain if the N data points were evenly spaced over the same span T, that is $f_c = N/(2T)$. The accompanying program includes an input parameter `hifac`, defined as f_{hi}/f_c. The number of different frequencies N_P returned by the program is then given by

$$N_P = \frac{\texttt{ofac} \times \texttt{hifac}}{2} N \tag{13.8.9}$$

(You have to remember to dimension the output arrays to at least this size.)

The code does the trigonometric recurrences in double precision and embodies a few tricks with trigonometric identities, to decrease roundoff errors. If you are an aficionado of such things you can puzzle it out. A final detail is that equation (13.8.7) will fail because of roundoff error if z is too large; but equation (13.8.8) is fine in this regime.

```
#include <math.h>
#include "nrutil.h"
#define TWOPID 6.2831853071795865

void period(float x[], float y[], int n, float ofac, float hifac, float px[],
    float py[], int np, int *nout, int *jmax, float *prob)
```
Given `n` data points with abscissas `x[1..n]` (which need not be equally spaced) and ordinates `y[1..n]`, and given a desired oversampling factor `ofac` (a typical value being 4 or larger), this routine fills array `px[1..np]` with an increasing sequence of frequencies (not angular

frequencies) up to hifac times the "average" Nyquist frequency, and fills array py[1..np] with the values of the Lomb normalized periodogram at those frequencies. The arrays x and y are not altered. np, the dimension of px and py, must be large enough to contain the output, or an error results. The routine also returns jmax such that py[jmax] is the maximum element in py, and prob, an estimate of the significance of that maximum against the hypothesis of random noise. A small value of prob indicates that a significant periodic signal is present.

```
{
    void avevar(float data[], unsigned long n, float *ave, float *var);
    int i,j;
    float ave,c,cc,cwtau,effm,expy,pnow,pymax,s,ss,sumc,sumcy,sums,sumsh,
        sumsy,swtau,var,wtau,xave,xdif,xmax,xmin,yy;
    double arg,wtemp,*wi,*wpi,*wpr,*wr;

    wi=dvector(1,n);
    wpi=dvector(1,n);
    wpr=dvector(1,n);
    wr=dvector(1,n);
    *nout=0.5*ofac*hifac*n;
    if (*nout > np) nrerror("output arrays too short in period");
    avevar(y,n,&ave,&var);                  Get mean and variance of the input data.
    if (var == 0.0) nrerror("zero variance in period");
    xmax=xmin=x[1];                         Go through data to get the range of abscis-
    for (j=1;j<=n;j++) {                        sas.
        if (x[j] > xmax) xmax=x[j];
        if (x[j] < xmin) xmin=x[j];
    }
    xdif=xmax-xmin;
    xave=0.5*(xmax+xmin);
    pymax=0.0;
    pnow=1.0/(xdif*ofac);                   Starting frequency.
    for (j=1;j<=n;j++) {                    Initialize values for the trigonometric recur-
        arg=TWOPID*((x[j]-xave)*pnow);          rences at each data point. The recur-
        wpr[j] = -2.0*SQR(sin(0.5*arg));        rences are done in double precision.
        wpi[j]=sin(arg);
        wr[j]=cos(arg);
        wi[j]=wpi[j];
    }
    for (i=1;i<=(*nout);i++) {              Main loop over the frequencies to be evalu-
        px[i]=pnow;                             ated.
        sumsh=sumc=0.0;                     First, loop over the data to get τ and related
        for (j=1;j<=n;j++) {                    quantities.
            c=wr[j];
            s=wi[j];
            sumsh += s*c;
            sumc += (c-s)*(c+s);
        }
        wtau=0.5*atan2(2.0*sumsh,sumc);
        swtau=sin(wtau);
        cwtau=cos(wtau);
        sums=sumc=sumsy=sumcy=0.0;          Then, loop over the data again to get the
        for (j=1;j<=n;j++) {                    periodogram value.
            s=wi[j];
            c=wr[j];
            ss=s*cwtau-c*swtau;
            cc=c*cwtau+s*swtau;
            sums += ss*ss;
            sumc += cc*cc;
            yy=y[j]-ave;
            sumsy += yy*ss;
            sumcy += yy*cc;
            wr[j]=((wtemp=wr[j])*wpr[j]-wi[j]*wpi[j])+wr[j];     Update the trigono-
            wi[j]=(wi[j]*wpr[j]+wtemp*wpi[j])+wi[j];             metric recurrences.
        }
        py[i]=0.5*(sumcy*sumcy/sumc+sumsy*sumsy/sums)/var;
        if (py[i] >= pymax) pymax=py[(*jmax=i)];
        pnow += 1.0/(ofac*xdif);          The next frequency.
    }
    expy=exp(-pymax);                     Evaluate statistical significance of the max-
    effm=2.0*(*nout)/ofac;                    imum.
    *prob=effm*expy;
    if (*prob > 0.01) *prob=1.0-pow(1.0-expy,effm);
    free_dvector(wr,1,n);
    free_dvector(wpr,1,n);
    free_dvector(wpi,1,n);
    free_dvector(wi,1,n);
}
```

Fast Computation of the Lomb Periodogram

We here show how equations (13.8.4) and (13.8.5) can be calculated — approximately, but to any desired precision — with an operation count only of order $N_P \log N_P$. The method uses the FFT, but it is in no sense an FFT periodogram of the data. It is an actual evaluation of equations (13.8.4) and (13.8.5), the Lomb normalized periodogram, with exactly that method's strengths and weaknesses. This fast algorithm, due to Press and Rybicki [6], makes feasible the application of the Lomb method to data sets at least as large as 10^6 points; it is already faster than straightforward evaluation of equations (13.8.4) and (13.8.5) for data sets as small as 60 or 100 points.

Notice that the trigonometric sums that occur in equations (13.8.5) and (13.8.4) can be reduced to four simpler sums. If we define

$$S_h \equiv \sum_{j=1}^{N} (h_j - \bar{h})\, \sin(\omega t_j) \qquad C_h \equiv \sum_{j=1}^{N} (h_j - \bar{h})\, \cos(\omega t_j) \tag{13.8.10}$$

and

$$S_2 \equiv \sum_{j=1}^{N} \sin(2\omega t_j) \qquad C_2 \equiv \sum_{j=1}^{N} \cos(2\omega t_j) \tag{13.8.11}$$

then

$$\begin{aligned}
&\sum_{j=1}^{N} (h_j - \bar{h})\, \cos\omega(t_j - \tau) = C_h \cos\omega\tau + S_h \sin\omega\tau \\
&\sum_{j=1}^{N} (h_j - \bar{h})\, \sin\omega(t_j - \tau) = S_h \cos\omega\tau - C_h \sin\omega\tau \\
&\sum_{j=1}^{N} \cos^2\omega(t_j - \tau) = \frac{N}{2} + \frac{1}{2}C_2 \cos(2\omega\tau) + \frac{1}{2}S_2 \sin(2\omega\tau) \\
&\sum_{j=1}^{N} \sin^2\omega(t_j - \tau) = \frac{N}{2} - \frac{1}{2}C_2 \cos(2\omega\tau) - \frac{1}{2}S_2 \sin(2\omega\tau)
\end{aligned} \tag{13.8.12}$$

Now notice that *if* the t_js *were* evenly spaced, then the four quantities S_h, C_h, S_2, and C_2 could be evaluated by two complex FFTs, and the results could then be substituted back through equation (13.8.12) to evaluate equations (13.8.5) and (13.8.4). The problem is therefore only to evaluate equations (13.8.10) and (13.8.11) for unevenly spaced data.

Interpolation, or rather reverse interpolation — we will here call it *extirpolation* — provides the key. Interpolation, as classically understood, uses several function values on a regular mesh to construct an accurate approximation at an arbitrary point. Extirpolation, just the opposite, *replaces* a function value at an arbitrary point by several function values on a regular mesh, doing this in such a way that sums over the mesh are an accurate approximation to sums over the original arbitrary point.

It is not hard to see that the weight functions for extirpolation are identical to those for interpolation. Suppose that the function $h(t)$ to be extirpolated is known only at the discrete (unevenly spaced) points $h(t_i) \equiv h_i$, and that the function $g(t)$ (which will be, e.g., $\cos\omega t$) can be evaluated anywhere. Let $\hat{t}_k$ be a sequence of evenly spaced points on a regular mesh. Then Lagrange interpolation (§3.1) gives an approximation of the form

$$g(t) \approx \sum_k w_k(t) g(\hat{t}_k) \tag{13.8.13}$$

where $w_k(t)$ are interpolation weights. Now let us evaluate a sum of interest by the following scheme:

$$\sum_{j=1}^{N} h_j g(t_j) \approx \sum_{j=1}^{N} h_j \left[\sum_k w_k(t_j) g(\hat{t}_k)\right] = \sum_k \left[\sum_{j=1}^{N} h_j w_k(t_j)\right] g(\hat{t}_k) \equiv \sum_k \widehat{h}_k\, g(\hat{t}_k) \tag{13.8.14}$$

Here $\widehat{h}_k \equiv \sum_j h_j w_k(t_j)$. Notice that equation (13.8.14) replaces the original sum by one on the regular mesh. Notice also that the accuracy of equation (13.8.13) depends only on the fineness of the mesh with respect to the function g and has nothing to do with the spacing of the points t_j or the function h; therefore the accuracy of equation (13.8.14) also has this property.

The general outline of the fast evaluation method is therefore this: (i) Choose a mesh size large enough to accommodate some desired oversampling factor, and large enough to have several extirpolation points per half-wavelength of the highest frequency of interest. (ii) Extirpolate the values h_i onto the mesh and take the FFT; this gives S_h and C_h in equation (13.8.10). (iii) Extirpolate the constant values 1 onto another mesh, and take its FFT; this, with some manipulation, gives S_2 and C_2 in equation (13.8.11). (iv) Evaluate equations (13.8.12), (13.8.5), and (13.8.4), in that order.

There are several other tricks involved in implementing this algorithm efficiently. You can figure most out from the code, but we will mention the following points: (a) A nice way to get transform values at frequencies 2ω instead of ω is to stretch the time-domain data by a factor 2, and then wrap it to double-cover the original length. (This trick goes back to Tukey.) In the program, this appears as a modulo function. (b) Trigonometric identities are used to get from the left-hand side of equation (13.8.5) to the various needed trigonometric functions of $\omega\tau$. C identifiers like (e.g.) `cwt` and `hs2wt` represent quantities like (e.g.) $\cos\omega\tau$ and $\frac{1}{2}\sin(2\omega\tau)$. (c) The function `spread` does extirpolation onto the M most nearly centered mesh points around an arbitrary point; its turgid code evaluates coefficients of the Lagrange interpolating polynomials, in an efficient manner.

```
#include <math.h>
#include "nrutil.h"
#define MOD(a,b)    while(a >= b) a -= b;       Positive numbers only.
#define MACC 4                                  Number of interpolation points per 1/4
                                                   cycle of highest frequency.
void fasper(float x[], float y[], unsigned long n, float ofac, float hifac,
    float wk1[], float wk2[], unsigned long nwk, unsigned long *nout,
    unsigned long *jmax, float *prob)
```

