FFTEASY

C Routines for Fast Fourier Transforms of Real and Complex Data

http://www.science.smith.edu/departments/Physics/fstaff/gfelder/fftreasy/

Gary Felder (gfelder@email.smith.edu)

September 30, 2003

Contents

The documentation for FFTEASY is divided into four sections for people asking four different kinds of questions:

Section 1: What is this? This section is very short and everyone even thinking about using FFTEASY should read it. It contains a two paragraph description of what FFTEASY is and isn't good for and the obligatory paragraph about what you are and aren't allowed to do with FFTEASY. (Don't worry, it's completely free.)

Section 2: How do I use this? The section is the heart of the FFTEASY documentation. If you don't care about the inner workings of the routines but simply want to stick them in your code and start doing Fourier transforms with them then you can read this section and go no further. This section includes a list of the C functions included in FFTEASY and how to call them from your C program. It also describes the conventions used by FFTEASY for defining Fourier transforms and for storing their output in C arrays.

Section 3: That wasn't satisfying. Can you tell me more about what FFTEASY is doing? If this is you, then read this section for a description of how FFTEASY does its calculations. This makes for fun summer reading if you're a serious nerd. On a more practical level this section may be helpful if you want to modify the code in some way.

Section 4: Can you give me some examples? This section lists sample codes for calling the FFTEASY routines, including code for defining an array of complex numbers in frequency space with the correct symmetries for transforming to a real function using FFTEASY.

1 What FFTEASY Is and What It Isn't

FFTEASY is a series of C functions for doing Fast Fourier Transforms (FFTs) of real or complex data in any number of dimensions. The routines are designed to be short, readable, and easy to use. If you are comfortable programming in C and are somewhat familiar with FFTs it shouldn't take you more than an hour to read these instructions, link FFTEASY into your C program, and start doing Fourier transforms to your heart's content.

FFTEASY is not the fastest or highest precision FFT routine out there. If you want to do millions of transforms of large three-dimensional grids you probably want to find a better optimized set of routines. FFTEASY also requires that the number of data points in each dimension be a power of two. The advantages of FFTEASY are that it is simple to understand, simple to use, free, and capable of handling arbitrary dimensions of real or complex data. (In my experience most FFT routines handle either multi-dimensional complex data or one-dimensional real data, but for some reason there seem to be very few out there for multi-dimensional real data, which is what I actually wrote the routines for in the first place.)

FFTEASY consists of the four C functions fftc1, fftcn, fftr1, and fftrn. FFTEASY is free. I am not in any way, shape, or form expecting to make money off of these routines. I wrote them because I needed them for some work I was doing and I'm putting them out on the Internet in case other people might find them useful. Feel free to download them, incorporate them into your code, modify them, translate the comment lines into Swahili, or whatever else you want. What I do want is the following:

1. Leave this notice (i.e. this entire paragraph beginning with "FFTEASY consists of..." and ending with my email address) in with the code wherever you put it. Even if you're just using it in-house in your department, business, or wherever else I would like these credits to remain with it. This is partly so that people can...

2. Give me feedback. Did FFTEASY work great for you and help your work? Did you hate it? Did you find a way to improve it, or translate it into another programming language? Whatever the case might be, I would love to hear about it. Please let me know at the email address below.

3. Finally, insofar as I have the legal right to do so I forbid you to make money off of this code without my consent. In other words if you want to publish these functions in a book or bundle them into commercial software or anything like that contact me about it first. I'll probably say yes, but I would like to reserve that right.

For any comments or questions you can reach me at gfelder@email.smith.edu.

Enjoy FFTEASY!

Gary Felder

2 Using FFTEASY

FFTEASY is a set of functions designed to be placed inside a C program. Because these functions can only be used inside a program that you have presumably written yourself, I will assume that you are already familiar with the C language. If not I'm afraid having a set of C functions isn't going to do you much good.

The next section lists the functions that make up FFTEASY and describes their arguments. Subsequent sections talk about how to integrate FFTEASY into your code and describe the storage scheme used for the Fourier transforms, including the special issues that come up in storing Fourier transforms of real data. The section on storage also describes the Fourier transform conventions used in FFTEASY.

2.1 Overview of the Functions and Their Arguments

Here then are the four functions included in FFTEASY. I describe below what all the arguments are.

1. fftc1(float $f[\,]$, int N, int $skip$, int $forward$). FFT of a one-dimensional set of complex data.

