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, $H_{N/2} = H_0$.

- 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) {
```

```
        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 $j' = 2N - j$

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