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 $n_1, \ldots, n_L$ 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 $H(n_1, \ldots, n_L)$ 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 $\vec{P}_1, \vec{P}_2, \vec{P}_3, \ldots$ denote the vectors $(N_1, 0, 0, \ldots)$, $(0, N_2, 0, \ldots)$, $(0, 0, N_3, \ldots)$, and so forth, then

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

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

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

In fact, this set of values is overcomplete, because there are additional symmetry relations among the transform values that have $n_L = 0$ and $n_L = N_L/2$. However these symmetries are complicated and their use becomes extremely confusing. Therefore, we will compute our FFT on the lattice subset of equation (12.5.3), even though this requires a small amount of extra storage for the answer, i.e., the transform is not *quite* "in place." (Although an in-place transform is in fact possible, we have found it virtually impossible to explain to any user how to unscramble its output, i.e., where to find the real and imaginary components of the transform at some particular frequency!)

We will implement the multidimensional real Fourier transform for the three dimensional case $L = 3$, with the input data stored as a real, three-dimensional array `data[1..nn1][1..nn2][1..nn3]`. This scheme will allow two-dimensional data to be processed with effectively no loss of efficiency simply by choosing `nn1` $= 1$. (Note that it must be the *first* dimension that is set to 1.) The output spectrum comes back packaged, logically at least, as a *complex*, three-dimensional array that we can call `SPEC[1..nn1][1..nn2][1..nn3/2+1]` (cf. equation 12.5.3). In the first two of its three dimensions, the respective frequency values f_1 or f_2 are stored in wrap-around order, that is with zero frequency in the first index value, the smallest positive frequency in the second index value, the smallest *negative* frequency in the *last* index value, and so on (cf. the discussion leading up to routines `four1` and `fourn`). The third of the three dimensions returns only the positive half of the frequency spectrum. Figure 12.5.1 shows the logical storage scheme. The returned portion of the complex output spectrum is shown as the unshaded part of the lower figure.

The physical, as opposed to logical, packaging of the output spectrum is necessarily a bit different from the logical packaging, because `C` does not have a convenient, portable mechanism for equivalencing real and complex arrays. The subscript range `SPEC[1..nn1][1..nn2][1..nn3/2]` is returned in the input array `data[1..nn1][1..nn2][1..nn3]`, with the correspondence

$$\begin{aligned}\mathrm{Re}(\texttt{SPEC[i1][i2][i3]}) &= \texttt{data[i1][i2][2*i3-1]}\\ \mathrm{Im}(\texttt{SPEC[i1][i2][i3]}) &= \texttt{data[i1][i2][2*i3]}\end{aligned} \tag{12.5.4}$$

The remaining "plane" of values, `SPEC[1..nn1][1..nn2][nn3/2+1]`, is returned in the two-dimensional `float` array `speq[1..nn1][1..2*nn2]`,with the correspondence

$$\begin{aligned}\mathrm{Re}(\texttt{SPEC[i1][i2][nn3/2+1]}) &= \texttt{speq[i1][2*i2-1]}\\ \mathrm{Im}(\texttt{SPEC[i1][i2][nn3/2+1]}) &= \texttt{speq[i1][2*i2]}\end{aligned} \tag{12.5.5}$$

Note that `speq` contains frequency components whose third component f_3 is at the Nyquist critical frequency $\pm f_c$. In some applications these values will in fact be ignored or set to zero, since they are intrinsically aliased between positive and negative frequencies.

With this much introduction, the implementing procedure, called `rlft3`, is something of an anticlimax. Look in the innermost loop in the procedure, and you will see equation (12.3.5) implemented on the *last* transform index. The case of `i3=1` is coded separately, to account for the fact that `speq` is to be filled instead of