Given `n` data points with abscissas `x[1..n]` (which need not be equally spaced) and ordinates `y[1..n]`, and given a desired oversampling factor `ofac` (a typical value being 4 or larger), this routine fills array `wk1[1..nwk]` with a sequence of `nout` increasing frequencies (not angular frequencies) up to `hifac` times the "average" Nyquist frequency, and fills array `wk2[1..nwk]` with the values of the Lomb normalized periodogram at those frequencies. The arrays `x` and `y` are not altered. `nwk`, the dimension of `wk1` and `wk2`, must be large enough for intermediate work space, or an error results. The routine also returns `jmax` such that `wk2[jmax]` is the maximum element in `wk2`, and `prob`, an estimate of the significance of that maximum against the hypothesis of random noise. A small value of `prob` indicates that a significant periodic signal is present.

```
{
    void avevar(float data[], unsigned long n, float *ave, float *var);
    void realft(float data[], unsigned long n, int isign);
    void spread(float y, float yy[], unsigned long n, float x, int m);
    unsigned long j,k,ndim,nfreq,nfreqt;
    float ave,ck,ckk,cterm,cwt,den,df,effm,expy,fac,fndim,hc2wt;
    float hs2wt,hypo,pmax,sterm,swt,var,xdif,xmax,xmin;
    *nout=0.5*ofac*hifac*n;
    nfreqt=ofac*hifac*n*MACC;                Size the FFT as next power of 2 above
    nfreq=64;                                    nfreqt.
    while (nfreq < nfreqt) nfreq <<= 1;
    ndim=nfreq << 1;
    if (ndim > nwk) nrerror("workspaces too small in fasper");
    avevar(y,n,&ave,&var);                   Compute the mean, variance, and range
    if (var == 0.0) nrerror("zero variance in fasper");     of the data.
    xmin=x[1];
    xmax=xmin;
    for (j=2;j<=n;j++) {
        if (x[j] < xmin) xmin=x[j];
        if (x[j] > xmax) xmax=x[j];
    }
    xdif=xmax-xmin;
    for (j=1;j<=ndim;j++) wk1[j]=wk2[j]=0.0;  Zero the workspaces.
    fac=ndim/(xdif*ofac);
    fndim=ndim;
    for (j=1;j<=n;j++) {                     Extirpolate the data into the workspaces.
        ck=(x[j]-xmin)*fac;
        MOD(ck,fndim)
        ckk=2.0*(ck++);
        MOD(ckk,fndim)
        ++ckk;
        spread(y[j]-ave,wk1,ndim,ck,MACC);
        spread(1.0,wk2,ndim,ckk,MACC);
    }
    realft(wk1,ndim,1);                      Take the Fast Fourier Transforms.
    realft(wk2,ndim,1);
    df=1.0/(xdif*ofac);
    pmax = -1.0;
    for (k=3,j=1;j<=(*nout);j++,k+=2) {      Compute the Lomb value for each fre-
        hypo=sqrt(wk2[k]*wk2[k]+wk2[k+1]*wk2[k+1]);     quency.
        hc2wt=0.5*wk2[k]/hypo;
        hs2wt=0.5*wk2[k+1]/hypo;
        cwt=sqrt(0.5+hc2wt);
        swt=SIGN(sqrt(0.5-hc2wt),hs2wt);
        den=0.5*n+hc2wt*wk2[k]+hs2wt*wk2[k+1];
        cterm=SQR(cwt*wk1[k]+swt*wk1[k+1])/den;
        sterm=SQR(cwt*wk1[k+1]-swt*wk1[k])/(n-den);
        wk1[j]=j*df;
        wk2[j]=(cterm+sterm)/(2.0*var);
        if (wk2[j] > pmax) pmax=wk2[(*jmax=j)];
    }
    expy=exp(-pmax);                         Estimate significance of largest peak value.
    effm=2.0*(*nout)/ofac;
    *prob=effm*expy;
    if (*prob > 0.01) *prob=1.0-pow(1.0-expy,effm);
}
```

```
#include "nrutil.h"

void spread(float y, float yy[], unsigned long n, float x, int m)
```
Given an array `yy[1..n]`, extirpolate (spread) a value `y` into `m` actual array elements that best approximate the "fictional" (i.e., possibly noninteger) array element number `x`. The weights used are coefficients of the Lagrange interpolating polynomial.
```
{
    int ihi,ilo,ix,j,nden;
    static long nfac[11]={0,1,1,2,6,24,120,720,5040,40320,362880};
    float fac;

    if (m > 10) nrerror("factorial table too small in spread");
    ix=(int)x;
    if (x == (float)ix) yy[ix] += y;
    else {
        ilo=LMIN(LMAX((long)(x-0.5*m+1.0),1),n-m+1);
        ihi=ilo+m-1;
        nden=nfac[m];
        fac=x-ilo;
        for (j=ilo+1;j<=ihi;j++) fac *= (x-j);
        yy[ihi] += y*fac/(nden*(x-ihi));
        for (j=ihi-1;j>=ilo;j--) {
            nden=(nden/(j+1-ilo))*(j-ihi);
            yy[j] += y*fac/(nden*(x-j));
        }
    }
}
```

CITED REFERENCES AND FURTHER READING:

Lomb, N.R. 1976, *Astrophysics and Space Science*, vol. 39, pp. 447–462. [1]

Barning, F.J.M. 1963, *Bulletin of the Astronomical Institutes of the Netherlands*, vol. 17, pp. 22–28. [2]

Vaníček, P. 1971, *Astrophysics and Space Science*, vol. 12, pp. 10–33. [3]

Scargle, J.D. 1982, *Astrophysical Journal*, vol. 263, pp. 835–853. [4]

Horne, J.H., and Baliunas, S.L. 1986, *Astrophysical Journal*, vol. 302, pp. 757–763. [5]

Press, W.H. and Rybicki, G.B. 1989, *Astrophysical Journal*, vol. 338, pp. 277–280. [6]

13.9 Computing Fourier Integrals Using the FFT

Not uncommonly, one wants to calculate accurate numerical values for integrals of the form

$$I = \int_a^b e^{i\omega t} h(t)dt \ , \tag{13.9.1}$$

or the equivalent real and imaginary parts

$$I_c = \int_a^b \cos(\omega t)h(t)dt \qquad I_s = \int_a^b \sin(\omega t)h(t)dt \ , \tag{13.9.2}$$

and one wants to evaluate this integral for many different values of ω. In cases of interest, $h(t)$ is often a smooth function, but it is not necessarily periodic in $[a, b]$, nor does it necessarily go to zero at a or b. While it seems intuitively obvious that the *force majeure* of the FFT ought to be applicable to this problem, doing so turns out to be a surprisingly subtle matter, as we will now see.

Let us first approach the problem naively, to see where the difficulty lies. Divide the interval $[a, b]$ into M subintervals, where M is a large integer, and define

$$\Delta \equiv \frac{b-a}{M} \ , \quad t_j \equiv a + j\Delta \ , \quad h_j \equiv h(t_j) \ , \quad j = 0, \ldots, M \tag{13.9.3}$$

Notice that $h_0 = h(a)$ and $h_M = h(b)$, and that there are $M + 1$ values h_j. We can approximate the integral I by a sum,

$$I \approx \Delta \sum_{j=0}^{M-1} h_j \exp(i\omega t_j) \tag{13.9.4}$$

which is at any rate first-order accurate. (If we centered the h_j's and the t_j's in the intervals, we could be accurate to second order.) Now for certain values of ω and M, the sum in equation (13.9.4) can be made into a discrete Fourier transform, or DFT, and evaluated by the fast Fourier transform (FFT) algorithm. In particular, we can choose M to be an integer power of 2, and define a set of special ω's by

$$\omega_m \Delta \equiv \frac{2\pi m}{M} \tag{13.9.5}$$

where m has the values $m = 0, 1, \ldots, M/2 - 1$. Then equation (13.9.4) becomes

$$I(\omega_m) \approx \Delta e^{i\omega_m a} \sum_{j=0}^{M-1} h_j e^{2\pi i m j/M} = \Delta e^{i\omega_m a} [\mathrm{DFT}(h_0 \ldots h_{M-1})]_m \tag{13.9.6}$$

Equation (13.9.6), while simple and clear, is emphatically *not recommended* for use: It is likely to give wrong answers!

The problem lies in the oscillatory nature of the integral (13.9.1). If $h(t)$ is at all smooth, and if ω is large enough to imply several cycles in the interval $[a, b]$ — in fact, ω_m in equation (13.9.5) gives exactly m cycles — then the value of I is typically very small, so small that it is easily swamped by first-order, or even (with centered values) second-order, truncation error. Furthermore, the characteristic "small parameter" that occurs in the error term is not $\Delta/(b-a) = 1/M$, as it would be if the integrand were not oscillatory, but $\omega\Delta$, which can be as large as π for ω's within the Nyquist interval of the DFT (cf. equation 13.9.5). The result is that equation (13.9.6) becomes systematically inaccurate as ω increases.

It is a sobering exercise to implement equation (13.9.6) for an integral that can be done analytically, and to see just how bad it is. We recommend that you try it.

Let us therefore turn to a more sophisticated treatment. Given the sampled points h_j, we can approximate the function $h(t)$ everywhere in the interval $[a, b]$ by interpolation on nearby h_j's. The simplest case is linear interpolation, using the two nearest h_j's, one to the left and one to the right. A higher-order interpolation, e.g., would be cubic interpolation, using two points to the left and two to the right — except in the first and last subintervals, where we must interpolate with three h_j's on one side, one on the other.

The formulas for such interpolation schemes are (piecewise) polynomial in the independent variable t, but with coefficients that are of course linear in the function values h_j. Although one does not usually think of it in this way, interpolation can be viewed as approximating a function by a sum of kernel functions (which depend only on the interpolation scheme) times sample values (which depend only on the function). Let us write

$$h(t) \approx \sum_{j=0}^{M} h_j \, \psi\left(\frac{t - t_j}{\Delta}\right) + \sum_{j=\text{endpoints}} h_j \, \varphi_j\left(\frac{t - t_j}{\Delta}\right) \tag{13.9.7}$$

Here $\psi(s)$ is the kernel function of an interior point: It is zero for s sufficiently negative or sufficiently positive, and becomes nonzero only when s is in the range where the h_j multiplying it is actually used in the interpolation. We always have $\psi(0) = 1$ and $\psi(m) = 0$, $m = \pm 1, \pm 2, \ldots$, since interpolation right on a sample point should give the sampled function value. For linear interpolation $\psi(s)$ is piecewise linear, rises from 0 to 1 for s in $(-1, 0)$, and falls back to 0 for s in $(0, 1)$. For higher-order interpolation, $\psi(s)$ is made up piecewise of segments of Lagrange interpolation polynomials. It has discontinuous derivatives at integer values of s, where the pieces join, because the set of points used in the interpolation changes discretely.

As already remarked, the subintervals closest to a and b require different (noncentered) interpolation formulas. This is reflected in equation (13.9.7) by the second sum, with the special endpoint kernels $\varphi_j(s)$. Actually, for reasons that will become clearer below, we have included *all* the points in the *first* sum (with kernel ψ), so the φ_j's are actually differences between true endpoint kernels and the interior kernel ψ. It is a tedious, but straightforward, exercise to write down all the $\varphi_j(s)$'s for any particular order of interpolation, each one consisting of differences of Lagrange interpolating polynomials spliced together piecewise.

Now apply the integral operator $\int_a^b dt \exp(i\omega t)$ to both sides of equation (13.9.7), interchange the sums and integral, and make the changes of variable $s = (t - t_j)/\Delta$ in the first sum, $s = (t - a)/\Delta$ in the second sum. The result is

$$I \approx \Delta e^{i\omega a}\left[W(\theta)\sum_{j=0}^{M} h_j e^{ij\theta} + \sum_{j=\text{endpoints}} h_j \alpha_j(\theta)\right] \tag{13.9.8}$$

Here $\theta \equiv \omega\Delta$, and the functions $W(\theta)$ and $\alpha_j(\theta)$ are defined by

$$W(\theta) \equiv \int_{-\infty}^{\infty} ds\, e^{i\theta s}\psi(s) \tag{13.9.9}$$

$$\alpha_j(\theta) \equiv \int_{-\infty}^{\infty} ds\, e^{i\theta s}\varphi_j(s-j) \tag{13.9.10}$$

The key point is that equations (13.9.9) and (13.9.10) can be evaluated, analytically, once and for all, for any given interpolation scheme. Then equation (13.9.8) is an algorithm for applying "endpoint corrections" to a sum which (as we will see) can be done using the FFT, giving a result with high-order accuracy.

We will consider only interpolations that are left-right symmetric. Then symmetry implies

$$\varphi_{M-j}(s) = \varphi_j(-s) \qquad \alpha_{M-j}(\theta) = e^{i\theta M}\overset{*}{\alpha}_j(\theta) = e^{i\omega(b-a)}\overset{*}{\alpha}_j(\theta) \tag{13.9.11}$$

where * denotes complex conjugation. Also, $\psi(s) = \psi(-s)$ implies that $W(\theta)$ is real.

