```
        }
    }
}
```

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,

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

For this to be practical, of course, both N_1 and N_2 should be some efficient length for an FFT, usually a power of 2. Programming a two-dimensional FFT, using (12.4.2) with a one-dimensional FFT routine, is a bit clumsier than it seems at first. Because the one-dimensional routine requires that its input be in consecutive order as a one-dimensional complex array, you find that you are endlessly copying things out of the multidimensional input array and then copying things back into it. This is not recommended technique. Rather, you should use a multidimensional FFT routine, such as the one we give below.

The generalization of (12.4.1) to more than two dimensions, say to L-dimensions, is evidently

$$H(n_1,\dots,n_L) \equiv \sum_{k_L=0}^{N_L-1} \cdots \sum_{k_1=0}^{N_1-1} \exp(2\pi i k_L n_L/N_L) \times \cdots \times \exp(2\pi i k_1 n_1/N_1)\ h(k_1,\dots,k_L) \tag{12.4.3}$$

where n_1 and k_1 range from 0 to N_1-1, ... , n_L and k_L range from 0 to N_L-1. How many calls to a one-dimensional FFT are in (12.4.3)? Quite a few! For each value of $k_1, k_2, \dots, k_{L-1}$ you FFT to transform the L index. Then for each value of $k_1, k_2, \dots, k_{L-2}$ and n_L you FFT to transform the $L-1$ index. And so on. It is best to rely on someone else having done the bookkeeping for once and for all.

The inverse transforms of (12.4.1) or (12.4.3) are just what you would expect them to be: Change the i's in the exponentials to $-i$'s, and put an overall factor of $1/(N_1 \times \cdots \times N_L)$ in front of the whole thing. Most other features of multidimensional FFTs are also analogous to features already discussed in the one-dimensional case:

- Frequencies are arranged in wrap-around order in the transform, but now for each separate dimension.
- The input data are also treated as if they were wrapped around. If they are discontinuous across this periodic identification (in any dimension) then the spectrum will have some excess power at high frequencies because of the discontinuity. The fix, if you care, is to remove multidimensional linear trends.
- If you are doing spatial filtering and are worried about wrap-around effects, then you need to zero-pad all around the border of the multidimensional array. However, be sure to notice how costly zero-padding is in multidimensional transforms. If you use too thick a zero-pad, you are going to waste a *lot* of storage, especially in 3 or more dimensions!
- Aliasing occurs as always if sufficient bandwidth limiting does not exist along one or more of the dimensions of the transform.

The routine `fourn` that we furnish herewith is a descendant of one written by N. M. Brenner. It requires as input (i) a scalar, telling the number of dimensions, e.g., 2; (ii) a vector, telling the length of the array in each dimension, e.g., (32,64). Note that these lengths *must all* be powers of 2, and are the numbers of *complex* values in each direction; (iii) the usual scalar equal to ± 1 indicating whether you want the transform or its inverse; and, finally (iv) the array of data.

A few words about the data array: `fourn` accesses it as a one-dimensional array of real numbers, that is, `data[1..`$(2N_1N_2\dots N_L)$`]`, of length equal to twice the