2. fftcn(float $f[\,]$, int $ndims$, int $size[\,]$, int $forward$). FFT of an ndims dimensional set of complex data.
3. fftr1(float $f[\,]$, int N, int $forward$). FFT of a one-dimensional set of real data.
4. fftrn(float $f[\,]$, float $fnyquist[\,]$, int $ndims$, int $size[\,]$, int $forward$). FFT of an ndims dimensional set of real data.

All of these functions take an array $f[\,]$ of the data to be transformed and an integer $forward$ specifying whether to take a forward transform ($forward = 1$) or an inverse transform ($forward = -1$). The one-dimensional routines take an integer N giving the number of data points in the array. For fftr1 N is the number of real data points while for fftc1 N is the number of complex data points, meaning that for fftc1 f should be an array of floats of size $f[2N]$. The n dimensional routines take an integer $ndims$ giving the number of dimensions and an array of integers $size$ giving the number of data points (real or complex) in each dimension. For the complex routines the data should alternate between real and imaginary parts, i.e. $f = \{Re(n1), Im(n1), Re(n2), Im(n2), ...\}$. Note that the last dimension of the array f should always be twice as large as the actual number of data points to account for the fact that it takes two floats to represent a complex number. (See the examples in section 4). Finally the function fftrn takes an additional array $fnyquist$ for holding the Nyquist frequency components of the transform. I explain this below when I describe the storage scheme for the results of transforms of real data.

Note that even fftcn and fftrn take as input one dimensional arrays. Actually the functions assume that the data are arranged as in an n-dimensional array, so all you have to do is take your multidimensional array and cast it as a (float *) when you pass it to FFTEASY.

2.2 Using FFTEASY in Your Program

The simplest way to incorporate FFTEASY into your program is to compile and link the file ffteasy.c containing all of the FFTEASY functions into your program. Then all you will need to do is include declarations of the FFTEASY functions you wish to call. The declarations are collected for convenience at the top of the file ffteasy.c, from which you can paste them directly into your program.

If you want you can instead copy the FFTEASY functions out of the file ffteasy.c and include them directly in your code. If you do so please remember to include the credits/legal paragraph along with the code. Also, you will need to take account of the following dependencies:

1. All of the functions use the struct complex, which simply consists of two floats called *real* and *imag* respectively. The declaration of this struct is at the top of the ffteasy.c file.
2. The function fftc1 works by itself.
3. The functions fftcn and fftr1 both call fftc1, so if either of them are included in your code fftc1 must be included as well.
4. The function fftrn calls the function fftcn.

When calling the FFTEASY functions bear in mind that they all take an array of numbers and replace it with its Fourier transform. In other words the transform is done "in place," overwriting the original data. If you want to keep your original data you can: a) make a copy of it before doing the transform, or b) use FFTEASY to do the inverse transform when you're done, thus restoring your original data up to small numerical errors.

2.3 Storage: What Goes Where in the Arrays

My conventions for storing the output of real and complex Fourier transforms follows those given in Numerical Recipes (www.nr.com) with one important exception, noted below. You can look there for a more detailed description of the arrangement, but this documentation contains all the information you should need to use the FFTEASY routines. Before I describe this storage scheme, though, I should point out the one difference between the output of FFTEASY and Numerical Recipes, which is that the Numerical Recipes routines require you to divide the output of their inverse transforms by the number of data points. FFTEASY does this division for you so that the result you get from the FFTEASY routines represents the correct value of the transform, forward or inverted. In other words if you take an array of data and run it through one of the FFTEASY routines forward and then inverted you will get back your original data, which is not true of the Numerical Recipes routines.

2.3.1 Fourier Transform Definitions and Conventions

The Fourier transform $F(k)$ of a function $f(x)$ is defined as

$$F(k) = \int_{-\infty}^{\infty} f(x) e^{2\pi ikx} dx \tag{1}$$

$$f(x) = \int_{-\infty}^{\infty} F(k) e^{-2\pi ikx} dk. \tag{2}$$

A discrete Fourier transform takes a function $f(x)$ known only at N discrete points $f_a \equiv f(x_a)$ and gives back a function F_b known only at the discrete points k_b

$$F_b \equiv \sum_{a=0}^{N-1} f_a e^{2\pi iab/N} \tag{3}$$

$$f_a = \frac{1}{N} \sum_{b=0}^{N-1} F_a e^{-2\pi ab/N}. \tag{4}$$

Note that the relationship between the discrete Fourier transform and the continuous one is

$$F_b \approx \Delta x F(k_b) \tag{5}$$

where Δx is the spacing between points x_a and the spacing between frequencies k_b is given by

$$\Delta k = \frac{1}{N\Delta x}. \tag{6}$$

(Note that I am following the conventions of Numerical Recipes using k to denote frequency rather than angular frequency. The conversion is simply $\omega = 2\pi k$.) Formally the discrete Fourier transform F_b is periodic with period N so b can take any set of N consecutive values. Practically speaking, though, if the points x_a represent a typical region of the function $f(x)$ then the results F_b give the frequency components in the range $-N/2 \leq b \leq N/2$ (where $F_{-N/2} = F_{N/2}$ by periodicity). The next section describes how these complex points are arranged in the output of the routines fftc1 and fftcn. The following section describes the additional information needed to interpret the results of the real Fourier transform routines fftr1 and fftrn.

