integer arithmetic modulo some large prime N+1, and the Nth root of 1 by the modulo arithmetic equivalent. Strictly speaking, these are not *Fourier* transforms at all, but the properties are quite similar and computational speed can be far superior. On the other hand, their use is somewhat restricted to quantities like correlations and convolutions since the transform itself is not easily interpretable as a "frequency" spectrum.

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). [1]

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

Van Loan, C. 1992, *Computational Frameworks for the Fast Fourier Transform* (Philadelphia: S.I.A.M.).

Beauchamp, K.G. 1984, *Applications of Walsh Functions and Related Functions* (New York: Academic Press) [non-Fourier transforms].

Heideman, M.T., Johnson, D.H., and Burris, C.S. 1984, *IEEE ASSP Magazine*, pp. 14–21 (October).

12.3 FFT of Real Functions, Sine and Cosine Transforms

It happens frequently that the data whose FFT is desired consist of real-valued samples $f_j,\ j = 0 \dots N-1$. To use `four1`, we put these into a complex array with all imaginary parts set to zero. The resulting transform $F_n,\ n = 0 \dots N-1$ satisfies $F_{N-n}{}^* = F_n$. Since this complex-valued array has real values for F_0 and $F_{N/2}$, and $(N/2)-1$ other independent values $F_1 \dots F_{N/2-1}$, it has the same $2(N/2-1)+2 = N$ "degrees of freedom" as the original, real data set. However, the use of the full complex FFT algorithm for real data is inefficient, both in execution time and in storage required. You would think that there is a better way.

There are *two* better ways. The first is "mass production": Pack two separate real functions into the input array in such a way that their individual transforms can be separated from the result. This is implemented in the program `twofft` below. This may remind you of a one-cent sale, at which you are coerced to purchase two of an item when you only need one. However, remember that for correlations and convolutions the Fourier transforms of two functions are involved, and this is a handy way to do them both at once. The second method is to pack the real input array cleverly, without extra zeros, into a complex array of half its length. One then performs a complex FFT on this shorter length; the trick is then to get the required answer out of the result. This is done in the program `realft` below.

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

Transform of Two Real Functions Simultaneously

First we show how to exploit the symmetry of the transform F_n to handle two real functions at once: Since the input data f_j are real, the components of the discrete Fourier transform satisfy

$$F_{N-n} = (F_n)^* \tag{12.3.1}$$

where the asterisk denotes complex conjugation. By the same token, the discrete Fourier transform of a purely imaginary set of g_j's has the opposite symmetry.

$$G_{N-n} = -(G_n)^* \tag{12.3.2}$$

Therefore we can take the discrete Fourier transform of two real functions each of length N simultaneously by packing the two data arrays as the real and imaginary parts, respectively, of the complex input array of `four1`. Then the resulting transform array can be unpacked into two complex arrays with the aid of the two symmetries. Routine `twofft` works out these ideas.

```
void twofft(float data1[], float data2[], float fft1[], float fft2[],
    unsigned long n)
```

Given two real input arrays `data1[1..n]` and `data2[1..n]`, this routine calls `four1` and returns two complex output arrays, `fft1[1..2n]` and `fft2[1..2n]`, each of complex length `n` (i.e., real length `2*n`), which contain the discrete Fourier transforms of the respective `data` arrays. `n` MUST be an integer power of 2.

```
{
    void four1(float data[], unsigned long nn, int isign);
    unsigned long nn3,nn2,jj,j;
    float rep,rem,aip,aim;

    nn3=1+(nn2=2+n+n);
    for (j=1,jj=2;j<=n;j++,jj+=2) {          Pack the two real arrays into one com-
        fft1[jj-1]=data1[j];                      plex array.
        fft1[jj]=data2[j];
    }
    four1(fft1,n,1);                          Transform the complex array.
    fft2[1]=fft1[2];
    fft1[2]=fft2[2]=0.0;
    for (j=3;j<=n+1;j+=2) {
        rep=0.5*(fft1[j]+fft1[nn2-j]);        Use symmetries to separate the two trans-
        rem=0.5*(fft1[j]-fft1[nn2-j]);            forms.
        aip=0.5*(fft1[j+1]+fft1[nn3-j]);
        aim=0.5*(fft1[j+1]-fft1[nn3-j]);
        fft1[j]=rep;                          Ship them out in two complex arrays.
        fft1[j+1]=aim;
        fft1[nn2-j]=rep;
        fft1[nn3-j] = -aim;
        fft2[j]=aip;
        fft2[j+1] = -rem;
        fft2[nn2-j]=aip;
        fft2[nn3-j]=rem;
    }
}
```