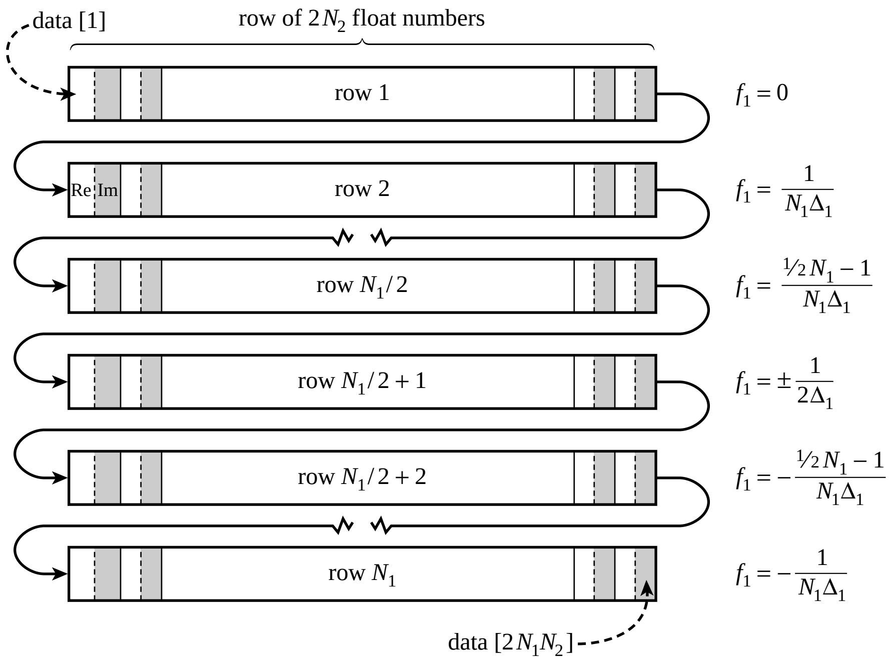
data [1]
row of $2N_2$ float numbers
row 1
Re
Im
row 2
row $N_1/2$
row $N_1/2+1$
row $N_1/2+2$
row N_1
data $[2N_1N_2]$
$f_1 = 0$
$f_1 = \frac{1}{N_1\Delta_1}$
$f_1 = \frac{½N_1 - 1}{N_1\Delta_1}$
$f_1 = \pm\frac{1}{2\Delta_1}$
$f_1 = -\frac{½N_1 - 1}{N_1\Delta_1}$
$f_1 = -\frac{1}{N_1\Delta_1}$

Figure 12.4.1. Storage arrangement of frequencies in the output $H(f_1, f_2)$ of a two-dimensional FFT. The input data is a two-dimensional $N_1 \times N_2$ array $h(t_1, t_2)$ (stored by rows of complex numbers). The output is also stored by complex rows. Each row corresponds to a particular value of f_1, as shown in the figure. Within each row, the arrangement of frequencies f_2 is exactly as shown in Figure 12.2.2. Δ_1 and Δ_2 are the sampling intervals in the 1 and 2 directions, respectively. The total number of (real) array elements is $2N_1N_2$. The program `fourn` can also do more than two dimensions, and the storage arrangement generalizes in the obvious way.

product of the lengths of the L dimensions. It assumes that the array represents an L-dimensional complex array, with individual components ordered as follows: (i) each complex value occupies two sequential locations, real part followed by imaginary; (ii) the first subscript changes least rapidly as one goes through the array; the last subscript changes most rapidly (that is, "store by rows," the `C` norm); (iii) subscripts range from 1 to their maximum values ($N_1, N_2, \ldots, N_L$, respectively), rather than from 0 to $N_1 - 1,\ N_2 - 1, \ldots,\ N_L - 1$. Almost all failures to get `fourn` to work result from improper understanding of the above ordering of the data array, so take care! (Figure 12.4.1 illustrates the format of the output array.)

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

void fourn(float data[], unsigned long nn[], int ndim, int isign)
```
Replaces `data` by its `ndim`-dimensional discrete Fourier transform, if `isign` is input as 1. `nn[1..ndim]` is an integer array containing the lengths of each dimension (number of complex values), which MUST all be powers of 2. `data` is a real array of length twice the product of these lengths, in which the data are stored as in a multidimensional complex array: real and imaginary parts of each element are in consecutive locations, and the rightmost index of the array increases most rapidly as one proceeds along `data`. For a two-dimensional array, this is equivalent to storing the array by rows. If `isign` is input as -1, `data` is replaced by its inverse transform times the product of the lengths of all dimensions.