2.3.2 Output of Fourier Transforms of Complex Data

The output of fftc1 is arranged as follows. The first half of the output array represents the positive frequencies $0 \leq i \leq N/2$. The index i then wraps around to $-N/2$ and begins ascending from there. (Remember that $F_{N/2}$ and $F_{-N/2}$ are the same point.) Bearing in mind that the array contains alternating real and imaginary components, the input and output arrays look like:

$$Inputarray = I[2N] = \{Re(f_{x1}), Im(f_{x1}), Re(f_{x2}), ..., Im(f_{xN})\} \tag{7}$$

$$Outputarray = O[2N] = \{Re(F_0), Im(F_0), Re(F_1), ..., Re(F_{N/2}), Im(F_{N/2}), Re(F_{-N/2+1}), ..., Im(F_{-1})\} \tag{8}$$

Remember that I and O are actually the same array because the output data are written over the original input array. The output of fftcn is arranged analogously to fftc1. Each dimension goes from 0 to $N/2$ and then from $-N/2$ up to -1 and the whole array alternates between real and imaginary parts.

To do an inverse Fourier transform simply arrange the values F_k as described here. If the array $f[\,]$ was produced as output of the FFTEASY routines then it is already arranged correctly to be put back in for the inverse Fourier transform.

2.3.3 Output of Fourier Transforms of Real Data

For real arrays there is a subtlety in the arrangement of the output. Of course a real function is just a special case of a complex function and you could always take the Fourier transform of an array of real numbers by putting it in an array twice as big with every other component set to 0 and then submitting it to fftc1 or fftcn. This method is obviously quite wasteful. The point of a Fourier transform routine for real data is to take advantage of the fact that N independent input points only produce N independent output points, so even though the result $F(k)$ consists of complex numbers they can all be stored in the original array. The reason this can be done is because the Fourier transform of a real set of data obeys the symmetry $F(-k) = F^*(k)$, so it's redundant to store the negative frequency output. You still need the values $F(0)$ and $F(N/2)$ but these are both real so they only take up half as much storage as the other frequencies. The net result is that N real inputs produce N real outputs in the form of $N/2-1$ complex numbers and two real ones.

For fftr1 this makes the output scheme very simple. The first two numbers in the output array are the real values $F(0)$ and $F(N/2)$. The rest of the values contain alternate real and imaginary parts for $F(1)$ through $F(N/2-1)$.

For fftrn the story becomes more complicated. If I let k_{ndims} stand for the momentum in the last dimension of the array then I don't need to output any negative values of k_{ndims}; they can all be recovered by symmetry. On the other hand the values at $k_{ndims} = 0$ and $k_{ndims} = N/2$ are in general no longer real. It is still true that the number of independent outputs equals the number of inputs, but there is no obvious scheme for packing all of the outputs into the original array. Rather than coming up with a complicated scheme that would make it difficult to find different frequency components I simply let the entire plane of values $k_{ndims} = 0$ be represented in the original array and store all the values with $k_{ndims} = N/2$ in a separate array $fnyquist$. Thus if you look for the values $F(k_1, k_2, ..., 0)$ and $F(-k_1, -k_2, ..., 0)$ you will find them both in the output and they will be complex conjugates of each other.

So after all is said and done the output of fftrn is arranged with the last dimension going up from 0 to $N/2-1$ and all the others wrapping around as described for the complex routines above. The values with $k_{ndims} = N/2$ are stored in the array $fnyquist[N_1][N_2]...[2N_{ndims-1}]$.

If you have generated an array using fftr1 or fftrn then you can simply give that array back to the same function to do an inverse transform and recover your original data. If, however, you want to generate data in frequency space and then transform it to position space you have to be careful to ensure that your input obeys the symmetry $F(-k) = F^*(k)$. For fftr1 this simply means generating real numbers for F_0 and $F_{N/2}$ and putting them in the first two slots of the array and then filling all the subsequent slots with complex numbers. For fftrn, however, you have to make sure that for all the values with $k_n = 0$ (the first plane of values in your lattice) this symmetry is obeyed. You then have to fill fnyquist with the values with $k_n = N/2$, obeying the same symmetry conditions.