Turn now to the first sum in equation (13.9.8), which we want to do by FFT methods. To do so, choose some N that is an integer power of 2 with $N \geq M + 1$. (Note that M need not be a power of two, so $M = N - 1$ is allowed.) If $N > M + 1$, define $h_j \equiv 0$, $M + 1 < j \leq N - 1$, i.e., "zero pad" the array of h_j's so that j takes on the range $0 \leq j \leq N - 1$. Then the sum can be done as a DFT for the special values $\omega = \omega_n$ given by

$$\omega_n\Delta \equiv \frac{2\pi n}{N} \equiv \theta \qquad n = 0, 1, \ldots, \frac{N}{2} - 1 \tag{13.9.12}$$

For fixed M, the larger N is chosen, the finer the sampling in frequency space. The value M, on the other hand, determines the *highest* frequency sampled, since Δ decreases with increasing M (equation 13.9.3), and the largest value of $\omega\Delta$ is always just under π (equation 13.9.12). In general it is advantageous to oversample by *at least* a factor of 4, i.e., $N > 4M$ (see below). We can now rewrite equation (13.9.8) in its final form as

$$\begin{aligned} I(\omega_n) = \Delta e^{i\omega_n a}\Big\{ & W(\theta)[\text{DFT}(h_0 \ldots h_{N-1})]_n \\ & + \alpha_0(\theta)h_0 + \alpha_1(\theta)h_1 + \alpha_2(\theta)h_2 + \alpha_3(\theta)h_3 + \ldots \\ & + e^{i\omega(b-a)}\left[\overset{*}{\alpha}_0(\theta)h_M + \overset{*}{\alpha}_1(\theta)h_{M-1} + \overset{*}{\alpha}_2(\theta)h_{M-2} + \overset{*}{\alpha}_3(\theta)h_{M-3} + \ldots\right]\Big\} \end{aligned} \tag{13.9.13}$$

For cubic (or lower) polynomial interpolation, at most the terms explicitly shown above are nonzero; the ellipses (...) can therefore be ignored, and we need explicit forms only for the functions $W, \alpha_0, \alpha_1, \alpha_2, \alpha_3$, calculated with equations (13.9.9) and (13.9.10). We have worked these out for you, in the trapezoidal (second-order) and cubic (fourth-order) cases. Here are the results, along with the first few terms of their power series expansions for small θ:

Trapezoidal order:

$$W(\theta) = \frac{2(1-\cos\theta)}{\theta^2} \approx 1 - \frac{1}{12}\theta^2 + \frac{1}{360}\theta^4 - \frac{1}{20160}\theta^6$$

$$\begin{aligned} \alpha_0(\theta) &= -\frac{(1-\cos\theta)}{\theta^2} + i\frac{(\theta - \sin\theta)}{\theta^2} \\ &\approx -\frac{1}{2} + \frac{1}{24}\theta^2 - \frac{1}{720}\theta^4 + \frac{1}{40320}\theta^6 + i\theta\left(\frac{1}{6} - \frac{1}{120}\theta^2 + \frac{1}{5040}\theta^4 - \frac{1}{362880}\theta^6\right) \end{aligned}$$

$$\alpha_1 = \alpha_2 = \alpha_3 = 0$$

Cubic order:

$$W(\theta) = \left(\frac{6+\theta^2}{3\theta^4}\right)(3 - 4\cos\theta + \cos 2\theta) \approx 1 - \frac{11}{720}\theta^4 + \frac{23}{15120}\theta^6$$

$$\alpha_0(\theta) = \frac{(-42+5\theta^2)+(6+\theta^2)(8\cos\theta - \cos 2\theta)}{6\theta^4} + i\frac{(-12\theta+6\theta^3)+(6+\theta^2)\sin 2\theta}{6\theta^4}$$
$$\approx -\frac{2}{3} + \frac{1}{45}\theta^2 + \frac{103}{15120}\theta^4 - \frac{169}{226800}\theta^6 + i\theta\left(\frac{2}{45} + \frac{2}{105}\theta^2 - \frac{8}{2835}\theta^4 + \frac{86}{467775}\theta^6\right)$$

$$\alpha_1(\theta) = \frac{14(3-\theta^2) - 7(6+\theta^2)\cos\theta}{6\theta^4} + i\frac{30\theta - 5(6+\theta^2)\sin\theta}{6\theta^4}$$
$$\approx \frac{7}{24} - \frac{7}{180}\theta^2 + \frac{5}{3456}\theta^4 - \frac{7}{259200}\theta^6 + i\theta\left(\frac{7}{72} - \frac{1}{168}\theta^2 + \frac{11}{72576}\theta^4 - \frac{13}{5987520}\theta^6\right)$$

$$\alpha_2(\theta) = \frac{-4(3-\theta^2) + 2(6+\theta^2)\cos\theta}{3\theta^4} + i\frac{-12\theta + 2(6+\theta^2)\sin\theta}{3\theta^4}$$
$$\approx -\frac{1}{6} + \frac{1}{45}\theta^2 - \frac{5}{6048}\theta^4 + \frac{1}{64800}\theta^6 + i\theta\left(-\frac{7}{90} + \frac{1}{210}\theta^2 - \frac{11}{90720}\theta^4 + \frac{13}{7484400}\theta^6\right)$$

$$\alpha_3(\theta) = \frac{2(3-\theta^2) - (6+\theta^2)\cos\theta}{6\theta^4} + i\frac{6\theta - (6+\theta^2)\sin\theta}{6\theta^4}$$
$$\approx \frac{1}{24} - \frac{1}{180}\theta^2 + \frac{5}{24192}\theta^4 - \frac{1}{259200}\theta^6 + i\theta\left(\frac{7}{360} - \frac{1}{840}\theta^2 + \frac{11}{362880}\theta^4 - \frac{13}{29937600}\theta^6\right)$$

The program dftcor, below, implements the endpoint corrections for the cubic case. Given input values of ω, Δ, a, b, and an array with the eight values $h_0, \ldots, h_3, h_{M-3}, \ldots, h_M$, it returns the real and imaginary parts of the endpoint corrections in equation (13.9.13), and the factor $W(\theta)$. The code is turgid, but only because the formulas above are complicated. The formulas have cancellations to high powers of θ. It is therefore necessary to compute the right-hand sides in double precision, even when the corrections are desired only to single precision. It is also necessary to use the series expansion for small values of θ. The optimal cross-over value of θ depends on your machine's wordlength, but you can always find it experimentally as the largest value where the two methods give identical results to machine precision.

```
#include <math.h>

void dftcor(float w, float delta, float a, float b, float endpts[],
    float *corre, float *corim, float *corfac)
```
For an integral approximated by a discrete Fourier transform, this routine computes the correction factor that multiplies the DFT and the endpoint correction to be added. Input is the angular frequency w, stepsize delta, lower and upper limits of the integral a and b, while the array endpts contains the first 4 and last 4 function values. The correction factor $W(\theta)$ is returned as corfac, while the real and imaginary parts of the endpoint correction are returned as corre and corim.
```
{
    void nrerror(char error_text[]);
    float a0i,a0r,a1i,a1r,a2i,a2r,a3i,a3r,arg,c,cl,cr,s,sl,sr,t;
    float t2,t4,t6;
    double cth,ctth,spth2,sth,sth4i,stth,th,th2,th4,tmth2,tth4i;

    th=w*delta;
    if (a >= b || th < 0.0e0 || th > 3.1416e0) nrerror("bad arguments to dftcor");
    if (fabs(th) < 5.0e-2) {        Use series.
        t=th;
        t2=t*t;
        t4=t2*t2;
        t6=t4*t2;
        *corfac=1.0-(11.0/720.0)*t4+(23.0/15120.0)*t6;
        a0r=(-2.0/3.0)+t2/45.0+(103.0/15120.0)*t4-(169.0/226800.0)*t6;
        a1r=(7.0/24.0)-(7.0/180.0)*t2+(5.0/3456.0)*t4-(7.0/259200.0)*t6;
        a2r=(-1.0/6.0)+t2/45.0-(5.0/6048.0)*t4+t6/64800.0;
        a3r=(1.0/24.0)-t2/180.0+(5.0/24192.0)*t4-t6/259200.0;
        a0i=t*(2.0/45.0+(2.0/105.0)*t2-(8.0/2835.0)*t4+(86.0/467775.0)*t6);
        a1i=t*(7.0/72.0-t2/168.0+(11.0/72576.0)*t4-(13.0/5987520.0)*t6);
        a2i=t*(-7.0/90.0+t2/210.0-(11.0/90720.0)*t4+(13.0/7484400.0)*t6);
        a3i=t*(7.0/360.0-t2/840.0+(11.0/362880.0)*t4-(13.0/29937600.0)*t6);
    } else {                         Use trigonometric formulas in double precision.
        cth=cos(th);
        sth=sin(th);
        ctth=cth*cth-sth*sth;
        stth=2.0e0*sth*cth;
        th2=th*th;
        th4=th2*th2;
        tmth2=3.0e0-th2;
        spth2=6.0e0+th2;
        sth4i=1.0/(6.0e0*th4);
        tth4i=2.0e0*sth4i;
        *corfac=tth4i*spth2*(3.0e0-4.0e0*cth+ctth);
        a0r=sth4i*(-42.0e0+5.0e0*th2+spth2*(8.0e0*cth-ctth));
        a0i=sth4i*(th*(-12.0e0+6.0e0*th2)+spth2*stth);
        a1r=sth4i*(14.0e0*tmth2-7.0e0*spth2*cth);
        a1i=sth4i*(30.0e0*th-5.0e0*spth2*sth);
        a2r=tth4i*(-4.0e0*tmth2+2.0e0*spth2*cth);
        a2i=tth4i*(-12.0e0*th+2.0e0*spth2*sth);
        a3r=sth4i*(2.0e0*tmth2-spth2*cth);
        a3i=sth4i*(6.0e0*th-spth2*sth);
    }
    cl=a0r*endpts[1]+a1r*endpts[2]+a2r*endpts[3]+a3r*endpts[4];
    sl=a0i*endpts[1]+a1i*endpts[2]+a2i*endpts[3]+a3i*endpts[4];
    cr=a0r*endpts[8]+a1r*endpts[7]+a2r*endpts[6]+a3r*endpts[5];
    sr = -a0i*endpts[8]-a1i*endpts[7]-a2i*endpts[6]-a3i*endpts[5];
    arg=w*(b-a);
    c=cos(arg);
    s=sin(arg);
    *corre=cl+c*cr-s*sr;
    *corim=sl+s*cr+c*sr;
}
```

Since the use of dftcor can be confusing, we also give an illustrative program dftint which uses dftcor to compute equation (13.9.1) for general a, b, ω, and $h(t)$. Several points within this program bear mentioning: The parameters M and NDFT correspond to M and N in the above discussion. On successive calls, we recompute the Fourier transform only if a or b or $h(t)$ has changed.

Since dftint is designed to work for any value of ω satisfying $\omega\Delta < \pi$, not just the special values returned by the DFT (equation 13.9.12), we do polynomial interpolation of degree MPOL on the DFT spectrum. You should be warned that a large factor of oversampling ($N \gg M$) is required for this interpolation to be accurate. After interpolation, we add the endpoint corrections from dftcor, which can be evaluated for any ω.