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

What about the reverse process? Suppose you have two complex transform arrays, each of which has the symmetry (12.3.1), so that you know that the inverses of both transforms are real functions. Can you invert both in a single FFT? This is even easier than the other direction. Use the fact that the FFT is linear and form the sum of the first transform plus i times the second. Invert using `four1` with `isign` $= -1$. The real and imaginary parts of the resulting complex array are the two desired real functions.

FFT of Single Real Function

To implement the second method, which allows us to perform the FFT of a *single* real function without redundancy, we split the data set in half, thereby forming two real arrays of half the size. We can apply the program above to these two, but of course the result will not be the transform of the original data. It will be a schizophrenic combination of two transforms, each of which has half of the information we need. Fortunately, this schizophrenia is treatable. It works like this:

The right way to split the original data is to take the even-numbered f_j as one data set, and the odd-numbered f_j as the other. The beauty of this is that we can take the original real array and treat it as a complex array h_j of half the length. The first data set is the real part of this array, and the second is the imaginary part, as prescribed for `twofft`. No repacking is required. In other words $h_j = f_{2j} + if_{2j+1}, \quad j = 0, \ldots, N/2-1$. We submit this to `four1`, and it will give back a complex array $H_n = F_n^e + iF_n^o, \quad n = 0, \ldots, N/2-1$ with

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

The discussion of program `twofft` tells you how to separate the two transforms F_n^e and F_n^o out of H_n. How do you work them into the transform F_n of the original data set f_j? Simply glance back at equation (12.2.3):

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

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

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

A few remarks:

- Since $F_{N-n}{}^* = F_n$ there is no point in saving the entire spectrum. The positive frequency half is sufficient and can be stored in the same array as the original data. The operation can, in fact, be done in place.
- Even so, we need values $H_n, \ n = 0, \ldots, N/2$ whereas `four1` gives only the values $n = 0, \ldots, N/2-1$. Symmetry to the rescue, $H_{N/2} = H_0$.

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

- The values F_0 and $F_{N/2}$ are real and independent. In order to actually get the entire F_n in the original array space, it is convenient to put $F_{N/2}$ into the imaginary part of F_0.
- Despite its complicated form, the process above is invertible. First peel $F_{N/2}$ out of F_0. Then construct

$$\begin{aligned} F_n^e &= \frac{1}{2}(F_n + F_{N/2-n}^*) \\ F_n^o &= \frac{1}{2}e^{-2\pi i n/N}(F_n - F_{N/2-n}^*) \end{aligned} \qquad n = 0, \ldots, N/2-1 \tag{12.3.6}$$

  and use `four1` to find the inverse transform of $H_n = F_n^{(1)} + iF_n^{(2)}$. Surprisingly, the actual algebraic steps are virtually identical to those of the forward transform.

Here is a representation of what we have said:

```
#include <math.h>

void realft(float data[], unsigned long n, int isign)
```

Calculates the Fourier transform of a set of `n` real-valued data points. Replaces this data (which is stored in array `data[1..n]`) by the positive frequency half of its complex Fourier transform. The real-valued first and last components of the complex transform are returned as elements `data[1]` and `data[2]`, respectively. `n` must be a power of 2. This routine also calculates the inverse transform of a complex data array if it is the transform of real data. (Result in this case must be multiplied by `2/n`.)

```
{
    void four1(float data[], unsigned long nn, int isign);
    unsigned long i,i1,i2,i3,i4,np3;
    float c1=0.5,c2,h1r,h1i,h2r,h2i;
    double wr,wi,wpr,wpi,wtemp,theta;              Double precision for the trigonomet-
                                                       ric recurrences.
    theta=3.141592653589793/(double) (n>>1);      Initialize the recurrence.
    if (isign == 1) {
        c2 = -0.5;
        four1(data,n>>1,1);                        The forward transform is here.
    } else {
        c2=0.5;                                    Otherwise set up for an inverse trans-
        theta = -theta;                                form.
    }
    wtemp=sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi=sin(theta);
    wr=1.0+wpr;
    wi=wpi;
    np3=n+3;
    for (i=2;i<=(n>>2);i++) {                      Case i=1 done separately below.
        i4=1+(i3=np3-(i2=1+(i1=i+i-1)));
        h1r=c1*(data[i1]+data[i3]);                The two separate transforms are sep-
        h1i=c1*(data[i2]-data[i4]);                    arated out of data.
        h2r = -c2*(data[i2]+data[i4]);
        h2i=c2*(data[i1]-data[i3]);
        data[i1]=h1r+wr*h2r-wi*h2i;                Here they are recombined to form
        data[i2]=h1i+wr*h2i+wi*h2r;                    the true transform of the origi-
        data[i3]=h1r-wr*h2r+wi*h2i;                    nal real data.
        data[i4] = -h1i+wr*h2i+wi*h2r;
        wr=(wtemp=wr)*wpr-wi*wpi+wr;               The recurrence.
        wi=wi*wpr+wtemp*wpi+wi;
    }
    if (isign == 1) {
```