3 How FFTEASY works

FFTEASY uses a Cooley-Tukey Fast Fourier Transform algorithm. For an excellent discussion of Fourier transforms (continuous and discrete) in general and the Cooley-Tukey algorithm in particular see Numerical Recipes. Here I only briefly describe the algorithm and how it is implemented in FFTEASY.

The heart of the method is the Danielson-Lanczos (DL) formula that allows one to compute a discrete Fourier transform (DFT) of size N by separately computing two FTs of size $N/2$. Let F_b be the DFT of a function $f_a = f(x_a), a = 0, 1, ..., N-1$. Then let F_b^e be the DFT of the even points $a = 0, 2, ..., N-2$ and F_b^o be the DFT of the odd points $a = 1, 3, ..., N-1$. The DL formula then says

$$F_b = F_b^e + e^{2\pi i b/N} F_b^o. \tag{9}$$

For simplicity define $W \equiv e^{2\pi i/N}$ so

$$F_b = F_b^e + W^b F_b^o. \tag{10}$$

Note that F^e and F^o both have period $N/2$ while F has period N. This formula can be applied recursively so that F^e and F^o are each in turn calculated from two DFTs of size $N/4$. Assuming the original N was a power of two this process can be repeated all the way down to DFTs of single points, which are just identity operations (i.e. $F_b = f_a$ for a single point).

It turns out that rather than keeping track of successive divisions into even and odd points, the data in the original array can be rearranged in such a way that F^e and F^o are always sequential. In other words once the array has been rearranged in this way the DL formula can be used to combine the first two elements into an $N = 2$ DFT, and then combine that DFT with the succeeding $N=2$ DFT into an $N=4$ DFT, and so on. The way to rearrange the array to accomplish this is to arrange it in bit-reverse order. In other words if $N=4$ then the four elements of your original array are f_{00}, f_{01}, f_{10}, and f_{11}. To bit-reverse this array you would swap the middle two elements, leaving the first and last where they are. With some work you should be able to convince yourself that bit-reversal is equivalent to pulling out successive arrays of even and odd points. Otherwise you can take my word for it because I'm not going to prove it to you here.

The following sections describe each of the four FFTEASY functions. Note that all of the functions take as input an array of floats but immediately typecast it as an array of complex numbers, each consisting of two floats representing real and complex parts. So any time I refer to the ith element of an array I mean the ith *complex* element, corresponding to the $2i$ and $2i+1$ elements of the actual array of floats. Formulas are given here in terms of complex numbers and then implemented in the functions explicitly in terms of real and imaginary parts.

3.1 FFTC1: One-Dimensional Complex Fourier Transforms

Whether you prove them to yourself or take my word for them the formulas given above should be enough for you to follow what the routine fftc1 does. The first section takes an array of N points and rearranges them into bit-reversed order. This is done by noting that if you know the bit-reversal BR_b of a binary number b it is easy to find the bit-reversal BR_{b+1} of the number $b+1$. To add 1 to b you start at the right and replace every 1 with a 0 until you reach the first 0, which you replace with a 1. So to go from BR_b to BR_{b+1} you do the bit-reverse of this process, starting at the left and replacing every 1 with a 0 until you reach the first 0, which you replace with a 1.

The next part of the routine implements the DL formula on successively larger blocks, beginning with combining $N=1$ transforms into $N=2$ transforms. Each time the routine calculates a transform of size $trans\_size$ the first $trans\_size/2$ elements represent F_b^e and the next half represent F_b^o. These are then used to calculate F_b and $F_{b+trans\_size/2}$. Note that in the DL formula $F^e_{b+trans\_size/2} = F^e_b$, $F^o_{b+trans\_size/2} = F^o_b$, and $W^{b+trans\_size/2} = -W^b$, so the only difference between the formulas for F_b and $F_{b+trans\_size/2}$ is the sign of W^b. Finally, the inner loop over the variable $trans$ iterates over all the transforms of size $trans\_size$ in the entire array, so if you are transforming an $N=128$ array of data you first have to calculate 64 $trans\_size=2$ transforms, then 32 $trans\_size=4$ transforms, and so on.

Note that the formulas for the real and imaginary parts of wb are nothing but $\cos(2\pi b/N)$ and $\sin(2\pi b/N)$. They could have simply been calculated directly but for the sake of efficiency they are calculated instead using a recursion formula each time b is incremented. This formula can be verified simply using the addition rules for sines and cosines.