While dftcor is good at what it does, dftint is illustrative only. It is not a general purpose program, because it does not adapt its parameters M, NDFT, MPOL, or its interpolation scheme, to any particular function $h(t)$. You will have to experiment with your own application.

```
#include <math.h>
#include "nrutil.h"
#define M 64
#define NDFT 1024
#define MPOL 6
#define TWOPI (2.0*3.14159265)
```

The values of M, NDFT, and MPOL are merely illustrative and should be optimized for your particular application. M is the number of subintervals, NDFT is the length of the FFT (a power of 2), and MPOL is the degree of polynomial interpolation used to obtain the desired frequency from the FFT.

```
void dftint(float (*func)(float), float a, float b, float w, float *cosint,
    float *sinint)
```
Example program illustrating how to use the routine dftcor. The user supplies an external function func that returns the quantity $h(t)$. The routine then returns $\int_a^b \cos(\omega t)h(t)\,dt$ as cosint and $\int_a^b \sin(\omega t)h(t)\,dt$ as sinint.
```
{
    void dftcor(float w, float delta, float a, float b, float endpts[],
        float *corre, float *corim, float *corfac);
    void polint(float xa[], float ya[], int n, float x, float *y, float *dy);
    void realft(float data[], unsigned long n, int isign);
    static int init=0;
    int j,nn;
    static float aold = -1.e30,bold = -1.e30,delta,(*funcold)(float);
    static float data[NDFT+1],endpts[9];
    float c,cdft,cerr,corfac,corim,corre,en,s;
    float sdft,serr,*cpol,*spol,*xpol;

    cpol=vector(1,MPOL);
    spol=vector(1,MPOL);
    xpol=vector(1,MPOL);
    if (init != 1 || a != aold || b != bold || func != funcold) {
        Do we need to initialize, or is only ω changed?
        init=1;
        aold=a;
        bold=b;
        funcold=func;
        delta=(b-a)/M;
        Load the function values into the data array.
        for (j=1;j<=M+1;j++)
            data[j]=(*func)(a+(j-1)*delta);
        for (j=M+2;j<=NDFT;j++)          Zero pad the rest of the data array.
            data[j]=0.0;
        for (j=1;j<=4;j++) {                     Load the endpoints.
            endpts[j]=data[j];
            endpts[j+4]=data[M-3+j];
        }
        realft(data,NDFT,1);
        realft returns the unused value corresponding to ω_{N/2} in data[2]. We actually want
        this element to contain the imaginary part corresponding to ω_0, which is zero.
        data[2]=0.0;
    }
    Now interpolate on the DFT result for the desired frequency. If the frequency is an ω_n,
    i.e., the quantity en is an integer, then cdft=data[2*en-1], sdft=data[2*en], and you
    could omit the interpolation.
    en=w*delta*NDFT/TWOPI+1.0;
    nn=IMIN(IMAX((int)(en-0.5*MPOL+1.0),1),NDFT/2-MPOL+1);  Leftmost point for the
    for (j=1;j<=MPOL;j++,nn++) {                                 interpolation.
        cpol[j]=data[2*nn-1];
        spol[j]=data[2*nn];
        xpol[j]=nn;
    }
    polint(xpol,cpol,MPOL,en,&cdft,&cerr);
    polint(xpol,spol,MPOL,en,&sdft,&serr);
    dftcor(w,delta,a,b,endpts,&corre,&corim,&corfac);     Now get the endpoint cor-
    cdft *= corfac;                                           rection and the mul-
    sdft *= corfac;                                           tiplicative factor W(θ).
    cdft += corre;
    sdft += corim;
    c=delta*cos(w*a);                          Finally multiply by Δ and exp(iωa).
    s=delta*sin(w*a);
    *cosint=c*cdft-s*sdft;
    *sinint=s*cdft+c*sdft;
    free_vector(cpol,1,MPOL);
    free_vector(spol,1,MPOL);
    free_vector(xpol,1,MPOL);
}
```

Sometimes one is interested only in the discrete frequencies ω_m of equation (13.9.5), the ones that have integral numbers of periods in the interval $[a, b]$. For smooth $h(t)$, the value of I tends to be much smaller in magnitude at these ω's than at values in between, since the integral half-periods tend to cancel precisely. (That is why one must oversample for interpolation to be accurate: $I(\omega)$ is oscillatory with small magnitude near the ω_m's.) If you want these ω_m's without messy (and possibly inaccurate) interpolation, you have to set N to a multiple of M (compare equations 13.9.5 and 13.9.12). In the method implemented above, however, N must be at least $M + 1$, so the smallest such multiple is $2M$, resulting in a factor $\sim$2 unnecessary computing. Alternatively, one can derive a formula like equation (13.9.13), but with the last sample function $h_M = h(b)$ omitted from the DFT, but included entirely in the endpoint correction for h_M. Then one can set $M = N$ (an integer power of 2) and get the special frequencies of equation (13.9.5) with no additional overhead. The modified formula is

$$\begin{aligned} I(\omega_m) = \Delta e^{i\omega_m a}\Big\{ & W(\theta)[\mathrm{DFT}(h_0 \dots h_{M-1})]_m \\ & + \alpha_0(\theta)h_0 + \alpha_1(\theta)h_1 + \alpha_2(\theta)h_2 + \alpha_3(\theta)h_3 \\ & + e^{i\omega(b-a)}\Big[A(\theta)h_M + \overset{*}{\alpha_1}(\theta)h_{M-1} + \overset{*}{\alpha_2}(\theta)h_{M-2} + \overset{*}{\alpha_3}(\theta)h_{M-3}\Big]\Big\} \end{aligned} \tag{13.9.14}$$

where $\theta \equiv \omega_m\Delta$ and $A(\theta)$ is given by

$$A(\theta) = -\alpha_0(\theta) \tag{13.9.15}$$

for the trapezoidal case, or

$$\begin{aligned} A(\theta) &= \frac{(-6 + 11\theta^2) + (6 + \theta^2)\cos 2\theta}{6\theta^4} - i\,\mathrm{Im}[\alpha_0(\theta)] \\ &\approx \frac{1}{3} + \frac{1}{45}\theta^2 - \frac{8}{945}\theta^4 + \frac{11}{14175}\theta^6 - i\,\mathrm{Im}[\alpha_0(\theta)] \end{aligned} \tag{13.9.16}$$

for the cubic case.

Factors like $W(\theta)$ arise naturally whenever one calculates Fourier coefficients of smooth functions, and they are sometimes called attenuation factors [1]. However, the endpoint corrections are equally important in obtaining accurate values of integrals. Narasimhan and Karthikeyan [2] have given a formula that is algebraically equivalent to our trapezoidal formula. However, their formula requires the evaluation of *two* FFTs, which is unnecessary. The basic idea used here goes back at least to Filon [3] in 1928 (before the FFT!). He used Simpson's rule (quadratic interpolation). Since this interpolation is not left-right symmetric, two Fourier transforms are required. An alternative algorithm for equation (13.9.14) has been given by Lyness in [4]; for related references, see [5]. To our knowledge, the cubic-order formulas derived here have not previously appeared in the literature.

Calculating Fourier transforms when the range of integration is $(-\infty, \infty)$ can be tricky. If the function falls off reasonably quickly at infinity, you can split the integral at a large enough value of t. For example, the integration to $+\infty$ can be written

$$\begin{aligned} \int_a^\infty e^{i\omega t}h(t)\,dt &= \int_a^b e^{i\omega t}h(t)\,dt + \int_b^\infty e^{i\omega t}h(t)\,dt \\ &= \int_a^b e^{i\omega t}h(t)\,dt - \frac{h(b)e^{i\omega b}}{i\omega} + \frac{h'(b)e^{i\omega b}}{(i\omega)^2} - \cdots \end{aligned} \tag{13.9.17}$$

The splitting point b must be chosen large enough that the remaining integral over (b,∞) is small. Successive terms in its asymptotic expansion are found by integrating by parts. The integral over (a,b) can be done using `dftint`. You keep as many terms in the asymptotic expansion as you can easily compute. See [6] for some examples of this idea. More powerful methods, which work well for long-tailed functions but which do not use the FFT, are described in [7-9].

CITED REFERENCES AND FURTHER READING:

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), p. 88. [1]

Narasimhan, M.S. and Karthikeyan, M. 1984, *IEEE Transactions on Antennas & Propagation*, vol. 32, pp. 404–408. [2]

Filon, L.N.G. 1928, *Proceedings of the Royal Society of Edinburgh*, vol. 49, pp. 38–47. [3]

Giunta, G. and Murli, A. 1987, *ACM Transactions on Mathematical Software*, vol. 13, pp. 97–107. [4]

Lyness, J.N. 1987, in *Numerical Integration*, P. Keast and G. Fairweather, eds. (Dordrecht: Reidel). [5]

Pantis, G. 1975, *Journal of Computational Physics*, vol. 17, pp. 229–233. [6]

Blakemore, M., Evans, G.A., and Hyslop, J. 1976, *Journal of Computational Physics*, vol. 22, pp. 352–376. [7]

Lyness, J.N., and Kaper, T.J. 1987, *SIAM Journal on Scientific and Statistical Computing*, vol. 8, pp. 1005–1011. [8]

Thakkar, A.J., and Smith, V.H. 1975, *Computer Physics Communications*, vol. 10, pp. 73–79. [9]

13.10 Wavelet Transforms

Like the fast Fourier transform (FFT), the discrete wavelet transform (DWT) is a fast, linear operation that operates on a data vector whose length is an integer power of two, transforming it into a numerically different vector of the same length. Also like the FFT, the wavelet transform is invertible and in fact orthogonal — the inverse transform, when viewed as a big matrix, is simply the transpose of the transform. Both FFT and DWT, therefore, can be viewed as a rotation in function space, from the input space (or time) domain, where the basis functions are the unit vectors $\mathbf{e}_i$, or Dirac delta functions in the continuum limit, to a different domain. For the FFT, this new domain has basis functions that are the familiar sines and cosines. In the wavelet domain, the basis functions are somewhat more complicated and have the fanciful names "mother functions" and "wavelets."

Of course there are an infinity of possible bases for function space, almost all of them uninteresting! What makes the wavelet basis interesting is that, *unlike* sines and cosines, individual wavelet functions are quite localized in space; simultaneously, *like* sines and cosines, individual wavelet functions are quite localized in frequency or (more precisely) characteristic scale. As we will see below, the particular kind of dual localization achieved by wavelets renders large classes of functions and operators sparse, or sparse to some high accuracy, when transformed into the wavelet domain. Analogously with the Fourier domain, where a class of computations, like convolutions, become computationally fast, there is a large class of computations

— those that can take advantage of sparsity — that become computationally fast in the wavelet domain [1].

Unlike sines and cosines, which define a unique Fourier transform, there is not one single unique set of wavelets; in fact, there are infinitely many possible sets. Roughly, the different sets of wavelets make different trade-offs between how compactly they are localized in space and how smooth they are. (There are further fine distinctions.)

Daubechies Wavelet Filter Coefficients

A particular set of wavelets is specified by a particular set of numbers, called *wavelet filter coefficients*. Here, we will largely restrict ourselves to wavelet filters in a class discovered by Daubechies [2]. This class includes members ranging from highly localized to highly smooth. The simplest (and most localized) member, often called *DAUB4*, has only four coefficients, $c_0, \ldots, c_3$. For the moment we specialize to this case for ease of notation.

Consider the following transformation matrix acting on a column vector of data to its right:

$$\begin{bmatrix}
c_0 & c_1 & c_2 & c_3 & & & & & & & \\
c_3 & -c_2 & c_1 & -c_0 & & & & & & & \\
 & & c_0 & c_1 & c_2 & c_3 & & & & & \\
 & & c_3 & -c_2 & c_1 & -c_0 & & & & & \\
\vdots & \vdots & & & & & \ddots & & & & \\
 & & & & & & & c_0 & c_1 & c_2 & c_3 \\
 & & & & & & & c_3 & -c_2 & c_1 & -c_0 \\
c_2 & c_3 & & & & & & & & c_0 & c_1 \\
c_1 & -c_0 & & & & & & & & c_3 & -c_2
\end{bmatrix} \tag{13.10.1}$$

Here blank entries signify zeroes. Note the structure of this matrix. The first row generates one component of the data convolved with the filter coefficients $c_0 \ldots, c_3$. Likewise the third, fifth, and other odd rows. If the even rows followed this pattern, offset by one, then the matrix would be a circulant, that is, an ordinary convolution that could be done by FFT methods. (Note how the last two rows wrap around like convolutions with periodic boundary conditions.) Instead of convolving with $c_0, \ldots, c_3$, however, the even rows perform a different convolution, with coefficients $c_3, -c_2, c_1, -c_0$. The action of the matrix, overall, is thus to perform two related convolutions, then to decimate each of them by half (throw away half the values), and interleave the remaining halves.

