The discrete form of Parseval's theorem is

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

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

CITED REFERENCES AND FURTHER READING:

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

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

12.2 Fast Fourier Transform (FFT)

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

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

Then (12.1.7) can be written as

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

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

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

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

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

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

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

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

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

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

The next trick is to figure out which value of n corresponds to which pattern of e's and o's in equation (12.2.4). The answer is: Reverse the pattern of e's and o's, then let $e = 0$ and $o = 1$, and you will have, *in binary* the value of n. Do you see why it works? It is because the successive subdivisions of the data into even and odd are tests of successive low-order (least significant) bits of n. This idea of *bit reversal* can be exploited in a very clever way which, along with the Danielson-Lanczos

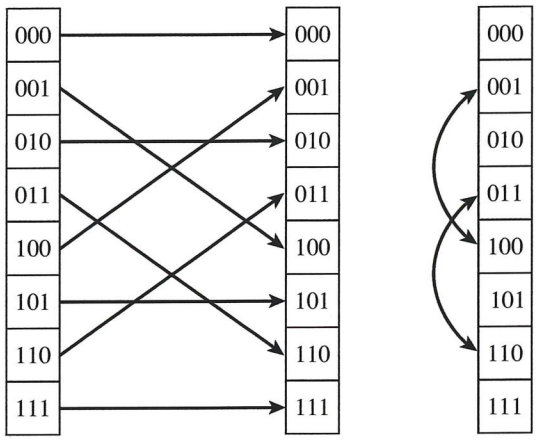

Figure 12.2.1. Reordering an array (here of length 8) by bit reversal, (a) between two arrays, versus (b) in place. Bit reversal reordering is a necessary part of the fast Fourier transform (FFT) algorithm.

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

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

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

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

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

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

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

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

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

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

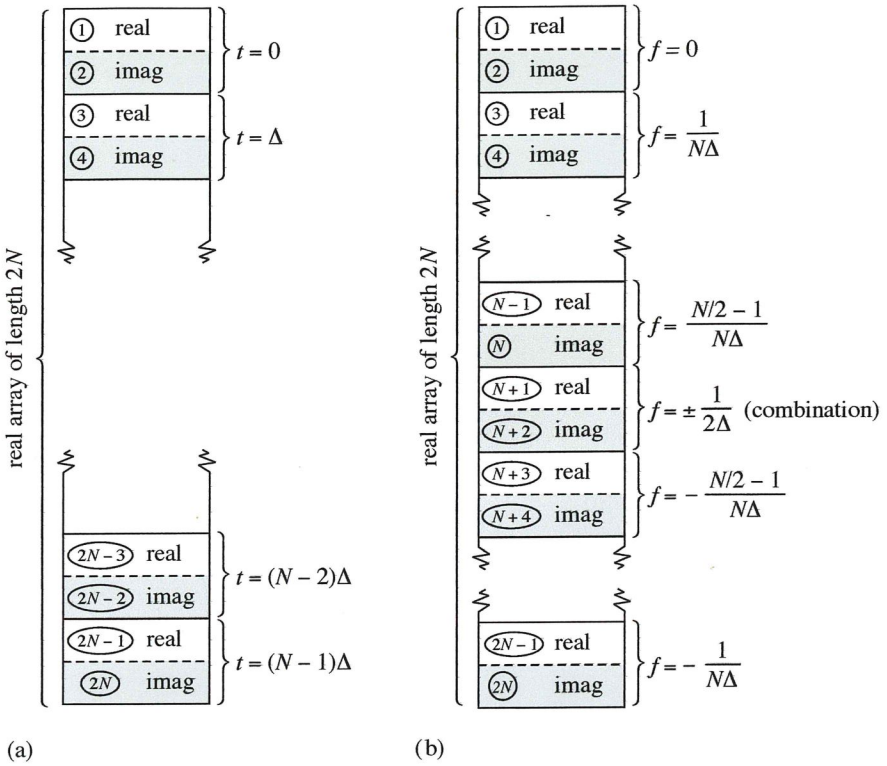

Figure 12.2.2. Input and output arrays for FFT. (a) The input array contains N (a power of 2) complex time samples in a real array of length $2N$, with real and imaginary parts alternating. (b) The output array contains the complex Fourier spectrum at N values of frequency. Real and imaginary parts again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