Note that the end result of this process is a transform with the results in sequential order from $b=0$ to $b=N-1$, which by the periodicity of F_b is exactly the storage scheme described in section 2.3.2.

The reverse transform is exactly identical except the sign of the exponential is reversed and the final output is divided by N.

The argument *skip* is used in case the elements of the input array are not sequential but evenly spaced in some larger array. To do a one-dimensional transform you would almost always want to set *skip* to one, but this option allows fftc1 to be used by fftcn, which does transforms of multi-dimensional arrays. For all but the last array index the one-dimensional transforms used by fftcn are not of sequential data.

3.2 FFTCN: Multi-Dimensional Complex Fourier Transforms

Actually once you understand fftc1 there is virtually nothing else left to fftcn. A multi-dimensional Fourier transform simply consists of successive one-dimensional transforms in every dimension. So all fftcn does is loop through the dimensions calling fftc1. The only tricky part about doing this is setting all the right skips correctly. For example, consider the middle dimension of a three dimensional array of size $\{N1, N2, N3\}$. Each DFT will be of length $N2$ and there will be $N1 \times N3$ of them. The skip between successive elements of the transform will be $N1$. There will be $N1$ transforms beginning at the elements $0, 1, ..., N1-1$ but the next one will begin at $N1 \times N2 - 1$. All of these stepsizes and skips are kept track of automatically by fftcn, and following the code itself is probably easier than following any explanation I could write of it.

Inverse transforms are done exactly the same as forward ones, only with all the calls to fftc1 being made for inverse transforms.

3.3 FFTR1: One-Dimensional Real Fourier Transforms

The trick to doing a DFT of real data is to once again use the DL formula. Specifically the routine begins by taking the original array and passing it as is to fftc1. Of course fftc1 treats the data as an array of complex numbers. Note, however, that the real parts of those numbers are the elements of F^e and the imaginary parts are the elements of F^o, so it is possible to extract the actual DFT of f from the results of fftc1. Let H_b represent the output of fftc1 and let F_b represent the desired output, i.e. the transform of the real function f. The input we gave to fftc1 was the combination $f^e + if^o$ so by the linearity of Fourier transforms we know that $H = F^e + iF^o$. From the DL formula we also know that $F_b = F_b^e + W^b F_b^o$. How can we extract F^e and F^o from H in order to plug them into the DL formula? The answer lies in the symmetry relation obeyed by the Fourier transforms of real data (which includes F^e and F^o), $F_{-b} = F_b^*$. So

$$H_b = F_b^e + iF_b^o \tag{11}$$

$$H_{-b} = F_b^{e*} + iF_b^{o*} \tag{12}$$

and thus

$$F_b^e = \frac{1}{2}\left(H_b + H_{-b}^*\right) \tag{13}$$

$$F_b^o = -\frac{i}{2}\left(H_b - H_{-b}^*\right) \tag{14}$$

Plugging these equations into the DL formula gives

$$F_b = \frac{1}{2}\left[H_b + H_{-b}^* - iW^b\left(H_b - H_{-b}^*\right)\right]. \tag{15}$$

Now all that's left is to figure out where all of these terms are stored in the array. Remember that all the array indices in these formulas are complex. In the case of F finding the elements is trivial because only non-negative values of k are given as output so the elements F_b are stored in order $b = 0, 1, ..., N/2 - 1$. (The case $b = N/2$ is treated separately below.) Recall, however, that H is the transform of a complex $N/2$ size array, so its elements range from $b = -N/4$ to $N/4$ in the wrap-around storage order described in section 2.3.2. In other words the value of H_{-b} is stored in the location $N/2 - b$. So

$$F_b = \frac{1}{2}\left[H_b + H_{N/2-b}^* - iW^b\left(H_b - H_{N/2-b}^*\right)\right]. \tag{16}$$

Plugging in $N/2 - b$ for b in this equation and noting that $W^{N/2-b} = \exp[2\pi i(N/2 - b)/N] = -(W^b)^*$ gives

$$F_{N/2-b} = \frac{1}{2}\left[H_b^* + H_{N/2-b} - i(W^b)^*\left(H_b^* - H_{N/2-b}^*\right)\right]. \tag{17}$$