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

```
        data[1] = (h1r=data[1])+data[2];        Squeeze the first and last data to-
        data[2] = h1r-data[2];                      gether to get them all within the
    } else {                                        original array.
        data[1]=c1*((h1r=data[1])+data[2]);
        data[2]=c1*(h1r-data[2]);
        four1(data,n>>1,-1);                    This is the inverse transform for the
    }                                               case isign=-1.
}
```

Fast Sine and Cosine Transforms

Among their other uses, the Fourier transforms of functions can be used to solve differential equations (see §19.4). The most common boundary conditions for the solutions are 1) they have the value zero at the boundaries, or 2) their derivatives are zero at the boundaries. In the first instance, the natural transform to use is the *sine* transform, given by

$$F_k = \sum_{j=1}^{N-1} f_j \sin(\pi jk/N) \qquad \text{sine transform} \tag{12.3.7}$$

where $f_j,\ j = 0, \ldots, N-1$ is the data array, and $f_0 \equiv 0$.

At first blush this appears to be simply the imaginary part of the discrete Fourier transform. However, the argument of the sine differs by a factor of two from the value that would make this so. The sine transform uses *sines only* as a complete set of functions in the interval from 0 to 2π, and, as we shall see, the cosine transform uses *cosines only*. By contrast, the normal FFT uses both sines and cosines, but only half as many of each. (See Figure 12.3.1.)

The expression (12.3.7) can be “force-fit” into a form that allows its calculation via the FFT. The idea is to extend the given function rightward past its last tabulated value. We extend the data to twice their length in such a way as to make them an *odd* function about $j = N$, with $f_N = 0$,

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

Consider the FFT of this extended function:

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

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

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

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

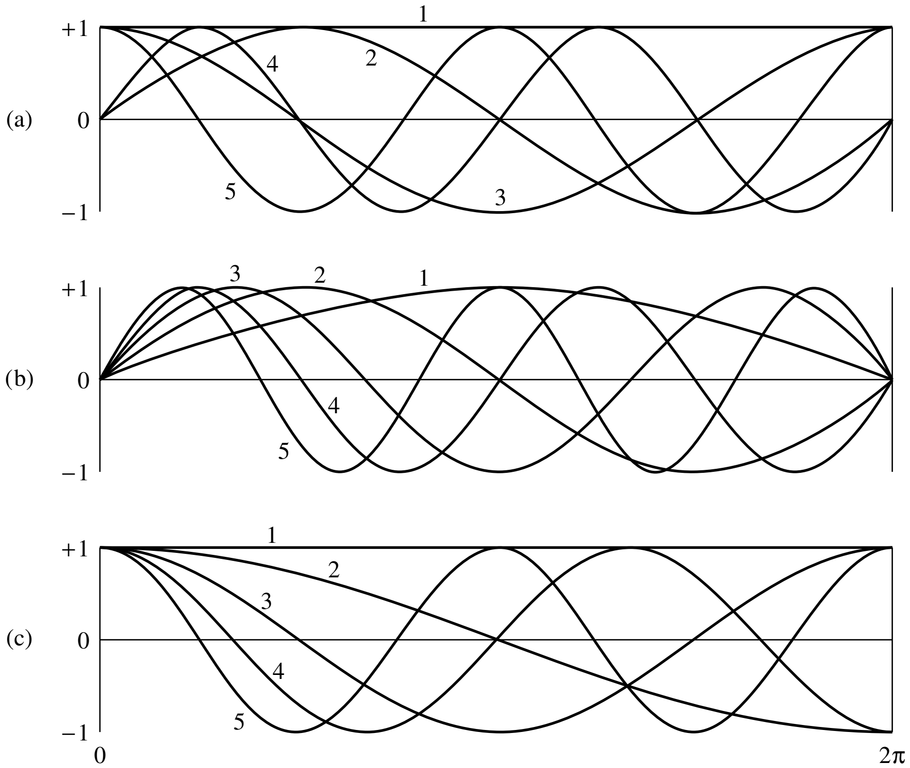

Figure 12.3.1. Basis functions used by the Fourier transform (a), sine transform (b), and cosine transform (c), are plotted. The first five basis functions are shown in each case. (For the Fourier transform, the real and imaginary parts of the basis functions are both shown.) While some basis functions occur in more than one transform, the basis sets are distinct. For example, the sine transform functions labeled (1), (3), (5) are not present in the Fourier basis. Any of the three sets can expand any function in the interval shown; however, the sine or cosine transform best expands functions matching the boundary conditions of the respective basis functions, namely zero function values for sine, zero derivatives for cosine.

so that

$$\begin{aligned} F_k &= \sum_{j=0}^{N-1} f_j \left[e^{2\pi ijk/(2N)} - e^{-2\pi ijk/(2N)} \right] \\ &= 2i \sum_{j=0}^{N-1} f_j \sin(\pi jk/N) \end{aligned} \tag{12.3.11}$$

Thus, up to a factor $2i$ we get the sine transform from the FFT of the extended function.

This method introduces a factor of two inefficiency into the computation by extending the data. This inefficiency shows up in the FFT output, which has zeros for the real part of every element of the transform. For a one-dimensional problem, the factor of two may be bearable, especially in view of the simplicity of the method. When we work with partial differential equations in two or three dimensions, though, the factor becomes four or eight, so efforts to eliminate the inefficiency are well rewarded.

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

From the original real data array f_j we will construct an auxiliary array y_j and apply to it the routine `realft`. The output will then be used to construct the desired transform. For the sine transform of data f_j, $j = 1, \ldots, N-1$, the auxiliary array is

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

This array is of the same dimension as the original. Notice that the first term is symmetric about $j = N/2$ and the second is antisymmetric. Consequently, when `realft` is applied to y_j, the result has real parts R_k and imaginary parts I_k given by

$$
\begin{aligned}
R_k &= \sum_{j=0}^{N-1} y_j \cos(2\pi jk/N) \\
&= \sum_{j=1}^{N-1} (f_j + f_{N-j}) \sin(j\pi/N) \cos(2\pi jk/N) \\
&= \sum_{j=0}^{N-1} 2 f_j \sin(j\pi/N) \cos(2\pi jk/N) \\
&= \sum_{j=0}^{N-1} f_j \left[\sin \frac{(2k+1)j\pi}{N} - \sin \frac{(2k-1)j\pi}{N} \right] \\
&= F_{2k+1} - F_{2k-1}
\end{aligned}
\tag{12.3.13}
$$

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

Therefore F_k can be determined as follows:

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

The even terms of F_k are thus determined very directly. The odd terms require a recursion, the starting point of which follows from setting $k = 0$ in equation (12.3.15) and using $F_1 = -F_{-1}$:

$$
F_1 = \frac{1}{2} R_0 \tag{12.3.16}
$$

The implementing program is

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

```
#include <math.h>

void sinft(float y[], int n)
Calculates the sine transform of a set of n real-valued data points stored in array y[1..n].
The number n must be a power of 2. On exit y is replaced by its transform. This program,
without changes, also calculates the inverse sine transform, but in this case the output array
should be multiplied by 2/n.
{
    void realft(float data[], unsigned long n, int isign);
    int j,n2=n+2;
    float sum,y1,y2;
    double theta,wi=0.0,wr=1.0,wpi,wpr,wtemp;       Double precision in the trigono-
                                                       metric recurrences.
    theta=3.14159265358979/(double) n;              Initialize the recurrence.
    wtemp=sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi=sin(theta);
    y[1]=0.0;
    for (j=2;j<=(n>>1)+1;j++) {
        wr=(wtemp=wr)*wpr-wi*wpi+wr;         Calculate the sine for the auxiliary array.
        wi=wi*wpr+wtemp*wpi+wi;              The cosine is needed to continue the recurrence.
        y1=wi*(y[j]+y[n2-j]);                Construct the auxiliary array.
        y2=0.5*(y[j]-y[n2-j]);
        y[j]=y1+y2;                          Terms j and N - j are related.
        y[n2-j]=y1-y2;
    }
    realft(y,n,1);                           Transform the auxiliary array.
    y[1]*=0.5;                               Initialize the sum used for odd terms below.
    sum=y[2]=0.0;
    for (j=1;j<=n-1;j+=2) {
        sum += y[j];
        y[j]=y[j+1];                         Even terms determined directly.
        y[j+1]=sum;                          Odd terms determined by this running sum.
    }
}
```