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

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

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

#define SIZE       8
#define PI         3.141592653589793
#define TWO_PI     (2.0 * PI)
#define SWAP(a,b)  tempr=(a);(a)=(b);(b)=tempr


//  The four1 FFT from Numerical Recipes in C,
//  p. 507 - 508.
//  Note:  changed float data types to double.
//  nn must be a power of 2, and use +1 for
//  isign for an FFT, and -1 for the Inverse FFT.
//  The data is complex, so the array size must be
//  nn*2. This code assumes the array starts
//  at index 1, not 0, so subtract 1 when
//  calling the routine (see main() below).

void four1(double data[], int nn, int isign)
{
    unsigned long n, mmax, m, j, istep, i;
    double wtemp, wr, wpr, wpi, wi, theta;
    double tempr, tempi;

    n = nn << 1;
    j = 1;

    for (i = 1; i < n; i += 2) {
        if (j > i) {
            SWAP(data[j], data[i]);
            SWAP(data[j+1], data[i+1]);
        }
        m = nn;
        while (m >= 2 && j > m) {
            j -= m;
            m >>= 1;
        }
        j += m;
    }

    mmax = 2;
    while (n > mmax) {
        istep = mmax << 1;
        theta = isign * (6.28318530717959 / mmax);
        wtemp = sin(0.5 * theta);
        wpr = -2.0 * wtemp * wtemp;
        wpi = sin(theta);
        wr = 1.0;
        wi = 0.0;
        for (m = 1; m < mmax; m += 2) {
            for (i = m; i <= n; i += istep) {
                j = i + mmax;
                tempr = wr * data[j] - wi * data[j+1];
                tempi = wr * data[j+1] + wi * data[j];
                data[j] = data[i] - tempr;
                data[j+1] = data[i+1] - tempi;
                data[i] += tempr;
                data[i+1] += tempi;
            }
            wr = (wtemp = wr) * wpr - wi * wpi + wr;
            wi = wi * wpr + wtemp * wpi + wi;
        }
        mmax = istep;
    }
}


// Creates a sine tone with the specified harmonic number.
// The array will be filled with complex numbers, and the
// signal is real (the imaginary parts are set to 0).

void createComplexSine(double data[], int size, int harmonicNumber)
{
    int i, ii;

    for (i = 0, ii = 0; i < size; i++, ii += 2) {
        data[ii] = sin((double)harmonicNumber * (double)i * TWO_PI / (double)size);
        data[ii+1] = 0.0;
    }
}


// Creates a cosine tone with the specified harmonic number.
// The array will be filled with complex numbers, and the
// signal is real (the imaginary parts are set to 0).

void createComplexCosine(double data[], int size, int harmonicNumber)
{
    int i, ii;

    for (i = 0, ii = 0; i < size; i++, ii += 2) {
        data[ii] = cos((double)harmonicNumber * (double)i * TWO_PI / (double)size);
        data[ii+1] = 0.0;
    }
}


// Creates a sawtooth wave, where each harmonic has
// the amplitude of 1 / harmonic_number.
// The array will be filled with complex numbers, and the
// signal is real (the imaginary parts are set to 0)

void createComplexSawtooth(double data[], int size)
{
    int i, ii, j;

    //  Calculate waveform using additive synthesis
    for (i = 0, ii = 0; i < size; i++, ii += 2) {
        data[ii] = 0.0;
        data[ii+1] = 0.0;
        for (j = 1; j <= size/2; j++) {
            data[ii] +=
                (cos((double)j * (double)i * TWO_PI / (double)size)) / (double)j;
        }
    }
}


// Display the real and imaginary parts
// the data contained in the array.