Using these two equations the program steps over all the elements of the array overwriting H_b and $H_{N/2-b}$ with F_b and $F_{N/2-b}$. The cases $b = 0$ and $b = N/2$ are a little different, however, because F_0 and $F_{N/2}$ are both real. The function fftr1 places F_0 in the first (real) slot of the $b = 0$ element of the array and $F_{N/2}$ in the second (imaginary) slot. From the formulas above, noting that $H_{N/2} = H_0$ by periodicity, we see that

$$F_0 = Re(H_0) + Im(H_0) \tag{18}$$

$$F_{N/2} = Re(H_0) - Im(H_0). \tag{19}$$

Inverting this process seems like a mess but actually works out to be nearly identical. I won't reproduce all the algebra here but if you invert the formulas above to find H from F you will find that except for a couple of minus signs and a factor of two for the $k = 0$ and $k = N/2$ terms the results look identical to the forward case. As a result the inverse transform simply uses the same loop to rearrange the array and then calls fftc1 to do an inverse transform and get the function $f(x)$ back.

3.4 FFTRN: Multi-Dimensional Real Fourier Transforms

The structure of fftrn is almost identical to fftr1 and the basic equations used are the same. The key difference comes from the fact that in more than one dimension the element H_{-b} can no longer simply be found at the location $N/2-b$. Rather the index must be reversed from b to $-b$ separately for each dimension. Likewise in the one-dimensional case F_b only needed to include non-negative value of b whereas in this case that is true only in the last dimension.

This indexing is particularly tricky since fftrn treats the data as a one-dimensional array. (There is no practical way to treat the data as a multi-dimensional array without knowing in advance how many dimensions are going to be used.) Fortunately there is a simple solution to this problem. The function fftrn defines an array called *indices* that keeps track of the separate indices in all the dimensions. At the bottom of the main loop these indices are incremented, advancing the last one until it reaches its maximum and then starting it over and incrementing the next to last one, and so on. At the top of the main loop is a call to the function find_index, which finds the one-dimensional array index that corresponds to these different multi-dimensional indices. More importantly, the function find_index also finds the one-dimensional index corresponding to the negative of the frequencies indicated by *indices*. From there the implementation is almost identical to fftr1. The inner loop runs over the last dimension and applies the exact same formulas as fftr1 to calculate F, the desired transform, from H, the output of fftcn. I leave it as an exercise for the reader to show that these formulas are still valid provided the frequencies in all but the last dimension are replaced by their corresponding negative frequencies as shown.

The 0 and $N/2$ (Nyquist) frequencies present a problem. In one dimension these both had to be real because $F(k) = F(-k)^*$. In multiple dimensions, however, it is no longer true that all values of F for which the last index are 0 or $N/2$ are real. To illustrate what I mean let me use k to indicate the frequency in the last dimension and p to indicate the frequency components in all other dimensions. Now the symmetry is simply $F(p,0) = F(-p,0)^*$. This means that applying equation [16] to the case $k = 0$ will no longer yield a real number as it did in the one-dimensional case and so $k = N/2$ can no longer be packaged with $k = 0$ into one complex array index. So fftrn takes as input an array $fnyquist$ that it uses for the values $k = N/2$ and uses both the real and imaginary parts of the first elements of the array f for $k = 0$. In practice this means that the output is redundant because it will contain $f(p,0)$ and $f(-p,0)$, which are complex conjugates of each other (and likewise for p and $-p$ in $fnyquist$).

As with fftr1 the algebra for the inverse transform turns out to be virtually identical to the forward case, and once again I leave the details to the reader to verify. Note that it is up to the user to make sure that the symmetries noted above are obeyed by the input to the inverse transform. In section [4.3] I provide sample code for generating data with all the correct symmetries in three dimensions. You can simply cut and paste this sample code into your own program and substitute whatever function you want to generate in frequency space.

4 Sample Codes for Using FFTEASY

4.1 One-Dimensional Forward Transform of Complex Data

This code generates an array with the values $f(x) = e^{-2\pi i D x/N}$. The resultant transform has value $F_D = N$ and all the other values equal to zero.

```
#define N 16 // Number of complex points in the data
#define D 5
void main()
{
  int a;
  float f[2*N]; // An array twice as large as the number of complex points
  float pi=2.*asin(1.),x=2.*pi*D/(float)N;

  for(a=0;a<N;a++)
  {
    f[2*a]=cos(x*(float)a); // f(i) = e^(-2 pi i D a/N)
    f[2*a+1]=-sin(x*(float)a);
  }

  fftc1(f,N,1,1);

  for(a=0;a<N;a++)
    printf("%d %f %f\n",a,f[2*a],f[2*a+1]);
}
```

```
OUTPUT:

0 0.000000 0.000001
1 0.000001 0.000001
2 0.000001 0.000001
3 0.000001 0.000001
4 0.000003 0.000001
5 15.999999 -0.000008
6 -0.000003 0.000001
7 -0.000001 0.000001
8 -0.000001 0.000001
9 -0.000001 0.000001
10 -0.000000 0.000001
11 -0.000000 0.000001
12 -0.000000 0.000001
13 0.000001 -0.000001
14 0.000000 0.000001
15 0.000000 0.000001
```