```
{
    int idim;
    unsigned long i1,i2,i3,i2rev,i3rev,ip1,ip2,ip3,ifp1,ifp2;
    unsigned long ibit,k1,k2,n,nprev,nrem,ntot;
    float tempi,tempr;
    double theta,wi,wpi,wpr,wr,wtemp;          Double precision for trigonometric recur-
                                                   rences.
    for (ntot=1,idim=1;idim<=ndim;idim++)      Compute total number of complex val-
        ntot *= nn[idim];                          ues.
    nprev=1;
    for (idim=ndim;idim>=1;idim--) {           Main loop over the dimensions.
        n=nn[idim];
        nrem=ntot/(n*nprev);
        ip1=nprev << 1;
        ip2=ip1*n;
        ip3=ip2*nrem;
        i2rev=1;
        for (i2=1;i2<=ip2;i2+=ip1) {           This is the bit-reversal section of the
            if (i2 < i2rev) {                      routine.
                for (i1=i2;i1<=i2+ip1-2;i1+=2) {
                    for (i3=i1;i3<=ip3;i3+=ip2) {
                        i3rev=i2rev+i3-i2;
                        SWAP(data[i3],data[i3rev]);
                        SWAP(data[i3+1],data[i3rev+1]);
                    }
                }
            }
            ibit=ip2 >> 1;
            while (ibit >= ip1 && i2rev > ibit) {
                i2rev -= ibit;
                ibit >>= 1;
            }
            i2rev += ibit;
        }
        ifp1=ip1;                              Here begins the Danielson-Lanczos sec-
        while (ifp1 < ip2) {                       tion of the routine.
            ifp2=ifp1 << 1;
            theta=isign*6.28318530717959/(ifp2/ip1);      Initialize for the trig. recur-
            wtemp=sin(0.5*theta);                             rence.
            wpr = -2.0*wtemp*wtemp;
            wpi=sin(theta);
            wr=1.0;
            wi=0.0;
            for (i3=1;i3<=ifp1;i3+=ip1) {
                for (i1=i3;i1<=i3+ip1-2;i1+=2) {
                    for (i2=i1;i2<=ip3;i2+=ifp2) {
                        k1=i2;                 Danielson-Lanczos formula:
                        k2=k1+ifp1;
                        tempr=(float)wr*data[k2]-(float)wi*data[k2+1];
                        tempi=(float)wr*data[k2+1]+(float)wi*data[k2];
                        data[k2]=data[k1]-tempr;
                        data[k2+1]=data[k1+1]-tempi;
                        data[k1] += tempr;
                        data[k1+1] += tempi;
                    }
                }
                wr=(wtemp=wr)*wpr-wi*wpi+wr;   Trigonometric recurrence.
                wi=wi*wpr+wtemp*wpi+wi;
            }
            ifp1=ifp2;
        }
        nprev *= n;
    }
}
```

CITED REFERENCES AND FURTHER READING:

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

12.5 Fourier Transforms of Real Data in Two and Three Dimensions

Two-dimensional FFTs are particularly important in the field of image processing. An image is usually represented as a two-dimensional array of pixel intensities, real (and usually positive) numbers. One commonly desires to filter high, or low, frequency spatial components from an image; or to convolve or deconvolve the image with some instrumental point spread function. Use of the FFT is by far the most efficient technique.

In three dimensions, a common use of the FFT is to solve Poisson's equation for a potential (e.g., electromagnetic or gravitational) on a three-dimensional lattice that represents the discretization of three-dimensional space. Here the source terms (mass or charge distribution) and the desired potentials are also real. In two and three dimensions, with large arrays, memory is often at a premium. It is therefore important to perform the FFTs, insofar as possible, on the data "in place." We want a routine with functionality similar to the multidimensional FFT routine `fourn` (§12.4), but which operates on real, not complex, input data. We give such a routine in this section. The development is analogous to that of §12.3 leading to the one-dimensional routine `realft`. (You might wish to review that material at this point, particularly equation 12.3.5.)

It is convenient to think of the independent variables $n_1, \dots, n_L$ in equation (12.4.3) as representing an L-dimensional vector $\vec{n}$ in wave-number space, with values on the lattice of integers. The transform $H(n_1, \dots, n_L)$ is then denoted $H(\vec{n})$.

It is easy to see that the transform $H(\vec{n})$ is periodic in each of its L dimensions. Specifically, if $\vec{P}_1, \vec{P}_2, \vec{P}_3, \dots$ denote the vectors $(N_1, 0, 0, \dots)$, $(0, N_2, 0, \dots)$, $(0, 0, N_3, \dots)$, and so forth, then

$$H(\vec{n} \pm \vec{P}_j) = H(\vec{n}) \qquad j = 1, \dots, L \tag{12.5.1}$$

Equation (12.5.1) holds for any input data, real or complex. When the data is real, we have the additional symmetry

$$H(-\vec{n}) = H(\vec{n})* \tag{12.5.2}$$

Equations (12.5.1) and (12.5.2) imply that the full transform can be trivially obtained from the subset of lattice values $\vec{n}$ that have

$$\begin{aligned} &0 \le n_1 \le N_1 - 1 \\ &0 \le n_2 \le N_2 - 1 \\ &\qquad \cdots \\ &0 \le n_L \le \frac{N_L}{2} \end{aligned} \tag{12.5.3}$$