void displayComplex(double data[], int size)
{
```

```
    int i, ii;

    printf("\t\tReal part \tImaginary Part\n");

    for (i = 0, ii = 0; i < size; i++, ii += 2)
        printf("data[%-d]: \t%.6f \t%.6f\n", i, data[ii], data[ii+1]);

    printf("\n");
}



// Performs the DFT on the input data,
// which is assumed to be a real signal.
// That is, only data at even indices is
// used to calculate the spectrum.

void complexDFT(double x[], int N)
{
    int n, k, nn;
    double omega = TWO_PI / (double)N;
    double *a, *b;

    // Allocate temporary arrays
    a = (double *)calloc(N, sizeof(double));
    b = (double *)calloc(N, sizeof(double));

    // Perform the DFT
    for (k = 0; k < N; k++) {
        a[k] = b[k] = 0.0;
        for (n = 0, nn = 0; n < N; n++, nn += 2) {
            a[k] += (x[nn] * cos(omega * n * k));
            b[k] -= (x[nn] * sin(omega * n * k));
        }
    }

    // Pack result back into input data array
    for (n = 0, k = 0; n < N*2; n += 2, k++) {
        x[n] = a[k];
        x[n+1] = b[k];
    }

    // Free up memory used for arrays
    free(a);
    free(b);
}



// Takes the results from a DFT or FFT, and
// calculates and displays the amplitudes of
// the harmonics.

void postProcessComplex(double x[], int N)
{
    int i, k, j;
    double *amplitude, *result;

    // Allocate temporary arrays
    amplitude = (double *)calloc(N, sizeof(double));
    result = (double *)calloc(N, sizeof(double));

    // Calculate amplitude
    for (k = 0, i = 0; k < N; k++, i += 2) {
        // Scale results by N
        double real = x[i] / (double)N;
        double imag = x[i+1] / (double) N;
        // Calculate amplitude
        amplitude[k] = sqrt(real * real + imag * imag);
    }

    // Combine amplitudes of positive and negative frequencies
    result[0] = amplitude[0];
    result[N/2] = amplitude[N/2];
    for (k = 1, j = N-1; k < N/2; k++, j--)
        result[k] = amplitude[k] + amplitude[j];


    // Print out final result
    printf("Harmonic \tAmplitude\n");
    printf("DC \t\t%.6f\n", result[0]);
    for (k = 1; k <= N/2; k++)
        printf("%-d \t\t%.6f\n", k, result[k]);
    printf("\n");

    // Free up memory used for arrays
    free(amplitude);
    free(result);
}



int main()
{
    int i;
    double complexData[SIZE*2];

    // Try the DFT on a sawtooth waveform
    createComplexSawtooth(complexData, SIZE);
    displayComplex(complexData, SIZE);
    complexDFT(complexData, SIZE);
    postProcessComplex(complexData, SIZE);

    // Try the FFT on the same data
    createComplexSawtooth(complexData, SIZE);
    displayComplex(complexData, SIZE);
    four1(complexData-1, SIZE, 1);
    postProcessComplex(complexData, SIZE);
}
```

	Real part	Imaginary Part
data[0]:	2.083333	0.000000
data[1]:	0.221405	0.000000
data[2]:	-0.250000	0.000000
data[3]:	-0.721405	0.000000
data[4]:	-0.583333	0.000000
data[5]:	-0.721405	0.000000
data[6]:	-0.250000	0.000000
data[7]:	0.221405	0.000000

Harmonic	Amplitude
DC	0.000000
1	1.000000
2	0.500000
3	0.333333
4	0.250000

	Real part	Imaginary Part
data[0]:	2.083333	0.000000
data[1]:	0.221405	0.000000
data[2]:	-0.250000	0.000000
data[3]:	-0.721405	0.000000
data[4]:	-0.583333	0.000000
data[5]:	-0.721405	0.000000
data[6]:	-0.250000	0.000000
data[7]:	0.221405	0.000000

Harmonic	Amplitude
DC	0.000000
1	1.000000
2	0.500000
3	0.333333
4	0.250000