4.2 Multi-Dimensional Forward Transform of Real Data

This code illustrates how to find the different momentum components in the output of fftrn. The initial array of size ($N1$,$N2$) is generated with only two sinusoidal modes excited, the real part of $\vec{k} = (3, 1)$ and the imaginary part of $\vec{k} = (2, 0)$. The loop at the end shows how to find the different momenta. The excitation $(3, 1)$ only shows up once but the mode $(2, 0)$ and its complex conjugate $(-2, 0)$ are redundantly included in the output. Note that no matter what you put in as input the output components $F_{0,0}$, $F_{0,N2/2}$, $F_{N1/2,0}$ and $F_{N1/2,N2/2}$ will be real.

```
#define N1 8 // Number of real points in the data - first dimension
#define N2 4 // Second dimension
void main()
{
  int a,b,ka,kb;
  float f[N1][N2]; // An array of real numbers
  float fnyquist[2*N1]; // For storing k_2 = N2/2 frequencies
  int size[2]={N1,N2}; // Size of data in all dimensions
  float pi=2.*asin(1.),x1=2.*pi/(float)N1,x2=2.*pi/(float)N2;

  for(a=0;a<N1;a++)
    for(b=0;b<N2;b++)
      f[a][b]=5.*cos(3.*x1*(float)a + x2*(float)b)+sin(2.*x1*(float)a);

  fftrn((float *)f,(float *)fnyquist,2,size,1);

  for(a=0;a<N1;a++)
  {
    ka = (a<=N1/2 ? a : a-N1); // ka is the frequency stored at position a
    for(b=0;b<N2/2;b++)
    {
      kb = b; // kb is the frequency stored at position b
      printf("Frequency={%d,%d}, Transform value={%f + i %f}\n",ka,kb,f[a][2*b],f[a][2*b+1]);
    }
    printf("Frequency={%d,%d}, Transform value={%f + i %f}\n",ka,N2/2,fnyquist[2*a],fnyquist[2*a+1]);
  }
}
```

```
OUTPUT:

Frequency={0,0}, Transform value={-0.000002 + i 0.000000}
Frequency={0,1}, Transform value={0.000000 + i 0.000003}
...
Frequency={2,0}, Transform value={0.000002 + i 15.999999}
...
Frequency={3,1}, Transform value={79.999992 + i -0.000023}
...
Frequency={-2,0}, Transform value={0.000002 + i -15.999999}
...
Frequency={-1,1}, Transform value={0.000004 + i 0.000004}
Frequency={-1,2}, Transform value={-0.000000 + i 0.000001}
```

4.3 Multi-Dimensional Inverse Transform of Real Data

These are the hardest transforms to do because the symmetry $F(-k) = F(k)^*$ has to be put into the input array by hand for the modes where the frequency component in the last dimension is 0 or the Nyquist frequency $N/2$. If the data you are using were generated from a forward transform then these symmetries should already be in place and you don't have to worry about them, but if you want to generate your own multi-dimensional data in frequency space and then inverse transform them to a real function you have to be careful. This code initializes a three-dimensional array of complex numbers satisfying the necessary symmetries. Since this is a fairly common problem I invite you to simply copy this code verbatim and alter the lines where the values are set. There are eight such lines, all marked with the comment **Customize Here**. I also marked the line where the lattice spacing dx is set since this is a physical parameter that has to be set for each problem (as are the sizes $N1$, $N2$, and $N3$ of course).

This particular code initializes the function F to

$$F(\vec{k}) = \frac{\exp[ik_x]}{\exp[|\vec{k}|^2]}. \tag{20}$$

The choice of function was basically arbitrary, but was guided by the following considerations. It falls off rapidly at large k. This is good because the transform will have no information about large frequencies so if they are important the resultant data will probably be meaningless, or at least skewed. Also, this function does explicitly satisfy the symmetry condition, so if I were to take the inverse Fourier transform of this continuous function I would indeed get a real result.