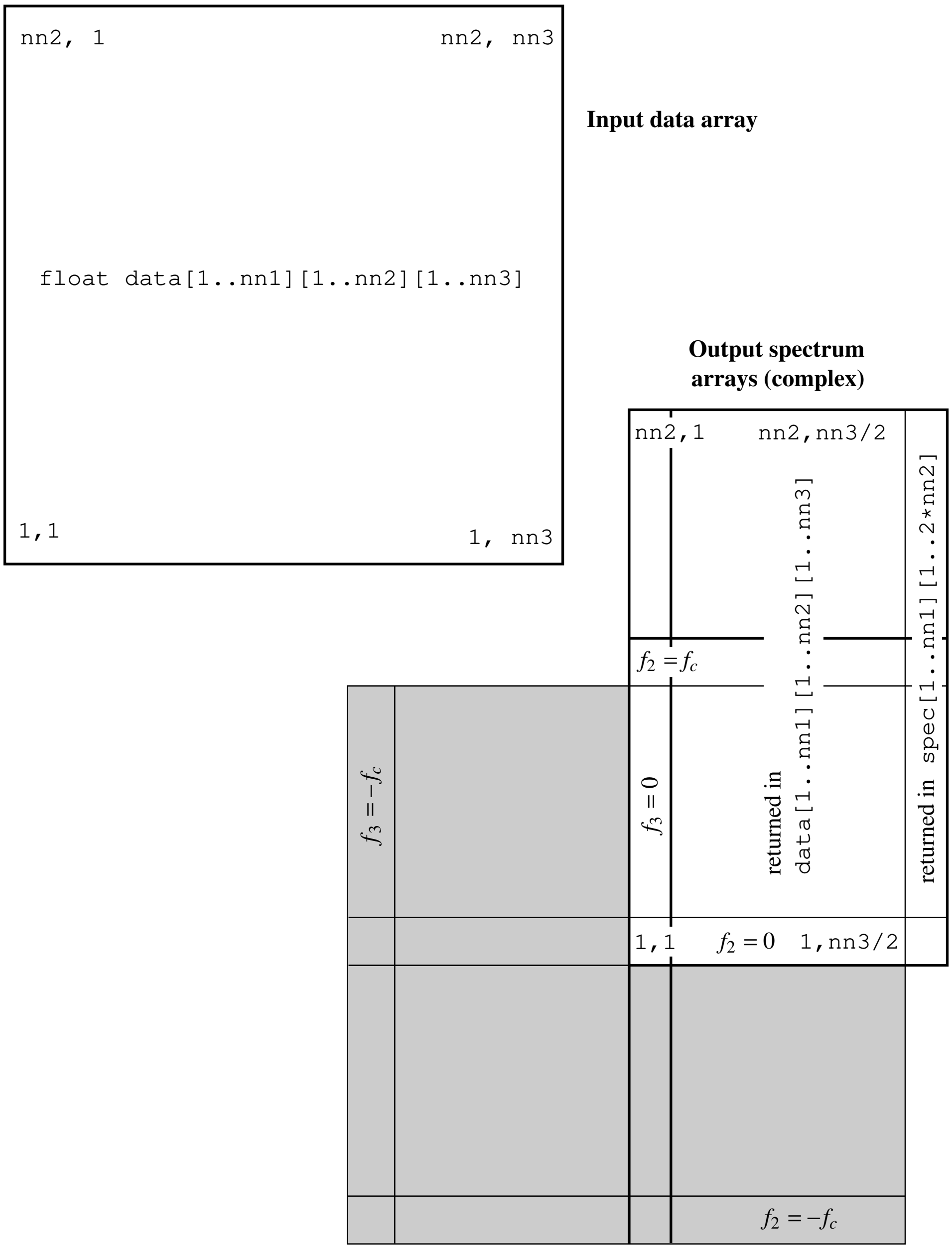

Figure 12.5.1. Input and output data arrangement for `rlft3`. All arrays shown are presumed to have a first (leftmost) dimension of range `[1..nn1]`, coming out of the page. The input data array is a real, three-dimensional array `data[1..nn1][1..nn2][1..nn3]`. (For two-dimensional data, one sets `nn1` = 1.) The output data can be viewed as a single complex array with dimensions `[1..nn1][1..nn2][1..nn3/2+1]` (cf. equation 12.5.3), corresponding to the frequency components f_1 and f_2 being stored in wrap-around order, but only positive f_3 values being stored (others being obtainable by symmetry). The output data is actually returned mostly in the input array `data`, but partly stored in the real array `speq[1..nn1][1..2*nn2]`. See text for details.

overwriting the input array of data. The three enclosing `for` loops (indices `i2`, `i3`, and `i1`, from inside to outside) could in fact be done in any order — their actions all commute. We chose the order shown because of the following considerations: (i) `i3` should not be the inner loop, because if it is, then the recurrence relations on `wr` and `wi` become burdensome. (ii) On virtual-memory machines, `i1` should be the outer loop, because (with C order of array storage) this results in the array `data`, which might be very large, being accessed in block sequential order.