The sine transform, curiously, is its own inverse. If you apply it twice, you get the original data, but multiplied by a factor of $N/2$.

The other common boundary condition for differential equations is that the derivative of the function is zero at the boundary. In this case the natural transform is the *cosine* transform. There are several possible ways of defining the transform. Each can be thought of as resulting from a different way of extending a given array to create an even array of double the length, and/or from whether the extended array contains $2N-1$, $2N$, or some other number of points. In practice, only two of the numerous possibilities are useful so we will restrict ourselves to just these two.

The first form of the cosine transform uses $N+1$ data points:

$$F_k = \frac{1}{2}[f_0 + (-1)^k f_N] + \sum_{j=1}^{N-1} f_j \cos(\pi jk/N) \tag{12.3.17}$$

It results from extending the given array to an even array about $j = N$, with

$$f_{2N-j} = f_j, \qquad j = 0, \ldots, N-1 \tag{12.3.18}$$

If you substitute this extended array into equation (12.3.9), and follow steps analogous to those leading up to equation (12.3.11), you will find that the Fourier transform is

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

just twice the cosine transform (12.3.17). Another way of thinking about the formula (12.3.17) is to notice that it is the Chebyshev Gauss-Lobatto quadrature formula (see §4.5), often used in Clenshaw-Curtis adaptive quadrature (see §5.9, equation 5.9.4).

Once again the transform can be computed without the factor of two inefficiency. In this case the auxiliary function is

$$y_j = \frac{1}{2}(f_j + f_{N-j}) - \sin(j\pi/N)(f_j - f_{N-j}) \qquad j = 0, \ldots, N-1 \quad (12.3.19)$$

Instead of equation (12.3.15), `realft` now gives

$$F_{2k} = R_k \qquad F_{2k+1} = F_{2k-1} + I_k \qquad k = 0, \ldots, (N/2-1) \qquad (12.3.20)$$

The starting value for the recursion for odd k in this case is

$$F_1 = \frac{1}{2}(f_0 - f_N) + \sum_{j=1}^{N-1} f_j \cos(j\pi/N) \qquad (12.3.21)$$

This sum does not appear naturally among the R_k and I_k, and so we accumulate it during the generation of the array y_j.

Once again this transform is its own inverse, and so the following routine works for both directions of the transformation. Note that although this form of the cosine transform has $N+1$ input and output values, it passes an array only of length N to `realft`.

```
#include <math.h>
#define PI 3.141592653589793

void cosft1(float y[], int n)
Calculates the cosine transform of a set y[1..n+1] of real-valued data points. The transformed
data replace the original data in array y. n must be a power of 2. This program, without
changes, also calculates the inverse cosine transform, but in this case the output array should
be multiplied by 2/n.
{
    void realft(float data[], unsigned long n, int isign);
    int j,n2;
    float sum,y1,y2;
    double theta,wi=0.0,wpi,wpr,wr=1.0,wtemp;
    Double precision for the trigonometric recurrences.

    theta=PI/n;                              Initialize the recurrence.
    wtemp=sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi=sin(theta);
    sum=0.5*(y[1]-y[n+1]);
    y[1]=0.5*(y[1]+y[n+1]);
    n2=n+2;
    for (j=2;j<=(n>>1);j++) {                j=n/2+1 unnecessary since y[n/2+1] unchanged.
        wr=(wtemp=wr)*wpr-wi*wpi+wr;         Carry out the recurrence.
        wi=wi*wpr+wtemp*wpi+wi;
        y1=0.5*(y[j]+y[n2-j]);               Calculate the auxiliary function.
        y2=(y[j]-y[n2-j]);
        y[j]=y1-wi*y2;                       The values for j and N − j are related.
        y[n2-j]=y1+wi*y2;
        sum += wr*y2;                        Carry along this sum for later use in unfold-
    }                                            ing the transform.
```