This code doesn't produce any output because I couldn't think of anything useful to illustrate with the output. The point of the code was to illustrate how to set up the data with the correct symmetries.

```
#define N1 8 // Number of real points in the data - first dimension
#define N2 4 // Second dimension
#define N3 4 // Third dimension
void main()
{
  int a,b,c;
  int aconj,bconj; // Positions of the frequencies -a and -b on the lattice
  float f[N1][N2][N3]; // An array of real numbers
  float fnyquist[N1][2*N2];
  int size[]={N1,N2,N3};
  float ka,kb,kc,k2; // Frequency in each dimension and square magnitude of frequency
  // The spacing dx is given as a parameter of the problem
  // The spacing dk then gives the space between adjacent frequencies on the lattice
  float dxa=3.5,dxb=2.3,dxc=1.4; **Customize Here**
  float dka=1./dxa/(float)N1,dkb=1./dxb/(float)N2,dkc=1./dxc/(float)N3;

  for(a=0;a<N1;a++) // Loop over first dimension
  {
    aconj=(a==0 ? 0 : N1-a); // Position on the lattice of frequency -a
    ka=(a<=N1/2 ? a : a-N1); // Value of frequency in first dimension
    ka *= dka; // Convert from index value to actual frequency
    for(b=0;b<N2;b++) // Loop over second dimension
    {
      kb=(b<=N2/2 ? b : b-N2); // Value of frequency in second dimension
      kb *= dkb;
      // Set all modes with 0<c<N3/2. The complex conjugates of these modes do not need to be set.
      for(c=1;c<N3/2;c++) // Main loop over last dimension
      {
        kc=c; // Value of frequency in last dimension
        kc *= dkc;
        k2 = ka*ka + kb*kb + kc*kc; // Set square magnitude of frequency
```

```
        f[a][b][2*c]=cos(ka)/exp(k2); // Re(F_a,b,c) = cos(kx)/e^(k^2) **Customize Here**
        f[a][b][2*c+1]=sin(ka)/exp(k2); // Im(F_a,b,c) = sin(kx)/e^(k^2) **Customize Here**
      }
      // Set modes with c=0 or c=N3/2.
      // The complex conjugates of these modes appear explicitly on the lattice
      //   and must be set to satisfy F(-k)=F(k)*
      if(b>N2/2 || (a>N1/2 && (b==0 || b==N2/2))) // Set each mode only once
      {
        bconj=(b==0 ? 0 : N2-b); // Position on the lattice of frequency -b
        // c=0
        k2 = ka*ka + kb*kb; // Square magnitude of frequency
        f[a][b][0]=cos(ka)/exp(k2); // Re(F_a,b,0) = cos(kx)/e^(k^2) **Customize Here**
        f[a][b][1]=sin(ka)/exp(k2); // Im(F_a,b,0) = sin(kx)/e^(k^2) **Customize Here**
        f[aconj][bconj][0]=f[a][b][0]; // Set complex conjugate mode
        f[aconj][bconj][1]=-f[a][b][1];
        // c=N3/2
        kc = N3/2.*dkc;
        k2 = ka*ka + kb*kb + kc*kc; // Square magnitude of frequency
        fnyquist[a][2*b]=cos(ka)/exp(k2); // Re(F_a,b,N3/2) = cos(kx)/e^(k^2) **Customize Here**
        fnyquist[a][2*b+1]=sin(ka)/exp(k2); // Im(F_a,b,N3/2) = sin(kx)/e^(k^2) **Customize Here**
        fnyquist[aconj][2*bconj]=fnyquist[a][2*b]; // Set complex conjugate mode
        fnyquist[aconj][2*bconj+1]=-fnyquist[a][2*b+1];
      }
      // The 8 "corners" of the lattice are set to real values
      else if((a==0 || a==N1/2) && (b==0 || b==N2/2))
      {
        k2 = ka*ka + kb*kb; // Square magnitude of frequency for c=0
        f[a][b][0]=cos(ka)/exp(k2); // Re(F_a,b,0) = cos(kx)/e^(k^2) **Customize Here**
        f[a][b][1]=0.; // Imaginary part is set to 0
        kc = N3/2.*dkc;
        k2 = ka*ka + kb*kb + kc*kc; // Square magnitude of frequency for c=N3/2
        fnyquist[a][2*b]=cos(ka)/exp(k2); // Re(F_a,b,N3/2) = cos(kx)/e^(k^2) **Customize Here**
        fnyquist[a][2*b+1]=0.; // Imaginary part is set to 0
      }
    } // End of b loop (second dimension)
  } // End of a loop (first dimension)

  fftrn((float *)f,(float *)fnyquist,3,size,-1); // Inverse transform of 3-D data
}
```