Note that the work done in `rlft3` is quite (logarithmically) small, compared to the associated complex FFT, `fourn`. Since C does not have a convenient `complex` type, the operations are carried out explicitly below in terms of real and imaginary parts. The routine `rlft3` is based on an earlier routine by G.B. Rybicki.

```
#include <math.h>

void rlft3(float ***data, float **speq, unsigned long nn1, unsigned long nn2,
    unsigned long nn3, int isign)
```

Given a three-dimensional real array `data[1..nn1][1..nn2][1..nn3]` (where `nn1` = 1 for the case of a logically two-dimensional array), this routine returns (for `isign=1`) the complex fast Fourier transform as two complex arrays: On output, `data` contains the zero and positive frequency values of the third frequency component, while `speq[1..nn1][1..2*nn2]` contains the Nyquist critical frequency values of the third frequency component. First (and second) frequency components are stored for zero, positive, and negative frequencies, in standard wrap-around order. See text for description of how complex values are arranged. For `isign=-1`, the inverse transform (times `nn1*nn2*nn3/2` as a constant multiplicative factor) is performed, with output `data` (viewed as a real array) deriving from input `data` (viewed as complex) and `speq`. For inverse transforms on data not generated first by a forward transform, make sure the complex input data array satisfies property (12.5.2). The dimensions `nn1`, `nn2`, `nn3` must always be integer powers of 2.

```
{
    void fourn(float data[], unsigned long nn[], int ndim, int isign);
    void nrerror(char error_text[]);
    unsigned long i1,i2,i3,j1,j2,j3,nn[4],ii3;
    double theta,wi,wpi,wpr,wr,wtemp;
    float c1,c2,h1r,h1i,h2r,h2i;

    if (1+&data[nn1][nn2][nn3]-&data[1][1][1] != nn1*nn2*nn3)
        nrerror("rlft3: problem with dimensions or contiguity of data array\n");
    c1=0.5;
    c2 = -0.5*isign;
    theta=isign*(6.28318530717959/nn3);
    wtemp=sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi=sin(theta);
    nn[1]=nn1;
    nn[2]=nn2;
    nn[3]=nn3 >> 1;
    if (isign == 1) {                               Case of forward transform.
        fourn(&data[1][1][1]-1,nn,3,isign);         Here is where most all of the com-
        for (i1=1;i1<=nn1;i1++)                         pute time is spent.
            for (i2=1,j2=0;i2<=nn2;i2++) {          Extend data periodically into speq.
                speq[i1][++j2]=data[i1][i2][1];
                speq[i1][++j2]=data[i1][i2][2];
            }
    }
    for (i1=1;i1<=nn1;i1++) {
        j1=(i1 != 1 ? nn1-i1+2 : 1);
        Zero frequency is its own reflection, otherwise locate corresponding negative frequency
        in wrap-around order.
        wr=1.0;                                     Initialize trigonometric recurrence.
        wi=0.0;
        for (ii3=1,i3=1;i3<=(nn3>>2)+1;i3++,ii3+=2) {
```

Figure 12.5.2. (a) A two-dimensional image with intensities either purely black or purely white. (b) The same image, after it has been low-pass filtered using `rlft3`. Regions with fine-scale features become gray.

```
            for (i2=1;i2<=nn2;i2++) {
                if (i3 == 1) {                      Equation (12.3.5).
                    j2=(i2 != 1 ? ((nn2-i2)<<1)+3 : 1);
                    h1r=c1*(data[i1][i2][1]+speq[j1][j2]);
                    h1i=c1*(data[i1][i2][2]-speq[j1][j2+1]);
                    h2i=c2*(data[i1][i2][1]-speq[j1][j2]);
                    h2r= -c2*(data[i1][i2][2]+speq[j1][j2+1]);
                    data[i1][i2][1]=h1r+h2r;
                    data[i1][i2][2]=h1i+h2i;
                    speq[j1][j2]=h1r-h2r;
                    speq[j1][j2+1]=h2i-h1i;
                } else {
                    j2=(i2 != 1 ? nn2-i2+2 : 1);
                    j3=nn3+3-(i3<<1);
                    h1r=c1*(data[i1][i2][ii3]+data[j1][j2][j3]);
                    h1i=c1*(data[i1][i2][ii3+1]-data[j1][j2][j3+1]);
                    h2i=c2*(data[i1][i2][ii3]-data[j1][j2][j3]);
                    h2r= -c2*(data[i1][i2][ii3+1]+data[j1][j2][j3+1]);
                    data[i1][i2][ii3]=h1r+wr*h2r-wi*h2i;
                    data[i1][i2][ii3+1]=h1i+wr*h2i+wi*h2r;
                    data[j1][j2][j3]=h1r-wr*h2r+wi*h2i;
                    data[j1][j2][j3+1]= -h1i+wr*h2i+wi*h2r;
                }
            }
            wr=(wtemp=wr)*wpr-wi*wpi+wr;            Do the recurrence.
            wi=wi*wpr+wtemp*wpi+wi;
        }
    }
    if (isign == -1)                                Case of reverse transform.
        fourn(&data[1][1][1]-1,nn,3,isign);
}
```