It is useful to think of the filter $c_0, \ldots, c_3$ as being a smoothing filter, call it H, something like a moving average of four points. Then, because of the minus signs, the filter $c_3, -c_2, c_1, -c_0$, call it G, is *not* a smoothing filter. (In signal processing contexts, H and G are called *quadrature mirror filters* [3].) In fact, the c's are chosen so as to make G yield, insofar as possible, a *zero* response to a sufficiently smooth data vector. This is done by requiring the sequence $c_3, -c_2, c_1, -c_0$ to have a certain number of vanishing moments. When this is the case for p moments (starting with the zeroth), a set of wavelets is said to satisfy an "approximation condition of order

p." This results in the output of H, decimated by half, accurately representing the data's "smooth" information. The output of G, also decimated, is referred to as the data's "detail" information [4].

For such a characterization to be useful, it must be possible to reconstruct the original data vector of length N from its $N/2$ smooth or s-components and its $N/2$ detail or d-components. That is effected by requiring the matrix (13.10.1) to be orthogonal, so that its inverse is just the transposed matrix

$$\begin{bmatrix} c_0 & c_3 & & & \cdots & & & & c_2 & c_1 \\ c_1 & -c_2 & & & \cdots & & & & c_3 & -c_0 \\ c_2 & c_1 & c_0 & c_3 & & & & & & \\ c_3 & -c_0 & c_1 & -c_2 & & & & & & \\ & & & & \ddots & & & & & \\ & & & & & c_2 & c_1 & c_0 & c_3 & \\ & & & & & c_3 & -c_0 & c_1 & -c_2 & \\ & & & & & & & c_2 & c_1 & c_0 & c_3 \\ & & & & & & & c_3 & -c_0 & c_1 & -c_2 \end{bmatrix} \tag{13.10.2}$$

One sees immediately that matrix (13.10.2) is inverse to matrix (13.10.1) if and only if these two equations hold,

$$\begin{aligned} c_0^2 + c_1^2 + c_2^2 + c_3^2 &= 1 \\ c_2c_0 + c_3c_1 &= 0 \end{aligned} \tag{13.10.3}$$

If additionally we require the approximation condition of order $p = 2$, then two additional relations are required,

$$\begin{aligned} c_3 - c_2 + c_1 - c_0 &= 0 \\ 0c_3 - 1c_2 + 2c_1 - 3c_0 &= 0 \end{aligned} \tag{13.10.4}$$

Equations (13.10.3) and (13.10.4) are 4 equations for the 4 unknowns $c_0, \ldots, c_3$, first recognized and solved by Daubechies. The unique solution (up to a left-right reversal) is

$$\begin{aligned} c_0 &= (1+\sqrt{3})/4\sqrt{2} \qquad & c_1 &= (3+\sqrt{3})/4\sqrt{2} \\ c_2 &= (3-\sqrt{3})/4\sqrt{2} \qquad & c_3 &= (1-\sqrt{3})/4\sqrt{2} \end{aligned} \tag{13.10.5}$$

In fact, DAUB4 is only the most compact of a sequence of wavelet sets: If we had six coefficients instead of four, there would be three orthogonality requirements in equation (13.10.3) (with offsets of zero, two, and four), and we could require the vanishing of $p = 3$ moments in equation (13.10.4). In this case, DAUB6, the solution coefficients can also be expressed in closed form,

$$\begin{aligned} c_0 &= (1+\sqrt{10}+\sqrt{5+2\sqrt{10}})/16\sqrt{2} \qquad & c_1 &= (5+\sqrt{10}+3\sqrt{5+2\sqrt{10}})/16\sqrt{2} \\ c_2 &= (10-2\sqrt{10}+2\sqrt{5+2\sqrt{10}})/16\sqrt{2} \qquad & c_3 &= (10-2\sqrt{10}-2\sqrt{5+2\sqrt{10}})/16\sqrt{2} \\ c_4 &= (5+\sqrt{10}-3\sqrt{5+2\sqrt{10}})/16\sqrt{2} \qquad & c_5 &= (1+\sqrt{10}-\sqrt{5+2\sqrt{10}})/16\sqrt{2} \end{aligned} \tag{13.10.6}$$

For higher p, up to 10, Daubechies [2] has tabulated the coefficients numerically. The number of coefficients increases by two each time p is increased by one.

Discrete Wavelet Transform

We have not yet defined the discrete wavelet transform (DWT), but we are almost there: The DWT consists of applying a wavelet coefficient matrix like (13.10.1) *hierarchically*, first to the full data vector of length N, then to the "smooth" vector of length $N/2$, then to the "smooth-smooth" vector of length $N/4$, and so on until only a trivial number of "smooth-...-smooth" components (usually 2) remain. The procedure is sometimes called a *pyramidal algorithm* [4], for obvious reasons. The output of the DWT consists of these remaining components and all the "detail" components that were accumulated along the way. A diagram should make the procedure clear:

$$
\begin{bmatrix} y_1\\ y_2\\ y_3\\ y_4\\ y_5\\ y_6\\ y_7\\ y_8\\ y_9\\ y_{10}\\ y_{11}\\ y_{12}\\ y_{13}\\ y_{14}\\ y_{15}\\ y_{16} \end{bmatrix}
\xrightarrow{13.10.1}
\begin{bmatrix} s_1\\ d_1\\ s_2\\ d_2\\ s_3\\ d_3\\ s_4\\ d_4\\ s_5\\ d_5\\ s_6\\ d_6\\ s_7\\ d_7\\ s_8\\ d_8 \end{bmatrix}
\xrightarrow{\text{permute}}
\begin{bmatrix} s_1\\ s_2\\ s_3\\ s_4\\ s_5\\ s_6\\ s_7\\ \underline{s_8}\\ d_1\\ d_2\\ d_3\\ d_4\\ d_5\\ d_6\\ d_7\\ d_8 \end{bmatrix}
\xrightarrow{13.10.1}
\begin{bmatrix} S_1\\ D_1\\ S_2\\ D_2\\ S_3\\ D_3\\ S_4\\ \underline{D_4}\\ d_1\\ d_2\\ d_3\\ d_4\\ d_5\\ d_6\\ d_7\\ d_8 \end{bmatrix}
\xrightarrow{\text{permute}}
\begin{bmatrix} S_1\\ S_2\\ S_3\\ \underline{S_4}\\ D_1\\ D_2\\ D_3\\ \underline{D_4}\\ d_1\\ d_2\\ d_3\\ d_4\\ d_5\\ d_6\\ d_7\\ d_8 \end{bmatrix}
\xrightarrow{\text{etc.}}
\begin{bmatrix} \mathcal{S}_1\\ \underline{\mathcal{S}_2}\\ \mathcal{D}_1\\ \underline{\mathcal{D}_2}\\ D_1\\ D_2\\ D_3\\ \underline{D_4}\\ d_1\\ d_2\\ d_3\\ d_4\\ d_5\\ d_6\\ d_7\\ d_8 \end{bmatrix}
\tag{13.10.7}
$$

If the length of the data vector were a higher power of two, there would be more stages of applying (13.10.1) (or any other wavelet coefficients) and permuting. The endpoint will always be a vector with two $\mathcal{S}$'s and a hierarchy of $\mathcal{D}$'s, D's, d's, etc. Notice that once d's are generated, they simply propagate through to all subsequent stages.

A value d_i of any level is termed a "wavelet coefficient" of the original data vector; the final values $\mathcal{S}_1, \mathcal{S}_2$ should strictly be called "mother-function coefficients," although the term "wavelet coefficients" is often used loosely for both d's and final $\mathcal{S}$'s. Since the full procedure is a composition of orthogonal linear operations, the whole DWT is itself an orthogonal linear operator.

To invert the DWT, one simply reverses the procedure, starting with the smallest level of the hierarchy and working (in equation 13.10.7) from right to left. The inverse matrix (13.10.2) is of course used instead of the matrix (13.10.1).

As already noted, the matrices (13.10.1) and (13.10.2) embody periodic ("wrap-around") boundary conditions on the data vector. One normally accepts this as a minor inconvenience: the last few wavelet coefficients at each level of the hierarchy are affected by data from both ends of the data vector. By circularly shifting the matrix (13.10.1) $N/2$ columns to the left, one can symmetrize the wrap-around; but this does not eliminate it. It is in fact possible to eliminate the wrap-around completely by altering the coefficients in the first and last N rows of (13.10.1), giving an orthogonal matrix that is purely band-diagonal [5]. This variant, beyond our scope here, is useful when, e.g., the data varies by many orders of magnitude from one end of the data vector to the other.

Here is a routine, `wt1`, that performs the pyramidal algorithm (or its inverse if `isign` is negative) on some data vector `a[1..n]`. Successive applications of the wavelet filter, and accompanying permutations, are done by an assumed routine `wtstep`, which must be provided. (We give examples of several different `wtstep` routines just below.)

```
void wt1(float a[], unsigned long n, int isign,
    void (*wtstep)(float [], unsigned long, int))
One-dimensional discrete wavelet transform. This routine implements the pyramid algorithm,
replacing a[1..n] by its wavelet transform (for isign=1), or performing the inverse operation
(for isign=-1). Note that n MUST be an integer power of 2. The routine wtstep, whose
actual name must be supplied in calling this routine, is the underlying wavelet filter. Examples
of wtstep are daub4 and (preceded by pwtset) pwt.
{
    unsigned long nn;

    if (n < 4) return;
    if (isign >= 0) {                               Wavelet transform.
        for (nn=n;nn>=4;nn>>=1) (*wtstep)(a,nn,isign);
        Start at largest hierarchy, and work towards smallest.
    } else {                                        Inverse wavelet transform.
        for (nn=4;nn<=n;nn<<=1) (*wtstep)(a,nn,isign);
        Start at smallest hierarchy, and work towards largest.
    }
}
```

Here, as a specific instance of `wtstep`, is a routine for the DAUB4 wavelets:

```
#include "nrutil.h"
#define C0 0.4829629131445341
#define C1 0.8365163037378079
#define C2 0.2241438680420134
#define C3 -0.1294095225512604

void daub4(float a[], unsigned long n, int isign)
Applies the Daubechies 4-coefficient wavelet filter to data vector a[1..n] (for isign=1) or
applies its transpose (for isign=-1). Used hierarchically by routines wt1 and wtn.
{
    float *wksp;
    unsigned long nh,nh1,i,j;

    if (n < 4) return;
    wksp=vector(1,n);
    nh1=(nh=n >> 1)+1;
    if (isign >= 0) {                               Apply filter.
        for (i=1,j=1;j<=n-3;j+=2,i++) {
            wksp[i]=C0*a[j]+C1*a[j+1]+C2*a[j+2]+C3*a[j+3];
            wksp[i+nh] = C3*a[j]-C2*a[j+1]+C1*a[j+2]-C0*a[j+3];
        }
        wksp[i]=C0*a[n-1]+C1*a[n]+C2*a[1]+C3*a[2];
        wksp[i+nh] = C3*a[n-1]-C2*a[n]+C1*a[1]-C0*a[2];
    } else {                                        Apply transpose filter.
        wksp[1]=C2*a[nh]+C1*a[n]+C0*a[1]+C3*a[nh1];
        wksp[2] = C3*a[nh]-C0*a[n]+C1*a[1]-C2*a[nh1];
        for (i=1,j=3;i<nh;i++) {
            wksp[j++]=C2*a[i]+C1*a[i+nh]+C0*a[i+1]+C3*a[i+nh1];
            wksp[j++] = C3*a[i]-C0*a[i+nh]+C1*a[i+1]-C2*a[i+nh1];
        }
    }
    for (i=1;i<=n;i++) a[i]=wksp[i];
    free_vector(wksp,1,n);
}
```