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

```
    realft(y,n,1);                    Calculate the transform of the auxiliary func-
    y[n+1]=y[2];                          tion.
    y[2]=sum;                         sum is the value of F1 in equation (12.3.21).
    for (j=4;j<=n;j+=2) {
        sum += y[j];                  Equation (12.3.20).
        y[j]=sum;
    }
}
```

The second important form of the cosine transform is defined by

$$F_k = \sum_{j=0}^{N-1} f_j \cos \frac{\pi k(j+\frac{1}{2})}{N} \tag{12.3.22}$$

with inverse

$$f_j = \frac{2}{N} \sum_{k=0}^{N-1}{}' F_k \cos \frac{\pi k(j+\frac{1}{2})}{N} \tag{12.3.23}$$

Here the prime on the summation symbol means that the term for $k = 0$ has a coefficient of $\frac{1}{2}$ in front. This form arises by extending the given data, defined for $j = 0, \ldots, N-1$, to $j = N, \ldots, 2N-1$ in such a way that it is even about the point $N - \frac{1}{2}$ and periodic. (It is therefore also even about $j = -\frac{1}{2}$.) The form (12.3.23) is related to Gauss-Chebyshev quadrature (see equation 4.5.19), to Chebyshev approximation (§5.8, equation 5.8.7), and Clenshaw-Curtis quadrature (§5.9).

This form of the cosine transform is useful when solving differential equations on "staggered" grids, where the variables are centered midway between mesh points. It is also the standard form in the field of data compression and image processing.

The auxiliary function used in this case is similar to equation (12.3.19):

$$y_j = \frac{1}{2}(f_j + f_{N-j-1}) - \sin \frac{\pi(j+\frac{1}{2})}{N}(f_j - f_{N-j-1}) \qquad j = 0, \ldots, N-1 \tag{12.3.24}$$

Carrying out the steps similar to those used to get from (12.3.12) to (12.3.15), we find

$$F_{2k} = \cos \frac{\pi k}{N} R_k - \sin \frac{\pi k}{N} I_k \tag{12.3.25}$$

$$F_{2k-1} = \sin \frac{\pi k}{N} R_k + \cos \frac{\pi k}{N} I_k + F_{2k+1} \tag{12.3.26}$$

Note that equation (12.3.26) gives

$$F_{N-1} = \frac{1}{2} R_{N/2} \tag{12.3.27}$$

Thus the even components are found directly from (12.3.25), while the odd components are found by recursing (12.3.26) down from $k = N/2 - 1$, using (12.3.27) to start.

Since the transform is not self-inverting, we have to reverse the above steps to find the inverse. Here is the routine:

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

```
#include <math.h>
#define PI 3.141592653589793

void cosft2(float y[], int n, int isign)
```
Calculates the "staggered" cosine transform of a set y[1..n] of real-valued data points. The transformed data replace the original data in array y. n must be a power of 2. Set isign to $+1$ for a transform, and to -1 for an inverse transform. For an inverse transform, the output array should be multiplied by $2/n$.
```
{
    void realft(float data[], unsigned long n, int isign);
    int i;
    float sum,sum1,y1,y2,ytemp;
    double theta,wi=0.0,wi1,wpi,wpr,wr=1.0,wr1,wtemp;
```
Double precision for the trigonometric recurrences.
```
    theta=0.5*PI/n;                           Initialize the recurrences.
    wr1=cos(theta);
    wi1=sin(theta);
    wpr = -2.0*wi1*wi1;
    wpi=sin(2.0*theta);
    if (isign == 1) {                         Forward transform.
        for (i=1;i<=n/2;i++) {
            y1=0.5*(y[i]+y[n-i+1]);           Calculate the auxiliary function.
            y2=wi1*(y[i]-y[n-i+1]);
            y[i]=y1+y2;
            y[n-i+1]=y1-y2;
            wr1=(wtemp=wr1)*wpr-wi1*wpi+wr1;  Carry out the recurrence.
            wi1=wi1*wpr+wtemp*wpi+wi1;
        }
        realft(y,n,1);                        Transform the auxiliary function.
        for (i=3;i<=n;i+=2) {                 Even terms.
            wr=(wtemp=wr)*wpr-wi*wpi+wr;
            wi=wi*wpr+wtemp*wpi+wi;
            y1=y[i]*wr-y[i+1]*wi;
            y2=y[i+1]*wr+y[i]*wi;
            y[i]=y1;
            y[i+1]=y2;
        }
        sum=0.5*y[2];                         Initialize recurrence for odd terms
        for (i=n;i>=2;i-=2) {                     with ½R_{N/2}.
            sum1=sum;                         Carry out recurrence for odd terms.
            sum += y[i];
            y[i]=sum1;
        }
    } else if (isign == -1) {                 Inverse transform.
        ytemp=y[n];
        for (i=n;i>=4;i-=2) y[i]=y[i-2]-y[i]; Form difference of odd terms.
        y[2]=2.0*ytemp;
        for (i=3;i<=n;i+=2) {                 Calculate R_k and I_k.
            wr=(wtemp=wr)*wpr-wi*wpi+wr;
            wi=wi*wpr+wtemp*wpi+wi;
            y1=y[i]*wr+y[i+1]*wi;
            y2=y[i+1]*wr-y[i]*wi;
            y[i]=y1;
            y[i+1]=y2;
        }
        realft(y,n,-1);
        for (i=1;i<=n/2;i++) {                Invert auxiliary array.
            y1=y[i]+y[n-i+1];
            y2=(0.5/wi1)*(y[i]-y[n-i+1]);
            y[i]=0.5*(y1+y2);
            y[n-i+1]=0.5*(y1-y2);
            wr1=(wtemp=wr1)*wpr-wi1*wpi+wr1;
            wi1=wi1*wpr+wtemp*wpi+wi1;
```

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

```
        }
    }
}
```

An alternative way of implementing this algorithm is to form an auxiliary function by copying the even elements of f_j into the first $N/2$ locations, and the odd elements into the next $N/2$ elements in reverse order. However, it is not easy to implement the alternative algorithm without a temporary storage array and we prefer the above in-place algorithm.

Finally, we mention that there exist fast cosine transforms for small N that do not rely on an auxiliary function or use an FFT routine. Instead, they carry out the transform directly, often coded in hardware for fixed N of small dimension [1].

CITED REFERENCES AND FURTHER READING:

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall), §10–10.

Sorensen, H.V., Jones, D.L., Heideman, M.T., and Burris, C.S. 1987, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-35, pp. 849–863.

Hou, H.S. 1987, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-35, pp. 1455–1461 [see for additional references].

Hockney, R.W. 1971, in *Methods in Computational Physics*, vol. 9 (New York: Academic Press).

Temperton, C. 1980, *Journal of Computational Physics*, vol. 34, pp. 314–329.

Clarke, R.J. 1985, *Transform Coding of Images*, (Reading, MA: Addison-Wesley).

Gonzalez, R.C., and Wintz, P. 1987, *Digital Image Processing*, (Reading, MA: Addison-Wesley).

Chen, W., Smith, C.H., and Fralick, S.C. 1977, *IEEE Transactions on Communications*, vol. COM-25, pp. 1004–1009. [1]

12.4 FFT in Two or More Dimensions

Given a complex function $h(k_1, k_2)$ defined over the two-dimensional grid $0 \le k_1 \le N_1 - 1,\ 0 \le k_2 \le N_2 - 1$, we can define its two-dimensional discrete Fourier transform as a complex function $H(n_1, n_2)$, defined over the same grid,

$$H(n_1, n_2) \equiv \sum_{k_2=0}^{N_2-1} \sum_{k_1=0}^{N_1-1} \exp(2\pi i k_2 n_2/N_2)\ \exp(2\pi i k_1 n_1/N_1)\ h(k_1, k_2) \tag{12.4.1}$$

By pulling the "subscripts 2" exponential outside of the sum over k_1, or by reversing the order of summation and pulling the "subscripts 1" outside of the sum over k_2, we can see instantly that the two-dimensional FFT can be computed by taking one-dimensional FFTs sequentially on each index of the original function. Symbolically,

$$\begin{aligned} H(n_1, n_2) &= \text{FFT-on-index-1}\,(\text{FFT-on-index-2}\,[h(k_1, k_2)]) \\ &= \text{FFT-on-index-2}\,(\text{FFT-on-index-1}\,[h(k_1, k_2)]) \end{aligned} \tag{12.4.2}$$

Sample page from NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43108-5)
Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs
visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).