We now give some fragments from notional calling programs, to clarify the use of `rlft3` for two- and three-dimensional data. Note again that the routine does not actually distinguish between two and three dimensions; two is treated like three, but with the first dimension having length 1. Since the first dimension is the outer loop, virtually no inefficiency is introduced.

The first program fragment FFTs a two-dimensional data array, allows for some processing on it, e.g., filtering, and then takes the inverse transform. Figure 12.5.2 shows an example of the use of this kind of code: A sharp image becomes blurry when its high-frequency spatial components are suppressed by the factor (here) max $(1 - 6f^2/f_c^2, 0)$. The second program example illustrates a three-dimensional transform, where the three dimensions have different lengths. The third program example is an example of convolution, as it might occur in a program to compute the potential generated by a three-dimensional distribution of sources.

```
#include <stdlib.h>
#include "nrutil.h"
#define N2 256
#define N3 256                     Note that the first component must be set to 1.

int main(void)        /* example1 */
This fragment shows how one might filter a 256 by 256 digital image.
{
    void rlft3(float ***data, float **speq, unsigned long nn1,
        unsigned long nn2, unsigned long nn3, int isign);
    float ***data, **speq;

    data=f3tensor(1,1,1,N2,1,N3);
    speq=matrix(1,1,1,2*N2);
/*    ...*/                        Here the image would be loaded into data.
    rlft3(data,speq,1,N2,N3,1);
/*    ...*/                        Here the arrays data and speq would be multiplied by a
    rlft3(data,speq,1,N2,N3,-1);       suitable filter function (of frequency).
/*    ...*/                        Here the filtered image would be unloaded from data.
    free_matrix(speq,1,1,1,2*N2);
    free_f3tensor(data,1,1,1,N2,1,N3);
    return 0;
}

#define N1 32
#define N2 64
#define N3 16

int main(void)        /* example2 */
This fragment shows how one might FFT a real three-dimensional array of size 32 by 64 by 16.
{
    void rlft3(float ***data, float **speq, unsigned long nn1,
        unsigned long nn2, unsigned long nn3, int isign);
    int j;
    float ***data,**speq;

    data=f3tensor(1,N1,1,N2,1,N3);
    speq=matrix(1,N1,1,2*N2);
/*    ...*/                        Here load data.
    rlft3(data,speq,N1,N2,N3,1);
/*    ...*/                        Here unload data and speq.
    free_matrix(speq,1,N1,1,2*N2);
    free_f3tensor(data,1,N1,1,N2,1,N3);
    return 0;
}

#define N 32

int main(void)        /* example3 */
This fragment shows how one might convolve two real, three-dimensional arrays of size 32 by
32 by 32, replacing the first array by the result.
{
    void rlft3(float ***data, float **speq, unsigned long nn1,
        unsigned long nn2, unsigned long nn3, int isign);
```

```
    int j;
    float fac,r,i,***data1,***data2,**speq1,**speq2,*sp1,*sp2;

    data1=f3tensor(1,N,1,N,1,N);
    data2=f3tensor(1,N,1,N,1,N);
    speq1=matrix(1,N,1,2*N);
    speq2=matrix(1,N,1,2*N);

/*    ...*/
    rlft3(data1,speq1,N,N,N,1);           FFT both input arrays.
    rlft3(data2,speq2,N,N,N,1);
    fac=2.0/(N*N*N);                      Factor needed to get normalized inverse.
    sp1 = &data1[1][1][1];
    sp2 = &data2[1][1][1];
    for (j=1;j<=N*N*N/2;j++) {            Note how this can be made a single for-loop in-
        r = sp1[0]*sp2[0] - sp1[1]*sp2[1];        stead of three nested ones by using
        i = sp1[0]*sp2[1] + sp1[1]*sp2[0];        the pointers sp1 and sp2.
        sp1[0] = fac*r;
        sp1[1] = fac*i;
        sp1 += 2;
        sp2 += 2;
    }
    sp1 = &speq1[1][1];
    sp2 = &speq2[1][1];
    for (j=1;j<=N*N;j++) {
        r = sp1[0]*sp2[0] - sp1[1]*sp2[1];
        i = sp1[0]*sp2[1] + sp1[1]*sp2[0];
        sp1[0] = fac*r;
        sp1[1] = fac*i;
        sp1 += 2;
        sp2 += 2;
    }
    rlft3(data1,speq1,N,N,N,-1);          Inverse FFT the product of the two FFTs.
/*    ...*/
    free_matrix(speq2,1,N,1,2*N);
    free_matrix(speq1,1,N,1,2*N);
    free_f3tensor(data2,1,N,1,N,1,N);
    free_f3tensor(data1,1,N,1,N,1,N);
    return 0;
}
```