For larger sets of wavelet coefficients, the wrap-around of the last rows or columns is a programming inconvenience. An efficient implementation would handle the wrap-arounds as special cases, outside of the main loop. Here, we will content ourselves with a more general scheme involving some extra arithmetic at run time. The following routine sets up any particular wavelet coefficients whose values you happen to know.

```
typedef struct {
    int ncof,ioff,joff;
    float *cc,*cr;
} wavefilt;

wavefilt wfilt;                    Defining declaration of a structure.

void pwtset(int n)
```
Initializing routine for pwt, here implementing the Daubechies wavelet filters with 4, 12, and 20 coefficients, as selected by the input value n. Further wavelet filters can be included in the obvious manner. This routine must be called (once) before the first use of pwt. (For the case n=4, the specific routine daub4 is considerably faster than pwt.)
```
{
    void nrerror(char error_text[]);
    int k;
    float sig = -1.0;
    static float c4[5]={0.0,0.4829629131445341,0.8365163037378079,
            0.2241438680420134,-0.1294095225512604};
    static float c12[13]={0.0,0.111540743350, 0.494623890398, 0.751133908021,
        0.315250351709,-0.226264693965,-0.129766867567,
        0.097501605587, 0.027522865530,-0.031582039318,
        0.000553842201, 0.004777257511,-0.001077301085};
    static float c20[21]={0.0,0.026670057901, 0.188176800078, 0.527201188932,
        0.688459039454, 0.281172343661,-0.249846424327,
        -0.195946274377, 0.127369340336, 0.093057364604,
        -0.071394147166,-0.029457536822, 0.033212674059,
        0.003606553567,-0.010733175483, 0.001395351747,
        0.001992405295,-0.000685856695,-0.000116466855,
        0.000093588670,-0.000013264203};
    static float c4r[5],c12r[13],c20r[21];

    wfilt.ncof=n;
    if (n == 4) {
        wfilt.cc=c4;
        wfilt.cr=c4r;
    }
    else if (n == 12) {
        wfilt.cc=c12;
        wfilt.cr=c12r;
    }
    else if (n == 20) {
        wfilt.cc=c20;
        wfilt.cr=c20r;
    }
    else nrerror("unimplemented value n in pwtset");
    for (k=1;k<=n;k++) {
        wfilt.cr[wfilt.ncof+1-k]=sig*wfilt.cc[k];
        sig = -sig;
    }
    wfilt.ioff = wfilt.joff = -(n >> 1);
    These values center the "support" of the wavelets at each level. Alternatively, the "peaks"
    of the wavelets can be approximately centered by the choices ioff=-2 and joff=-n+2.
    Note that daub4 and pwtset with n=4 use different default centerings.
}
```

Once pwtset has been called, the following routine can be used as a specific instance of wtstep.

```
#include "nrutil.h"

typedef struct {
    int ncof,ioff,joff;
    float *cc,*cr;
} wavefilt;

extern wavefilt wfilt;                          Defined in pwtset.

void pwt(float a[], unsigned long n, int isign)
Partial wavelet transform: applies an arbitrary wavelet filter to data vector a[1..n] (for isign =
1) or applies its transpose (for isign = −1). Used hierarchically by routines wt1 and wtn.
The actual filter is determined by a preceding (and required) call to pwtset, which initializes
the structure wfilt.
{
    float ai,ai1,*wksp;
    unsigned long i,ii,j,jf,jr,k,n1,ni,nj,nh,nmod;

    if (n < 4) return;
    wksp=vector(1,n);
    nmod=wfilt.ncof*n;                          A positive constant equal to zero mod n.
    n1=n-1;                                     Mask of all bits, since n a power of 2.
    nh=n >> 1;
    for (j=1;j<=n;j++) wksp[j]=0.0;
    if (isign >= 0) {                           Apply filter.
        for (ii=1,i=1;i<=n;i+=2,ii++) {
            ni=i+nmod+wfilt.ioff;               Pointer to be incremented and wrapped-around.
            nj=i+nmod+wfilt.joff;
            for (k=1;k<=wfilt.ncof;k++) {
                jf=n1 & (ni+k);                 We use bitwise and to wrap-around the point-
                jr=n1 & (nj+k);                     ers.
                wksp[ii] += wfilt.cc[k]*a[jf+1];
                wksp[ii+nh] += wfilt.cr[k]*a[jr+1];
            }
        }
    } else {                                    Apply transpose filter.
        for (ii=1,i=1;i<=n;i+=2,ii++) {
            ai=a[ii];
            ai1=a[ii+nh];
            ni=i+nmod+wfilt.ioff;               See comments above.
            nj=i+nmod+wfilt.joff;
            for (k=1;k<=wfilt.ncof;k++) {
                jf=(n1 & (ni+k))+1;
                jr=(n1 & (nj+k))+1;
                wksp[jf] += wfilt.cc[k]*ai;
                wksp[jr] += wfilt.cr[k]*ai1;
            }
        }
    }
    for (j=1;j<=n;j++) a[j]=wksp[j];            Copy the results back from workspace.
    free_vector(wksp,1,n);
}
```

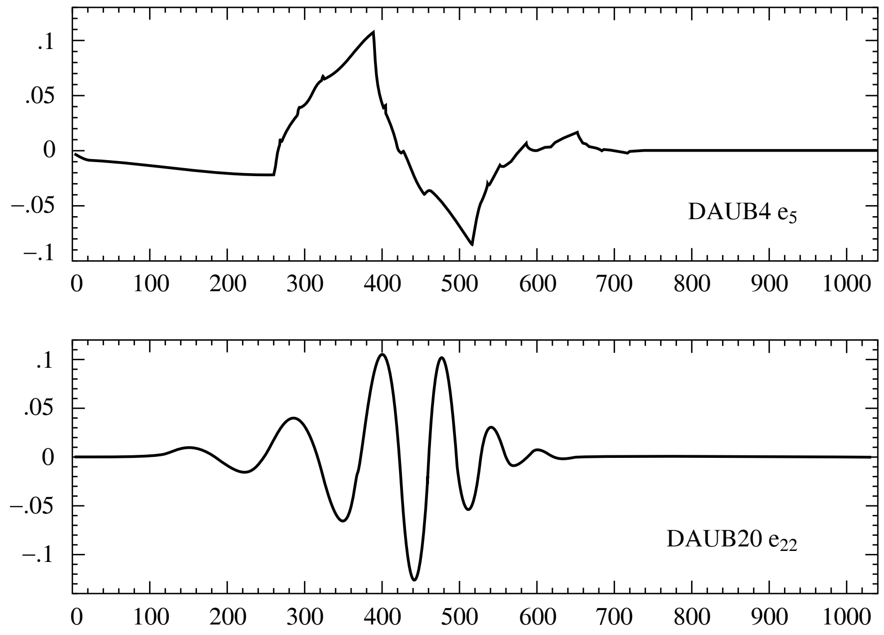


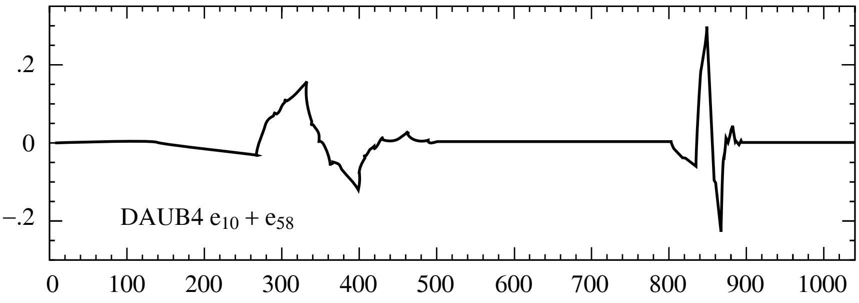
Figure 13.10.1. Wavelet functions, that is, single basis functions from the wavelet families DAUB4 and DAUB20. A complete, orthonormal wavelet basis consists of scalings and translations of either one of these functions. DAUB4 has an infinite number of cusps; DAUB20 would show similar behavior in a higher derivative.

What Do Wavelets Look Like?

We are now in a position actually to see some wavelets. To do so, we simply run unit vectors through any of the above discrete wavelet transforms, with `isign` negative so that the inverse transform is performed. Figure 13.10.1 shows the DAUB4 wavelet that is the inverse DWT of a unit vector in the 5th component of a vector of length 1024, and also the DAUB20 wavelet that is the inverse of the 22nd component. (One needs to go to a later hierarchical level for DAUB20, to avoid a wavelet with a wrapped-around tail.) Other unit vectors would give wavelets with the same shapes, but different positions and scales.

One sees that both DAUB4 and DAUB20 have wavelets that are continuous. DAUB20 wavelets also have higher continuous derivatives. DAUB4 has the peculiar property that its derivative exists only *almost* everywhere. Examples of where it fails to exist are the points $p/2^n$, where p and n are integers; at such points, DAUB4 is left differentiable, but not right differentiable! This kind of discontinuity — at least in some derivative — is a necessary feature of wavelets with compact support, like the Daubechies series. For every increase in the number of wavelet coefficients by two, the Daubechies wavelets gain about *half* a derivative of continuity. (But not exactly half; the actual orders of regularity are irrational numbers!)

Note that the fact that wavelets are not smooth does not prevent their having exact representations for some smooth functions, as demanded by their approximation order p. The continuity of a wavelet is not the same as the continuity of functions



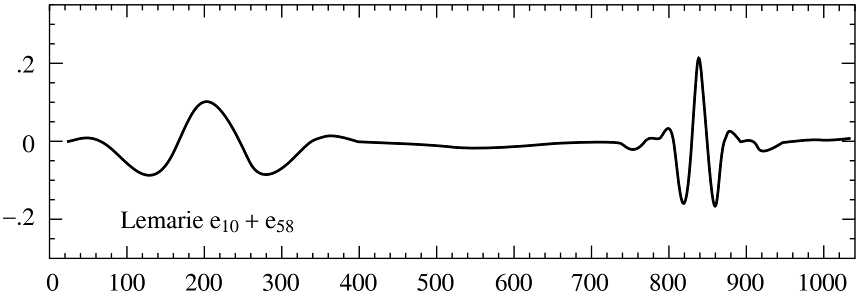


Figure 13.10.2. More wavelets, here generated from the sum of two unit vectors, $\mathbf{e}_{10} + \mathbf{e}_{58}$, which are in different hierarchical levels of scale, and also at different spatial positions. DAUB4 wavelets (a) are defined by a filter in coordinate space (equation 13.10.5), while Lemarie wavelets (b) are defined by a filter most easily written in Fourier space (equation 13.10.14).

that a set of wavelets can represent. For example, DAUB4 can represent (piecewise) linear functions of arbitrary slope: in the correct linear combinations, the cusps all cancel out. Every increase of two in the number of coefficients allows one higher order of polynomial to be exactly represented.

Figure 13.10.2 shows the result of performing the inverse DWT on the input vector $\mathbf{e}_{10} + \mathbf{e}_{58}$, again for the two different particular wavelets. Since 10 lies early in the hierarchical range of $9 - 16$, that wavelet lies on the left side of the picture. Since 58 lies in a later (smaller-scale) hierarchy, it is a narrower wavelet; in the range of 33–64 it is towards the end, so it lies on the right side of the picture. Note that smaller-scale wavelets are taller, so as to have the same squared integral.

Wavelet Filters in the Fourier Domain

The Fourier transform of a set of filter coefficients c_j is given by

$$H(\omega) = \sum_j c_j e^{ij\omega} \tag{13.10.8}$$

Here H is a function periodic in 2π, and it has the same meaning as before: It is the wavelet filter, now written in the Fourier domain. A very useful fact is that the orthogonality conditions for the c's (e.g., equation 13.10.3 above) collapse to two simple relations in the Fourier domain,

$$\frac{1}{2}|H(0)|^2 = 1 \tag{13.10.9}$$

and

$$\frac{1}{2}\left[|H(\omega)|^2 + |H(\omega+\pi)|^2\right] = 1 \qquad (13.10.10)$$

Likewise the approximation condition of order p (e.g., equation 13.10.4 above) has a simple formulation, requiring that $H(\omega)$ have a pth order zero at $\omega = \pi$, or (equivalently)

$$H^{(m)}(\pi) = 0 \qquad m = 0, 1, \dots, p-1 \qquad (13.10.11)$$

It is thus relatively straightforward to invent wavelet sets in the Fourier domain. You simply invent a function $H(\omega)$ satisfying equations (13.10.9)–(13.10.11). To find the actual c_j's applicable to a data (or s-component) vector of length N, and with periodic wrap-around as in matrices (13.10.1) and (13.10.2), you invert equation (13.10.8) by the discrete Fourier transform

$$c_j = \frac{1}{N}\sum_{k=0}^{N-1} H\left(\frac{2\pi k}{N}\right) e^{-2\pi ijk/N} \qquad (13.10.12)$$

The quadrature mirror filter G (reversed c_j's with alternating signs), incidentally, has the Fourier representation

$$G(\omega) = e^{-i\omega} H^*(\omega+\pi) \qquad (13.10.13)$$

where asterisk denotes complex conjugation.

In general the above procedure will *not* produce wavelet filters with compact support. In other words, all N of the c_j's, $j = 0, \dots, N-1$ will in general be nonzero (though they may be rapidly decreasing in magnitude). The Daubechies wavelets, or other wavelets with compact support, are specially chosen so that $H(\omega)$ is a trigonometric polynomial with only a small number of Fourier components, guaranteeing that there will be only a small number of nonzero c_j's.

On the other hand, there is sometimes no particular reason to demand compact support. Giving it up in fact allows the ready construction of relatively smoother wavelets (higher values of p). Even without compact support, the convolutions implicit in the matrix (13.10.1) can be done efficiently by FFT methods.

Lemarie's wavelet (see [4]) has $p = 4$, does not have compact support, and is defined by the choice of $H(\omega)$,

$$H(\omega) = \left[2(1-u)^4 \frac{315 - 420u + 126u^2 - 4u^3}{315 - 420v + 126v^2 - 4v^3}\right]^{1/2} \qquad (13.10.14)$$

where

$$u \equiv \sin^2\frac{\omega}{2} \qquad v \equiv \sin^2\omega \qquad (13.10.15)$$

It is beyond our scope to explain where equation (13.10.14) comes from. An informal description is that the quadrature mirror filter $G(\omega)$ deriving from equation (13.10.14) has the property that it gives identically zero when applied to any function whose odd-numbered samples are equal to the cubic spline interpolation of its even-numbered samples. Since this class of functions includes many very smooth members, it follows that $H(\omega)$ does a good job of truly selecting a function's smooth information content. Sample Lemarie wavelets are shown in Figure 13.10.2.

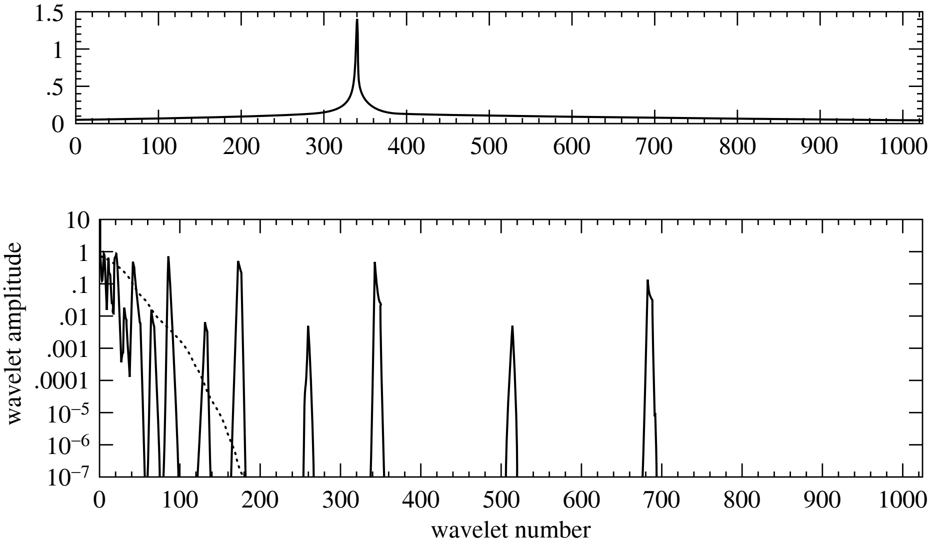


Figure 13.10.3. (a) Arbitrary test function, with cusp, sampled on a vector of length 1024. (b) Absolute value of the 1024 wavelet coefficients produced by the discrete wavelet transform of (a). Note log scale. The dotted curve plots the same amplitudes when sorted by decreasing size. One sees that only 130 out of 1024 coefficients are larger than 10^{-4} (or larger than about 10^{-5} times the largest coefficient, whose value is $\sim$ 10).

Truncated Wavelet Approximations

Most of the usefulness of wavelets rests on the fact that wavelet transforms can usefully be severely truncated, that is, turned into sparse expansions. The case of Fourier transforms is different: FFTs are ordinarily used without truncation, to compute fast convolutions, for example. This works because the convolution operator is particularly simple in the Fourier basis. There are not, however, any standard mathematical operations that are especially simple in the wavelet basis.

To see how truncation works, consider the simple example shown in Figure 13.10.3. The upper panel shows an arbitrarily chosen test function, smooth except for a square-root cusp, sampled onto a vector of length 2^{10}. The bottom panel (solid curve) shows, on a log scale, the absolute value of the vector's components after it has been run through the DAUB4 discrete wavelet transform. One notes, from right to left, the different levels of hierarchy, 513–1024, 257–512, 129–256, etc. Within each level, the wavelet coefficients are non-negligible only very near the location of the cusp, or very near the left and right boundaries of the hierarchical range (edge effects).

The dotted curve in the lower panel of Figure 13.10.3 plots the same amplitudes as the solid curve, but sorted into decreasing order of size. One can read off, for example, that the 130th largest wavelet coefficient has an amplitude less than 10^{-5} of the largest coefficient, whose magnitude is $\sim$ 10 (power or square integral ratio less than 10^{-10}). Thus, the example function can be represented quite accurately by only 130, rather than 1024, coefficients — the remaining ones being set to zero. Note that this kind of truncation makes the vector sparse, but not shorter than 1024. It is very important that vectors in wavelet space be truncated according to the *amplitude* of the components, not their position in the vector. Keeping the first 256 components of the vector (all levels of the hierarchy except the last two) would give an extremely poor, and jagged, approximation to the function. When you compress a function with wavelets, you have to record both the values *and the positions* of the nonzero coefficients.

Generally, compact (and therefore unsmooth) wavelets are better for lower accuracy approximation and for functions with discontinuities (like edges), while smooth (and therefore noncompact) wavelets are better for achieving high numerical accuracy. This makes compact wavelets a good choice for image compression, for example, while it makes smooth wavelets best for fast solution of integral equations.

Wavelet Transform in Multidimensions

A wavelet transform of a d-dimensional array is most easily obtained by transforming the array sequentially on its first index (for all values of its other indices), then on its second, and so on. Each transformation corresponds to multiplication by an orthogonal matrix. By matrix associativity, the result is independent of the order in which the indices were transformed. The situation is exactly like that for multidimensional FFTs. A routine for effecting the multidimensional DWT can thus be modeled on a multidimensional FFT routine like `fourn`:

```
#include "nrutil.h"

void wtn(float a[], unsigned long nn[], int ndim, int isign,
    void (*wtstep)(float [], unsigned long, int))
```
Replaces `a` by its `ndim`-dimensional discrete wavelet transform, if `isign` is input as 1. Here `nn[1..ndim]` is an integer array containing the lengths of each dimension (number of real values), which MUST all be powers of 2. `a` is a real array of length equal to the product of these lengths, in which the data are stored as in a multidimensional real array. If `isign` is input as -1, `a` is replaced by its inverse wavelet transform. The routine `wtstep`, whose actual name must be supplied in calling this routine, is the underlying wavelet filter. Examples of `wtstep` are `daub4` and (preceded by `pwtset`) `pwt`.
```
{
    unsigned long i1,i2,i3,k,n,nnew,nprev=1,nt,ntot=1;
    int idim;
    float *wksp;

    for (idim=1;idim<=ndim;idim++) ntot *= nn[idim];
    wksp=vector(1,ntot);
    for (idim=1;idim<=ndim;idim++) {          Main loop over the dimensions.
        n=nn[idim];
        nnew=n*nprev;
        if (n > 4) {
            for (i2=0;i2<ntot;i2+=nnew) {
                for (i1=1;i1<=nprev;i1++) {
                    for (i3=i1+i2,k=1;k<=n;k++,i3+=nprev) wksp[k]=a[i3];
                    Copy the relevant row or column or etc. into workspace.
                    if (isign >= 0) {          Do one-dimensional wavelet transform.
                        for(nt=n;nt>=4;nt >>= 1)
                            (*wtstep)(wksp,nt,isign);
                    } else {                   Or inverse transform.
                        for(nt=4;nt<=n;nt <<= 1)
                            (*wtstep)(wksp,nt,isign);
                    }

                    for (i3=i1+i2,k=1;k<=n;k++,i3+=nprev) a[i3]=wksp[k];
                    Copy back from workspace.
                }
            }
        }
        nprev=nnew;
    }
    free_vector(wksp,1,ntot);
}
```

Here, as before, wtstep is an individual wavelet step, either daub4 or pwt.

Compression of Images

An immediate application of the multidimensional transform wtn is to image compression. The overall procedure is to take the wavelet transform of a digitized image, and then to "allocate bits" among the wavelet coefficients in some highly nonuniform, optimized, manner. In general, large wavelet coefficients get quantized accurately, while small coefficients are quantized coarsely with only a bit or two — or else are truncated completely. If the resulting quantization levels are still statistically nonuniform, they may then be further compressed by a technique like Huffman coding (§20.4).

While a more detailed description of the "back end" of this process, namely the quantization and coding of the image, is beyond our scope, it is quite straightforward to demonstrate the "front-end" wavelet encoding with a simple truncation: We keep (with full accuracy) all wavelet coefficients larger than some threshold, and we delete (set to zero) all smaller wavelet coefficients. We can then adjust the threshold to vary the fraction of preserved coefficients.

Figure 13.10.4 shows a sequence of images that differ in the number of wavelet coefficients that have been kept. The original picture (a), which is an official IEEE test image, has 256 by 256 pixels with an 8-bit grayscale. The two reproductions following are reconstructed with 23% (b) and 5.5% (c) of the 65536 wavelet coefficients. The latter image illustrates the kind of compromises made by the truncated wavelet representation. High-contrast edges (the model's right cheek and hair highlights, e.g.) are maintained at a relatively high resolution, while low-contrast areas (the model's left eye and cheek, e.g.) are washed out into what amounts to large constant pixels. Figure 13.10.4 (d) is the result of performing the identical procedure with Fourier, instead of wavelet, transforms: The figure is reconstructed from the 5.5% of 65536 real Fourier components having the largest magnitudes. One sees that, since sines and cosines are nonlocal, the resolution is uniformly poor across the picture; also, the deletion of any components produces a mottled "ringing" everywhere. (Practical Fourier image compression schemes therefore break up an image into small blocks of pixels, 16×16, say, and do rather elaborate smoothing across block boundaries when the image is reconstructed.)

Fast Solution of Linear Systems

One of the most interesting, and promising, wavelet applications is linear algebra. The basic idea [1] is to think of an integral operator (that is, a large matrix) as a digital image. Suppose that the operator compresses well under a two-dimensional wavelet transform, i.e., that a large fraction of its wavelet coefficients are so small as to be negligible. Then any linear system involving the operator becomes a sparse system in the wavelet basis. In other words, to solve

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{13.10.16}$$

Figure 13.10.4. (a) IEEE test image, 256×256 pixels with 8-bit grayscale. (b) The image is transformed into the wavelet basis; 77% of its wavelet components are set to zero (those of smallest magnitude); it is then reconstructed from the remaining 23%. (c) Same as (b), but 94.5% of the wavelet components are deleted. (d) Same as (c), but the Fourier transform is used instead of the wavelet transform. Wavelet coefficients are better than the Fourier coefficients at preserving relevant details.

we first wavelet-transform the operator **A** and the right-hand side **b** by

$$\widetilde{\mathbf{A}} \equiv \mathbf{W} \cdot \mathbf{A} \cdot \mathbf{W}^T, \qquad \widetilde{\mathbf{b}} \equiv \mathbf{W} \cdot \mathbf{b} \tag{13.10.17}$$

where **W** represents the one-dimensional wavelet transform, then solve

$$\widetilde{\mathbf{A}} \cdot \widetilde{\mathbf{x}} = \widetilde{\mathbf{b}} \tag{13.10.18}$$

and finally transform to the answer by the inverse wavelet transform

$$\mathbf{x} = \mathbf{W}^T \cdot \widetilde{\mathbf{x}} \tag{13.10.19}$$

(Note that the routine `wtn` does the complete transformation of **A** into $\widetilde{\mathbf{A}}$.)