To extend `rlft3` to four dimensions, you simply add an additional (outer) nested `for` loop in `i0`, analogous to the present `i1`. (Modifying the routine to do an *arbitrary* number of dimensions, as in `fourn`, is a good programming exercise for the reader.)

CITED REFERENCES AND FURTHER READING:

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

Swartztrauber, P. N. 1986, *Mathematics of Computation*, vol. 47, pp. 323–346.

12.6 External Storage or Memory-Local FFTs

Sometime in your life, you might have to compute the Fourier transform of a *really large* data set, larger than the size of your computer's physical memory. In such a case, the data will be stored on some external medium, such as magnetic or optical tape or disk. Needed is an algorithm that makes some manageable number of sequential passes through the external data, processing it on the fly and outputting intermediate results to other external media, which can be read on subsequent passes.

In fact, an algorithm of just this description was developed by Singleton [1] very soon after the discovery of the FFT. The algorithm requires four sequential storage devices, each capable of holding half of the input data. The first half of the input data is initially on one device, the second half on another.

Singleton's algorithm is based on the observation that it is possible to bit-reverse 2^M values by the following sequence of operations: On the first pass, values are read alternately from the two input devices, and written to a single output device (until it holds half the data), and then to the other output device. On the second pass, the output devices become input devices, and vice versa. Now, we copy *two* values from the first device, then *two* values from the second, writing them (as before) first to fill one output device, then to fill a second. Subsequent passes read 4, 8, etc., input values at a time. After completion of pass $M - 1$, the data are in bit-reverse order.

Singleton's next observation is that it is possible to alternate the passes of essentially this bit-reversal technique with passes that implement one stage of the Danielson-Lanczos combination formula (12.2.3). The scheme, roughly, is this: One starts as before with half the input data on one device, half on another. In the first pass, one complex value is read from each input device. Two combinations are formed, and one is written to each of two output devices. After this "computing" pass, the devices are rewound, and a "permutation" pass is performed, where groups of values are read from the first input device and alternately written to the first and second output devices; when the first input device is exhausted, the second is similarly processed. This sequence of computing and permutation passes is repeated $M - K - 1$ times, where 2^K is the size of internal buffer available to the program. The second phase of the computation consists of a final K computation passes. What distinguishes the second phase from the first is that, now, the permutations are local enough to do in place during the computation. There are thus no separate permutation passes in the second phase. In all, there are $2M - K - 2$ passes through the data.

Here is an implementation of Singleton's algorithm, based on [1]:

```
#include <stdio.h>
#include <math.h>
#include "nrutil.h"
#define KBF 128

void fourfs(FILE *file[5], unsigned long nn[], int ndim, int isign)
```

One- or multi-dimensional Fourier transform of a large data set stored on external media. On input, `ndim` is the number of dimensions, and `nn[1..ndim]` contains the lengths of each dimension (number of real and imaginary value pairs), which must be powers of two. `file[1..4]` contains the stream pointers to 4 temporary files, each large enough to hold half of the data. The four streams must be opened in the system's "binary" (as opposed to "text") mode. The input data must be in C normal order, with its first half stored in file `file[1]`, its second half in `file[2]`, in native floating point form. `KBF` real numbers are processed per buffered read or write. `isign` should be set to 1 for the Fourier transform, to -1 for its inverse. On output, values in the array `file` may have been permuted; the first half of the result is stored in `file[3]`, the second half in `file[4]`. N.B.: For `ndim` > 1, the output is stored by columns, i.e., *not* in C normal order; in other words, the output is the transpose of that which would have been produced by routine `fourn`.

```
{
    void fourew(FILE *file[5], int *na, int *nb, int *nc, int *nd);
    unsigned long j,j12,jk,k,kk,n=1,mm,kc=0,kd,ks,kr,nr,ns,nv;
    int cc,na,nb,nc,nd;
```