Figure 13.10.5. Wavelet transform of a 256×256 matrix, represented graphically. The original matrix has a discontinuous cusp along its diagonal, decaying smoothly away on both sides of the diagonal. In wavelet basis, the matrix becomes sparse: Components larger than 10^{-3} are shown as black, components larger than 10^{-6} as gray, and smaller-magnitude components are white. The matrix indices i and j number from the lower left.

A typical integral operator that compresses well into wavelets has arbitrary (or even nearly singular) elements near to its main diagonal, but becomes smooth away from the diagonal. An example might be

$$A_{ij} = \begin{cases} -1 & \text{if } i = j \\ |i-j|^{-1/2} & \text{otherwise} \end{cases} \tag{13.10.20}$$

Figure 13.10.5 shows a graphical representation of the wavelet transform of this matrix, where i and j range over $1 \ldots 256$, using the DAUB12 wavelets. Elements larger in magnitude than 10^{-3} times the maximum element are shown as black pixels, while elements between 10^{-3} and 10^{-6} are shown in gray. White pixels are $< 10^{-6}$. The indices i and j each number from the lower left.

In the figure, one sees the hierarchical decomposition into power-of-two sized blocks. At the edges or corners of the various blocks, one sees edge effects caused by the wrap-around wavelet boundary conditions. Apart from edge effects, within each block, the nonnegligible elements are concentrated along the block diagonals. This is a statement that, for this type of linear operator, a wavelet is coupled mainly to near neighbors in its own hierarchy (square blocks along the main diagonal) and near neighbors in other hierarchies (rectangular blocks off the diagonal).

The number of nonnegligible elements in a matrix like that in Figure 13.10.5 scales only as N, the linear size of the matrix; as a rough rule of thumb it is about $10N \log_{10}(1/\epsilon)$, where ϵ is the truncation level, e.g., 10^{-6}. For a 2000 by 2000 matrix, then, the matrix is sparse by a factor on the order of 30.

Various numerical schemes can be used to solve sparse linear systems of this "hierarchically band diagonal" form. Beylkin, Coifman, and Rokhlin [1] make the interesting observations that (1) the product of two such matrices is itself hierarchically band diagonal (truncating, of course, newly generated elements that are smaller than the predetermined threshold ϵ); and moreover that (2) the product can be formed in order N operations.

Fast matrix multiplication makes it possible to find the matrix inverse by Schultz's (or Hotelling's) method, see §2.5.

Other schemes are also possible for fast solution of hierarchically band diagonal forms. For example, one can use the conjugate gradient method, implemented in §2.7 as `linbcg`.

CITED REFERENCES AND FURTHER READING:

Daubechies, I. 1992, *Wavelets* (Philadelphia: S.I.A.M.).

Strang, G. 1989, *SIAM Review*, vol. 31, pp. 614–627.

Beylkin, G., Coifman, R., and Rokhlin, V. 1991, *Communications on Pure and Applied Mathematics*, vol. 44, pp. 141–183. [1]

Daubechies, I. 1988, *Communications on Pure and Applied Mathematics*, vol. 41, pp. 909–996. [2]

Vaidyanathan, P.P. 1990, *Proceedings of the IEEE*, vol. 78, pp. 56–93. [3]

Mallat, S.G. 1989, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, pp. 674–693. [4]

Freedman, M.H., and Press, W.H. 1992, preprint. [5]

13.11 Numerical Use of the Sampling Theorem

In §6.10 we implemented an approximating formula for Dawson's integral due to Rybicki. Now that we have become Fourier sophisticates, we can learn that the formula derives from *numerical* application of the sampling theorem (§12.1), normally considered to be a purely analytic tool. Our discussion is identical to Rybicki [1].

For present purposes, the sampling theorem is most conveniently stated as follows: Consider an arbitrary function $g(t)$ and the grid of sampling points $t_n = \alpha + nh$, where n ranges over the integers and α is a constant that allows an arbitrary shift of the sampling grid. We then write

$$g(t) = \sum_{n=-\infty}^{\infty} g(t_n) \operatorname{sinc} \frac{\pi}{h}(t - t_n) + e(t) \tag{13.11.1}$$

where $\operatorname{sinc} x \equiv \sin x/x$. The summation over the sampling points is called the *sampling representation* of $g(t)$, and $e(t)$ is its error term. The sampling theorem asserts that the sampling representation is exact, that is, $e(t) \equiv 0$, if the Fourier transform of $g(t)$,

$$G(\omega) = \int_{-\infty}^{\infty} g(t) e^{i\omega t}\, dt \tag{13.11.2}$$

vanishes identically for $|\omega| \geq \pi/h$.

When can sampling representations be used to advantage for the approximate numerical computation of functions? In order that the error term be small, the Fourier transform $G(\omega)$ must be sufficiently small for $|\omega| \geq \pi/h$. On the other hand, in order for the summation in (13.11.1) to be approximated by a reasonably small number of terms, the function $g(t)$

itself should be very small outside of a fairly limited range of values of t. Thus we are led to two conditions to be satisfied in order that (13.11.1) be useful numerically: Both the function $g(t)$ and its Fourier transform $G(\omega)$ must rapidly approach zero for large values of their respective arguments.

Unfortunately, these two conditions are mutually antagonistic — the Uncertainty Principle in quantum mechanics. There exist strict limits on how rapidly the simultaneous approach to zero can be in both arguments. According to a theorem of Hardy [2], if $g(t) = O(e^{-t^2})$ as $|t| \to \infty$ and $G(\omega) = O(e^{-\omega^2/4})$ as $|\omega| \to \infty$, then $g(t) \equiv Ce^{-t^2}$, where C is a constant. This can be interpreted as saying that of all functions the Gaussian is the most rapidly decaying in both t and ω, and in this sense is the "best" function to be expressed numerically as a sampling representation.

Let us then write for the Gaussian $g(t) = e^{-t^2}$,

$$e^{-t^2} = \sum_{n=-\infty}^{\infty} e^{-t_n^2} \operatorname{sinc} \frac{\pi}{h}(t - t_n) + e(t) \tag{13.11.3}$$

The error $e(t)$ depends on the parameters h and α as well as on t, but it is sufficient for the present purposes to state the bound,

$$|e(t)| < e^{-(\pi/2h)^2} \tag{13.11.4}$$

which can be understood simply as the order of magnitude of the Fourier transform of the Gaussian at the point where it "spills over" into the region $|\omega| > \pi/h$.

When the summation in (13.11.3) is approximated by one with finite limits, say from $N_0 - N$ to $N_0 + N$, where N_0 is the integer nearest to $-\alpha/h$, there is a further truncation error. However, if N is chosen so that $N > \pi/(2h^2)$, the truncation error in the summation is less than the bound given by (13.11.4), and, since this bound is an overestimate, we shall continue to use it for (13.11.3) as well. The truncated summation gives a remarkably accurate representation for the Gaussian even for moderate values of N. For example, $|e(t)| < 5 \times 10^{-5}$ for $h = 1/2$ and $N = 7$; $|e(t)| < 2 \times 10^{-10}$ for $h = 1/3$ and $N = 15$; and $|e(t)| < 7 \times 10^{-18}$ for $h = 1/4$ and $N = 25$.

One may ask, what is the point of such a numerical representation for the Gaussian, which can be computed so easily and quickly as an exponential? The answer is that many transcendental functions can be expressed as an integral involving the Gaussian, and by substituting (13.11.3) one can often find excellent approximations to the integrals as a sum over elementary functions.

Let us consider as an example the function $w(z)$ of the complex variable $z = x + iy$, related to the complex error function by

$$w(z) = e^{-z^2} \operatorname{erfc}(-iz) \tag{13.11.5}$$

having the integral representation

$$w(z) = \frac{1}{\pi i} \int_C \frac{e^{-t^2}\, dt}{t - z} \tag{13.11.6}$$

where the contour C extends from $-\infty$ to ∞, passing below z (see, e.g., [3]). Many methods exist for the evaluation of this function (e.g., [4]). Substituting the sampling representation (13.11.3) into (13.11.6) and performing the resulting elementary contour integrals, we obtain

$$w(z) \approx \frac{1}{\pi i} \sum_{n=-\infty}^{\infty} h e^{-t_n^2} \frac{1 - (-1)^n e^{-\pi i(\alpha - z)/h}}{t_n - z} \tag{13.11.7}$$

where we now omit the error term. One should note that there is no singularity as $z \to t_m$ for some $n = m$, but a special treatment of the mth term will be required in this case (for example, by power series expansion).

An alternative form of equation (13.11.7) can be found by expressing the complex exponential in (13.11.7) in terms of trigonometric functions and using the sampling representation

(13.11.3) with z replacing t. This yields

$$w(z) \approx e^{-z^2} + \frac{1}{\pi i} \sum_{n=-\infty}^{\infty} h e^{-t_n^2} \frac{1-(-1)^n \cos \pi(\alpha - z)/h}{t_n - z} \tag{13.11.8}$$

This form is particularly useful in obtaining Re $w(z)$ when $|y| \ll 1$. Note that in evaluating (13.11.7) the exponential inside the summation is a constant and needs to be evaluated only once; a similar comment holds for the cosine in (13.11.8).

There are a variety of formulas that can now be derived from either equation (13.11.7) or (13.11.8) by choosing particular values of α. Eight interesting choices are: $\alpha = 0$, x, iy, or z, plus the values obtained by adding $h/2$ to each of these. Since the error bound (13.11.3) assumed a real value of α, the choices involving a complex α are useful only if the imaginary part of z is not too large. This is not the place to catalog all sixteen possible formulas, and we give only two particular cases that show some of the important features.

First of all let $\alpha = 0$ in equation (13.11.8), which yields,

$$w(z) \approx e^{-z^2} + \frac{1}{\pi i} \sum_{n=-\infty}^{\infty} h e^{-(nh)^2} \frac{1-(-1)^n \cos(\pi z/h)}{nh - z} \tag{13.11.9}$$

This approximation is good over the entire z-plane. As stated previously, one has to treat the case where one denominator becomes small by expansion in a power series. Formulas for the case $\alpha = 0$ were discussed briefly in [5]. They are similar, but not identical, to formulas derived by Chiarella and Reichel [6], using the method of Goodwin [7].

Next, let $\alpha = z$ in (13.11.7), which yields

$$w(z) \approx e^{-z^2} - \frac{2}{\pi i} \sum_{n \text{ odd}} \frac{e^{-(z-nh)^2}}{n} \tag{13.11.10}$$

the sum being over all odd integers (positive and negative). Note that we have made the substitution $n \to -n$ in the summation. This formula is simpler than (13.11.9) and contains half the number of terms, but its error is worse if y is large. Equation (13.11.10) is the source of the approximation formula (6.10.3) for Dawson's integral, used in §6.10.

CITED REFERENCES AND FURTHER READING:

Rybicki, G.B. 1989, *Computers in Physics*, vol. 3, no. 2, pp. 85–87. [1]

Hardy, G.H. 1933, *Journal of the London Mathematical Society*, vol. 8, pp. 227–231. [2]

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York). [3]

Gautschi, W. 1970, *SIAM Journal on Numerical Analysis*, vol. 7, pp. 187–198. [4]

Armstrong, B.H., and Nicholls, R.W. 1972, *Emission, Absorption and Transfer of Radiation in Heated Atmospheres* (New York: Pergamon). [5]

Chiarella, C., and Reichel, A. 1968, *Mathematics of Computation*, vol. 22, pp. 137–143. [6]

Goodwin, E.T. 1949, *Proceedings of the Cambridge Philosophical Society*, vol. 45, pp. 241–245. [7]