PHYSICS P411/P610: Fourier Transforms

Introduction | Analytic | Discrete | FFT

Introduction

See *NR* chapter 12 - also the Wolfram Mathworld and Wikipedia articles on Fourier Transforms.

Many physical processes can be described in more than one *domain*. For example:

1. Analysis of a signal as a function of time, *t* or frequency *f*.
2. Representation of a wavefunction in position-space, *x* or momentum-space *p*.

Different calculations can be simplified by choosing to do them in the appropriate domain. So we will want to develop efficient techniques for translating functions back and forth between the domains.

back to top

Analytic Fourier Transforms

See *NR* section 12.0.

Fourier transforms between the time and frequency domains can be written:

$$H(f) = \int_{-\infty}^{+\infty} h(t)\, e^{2\pi i f t}\, dt \equiv F[h](f)$$

$$h(t) = \int_{-\infty}^{+\infty} H(f)\, e^{-2\pi i f t}\, df \equiv F^{-1}[H](t)$$

Note that normalization coefficients of *1/2π* or *1/√(2π)* are often included in these definitions depending on the nature of the variables being used.

Properties of Fourier Transforms

Fourier Transforms have many useful symmetry and transform properties, which you can read about in *NR*, Wikipedia, and other sources. We will list only a few that will be useful to keep in mind in our discussions later.

h(t) is real $\Rightarrow$ $H(-f) = H(f)^*$

h(t) is imaginary $\Rightarrow$ $H(-f) = -H(f)^*$

Power Spectral Density

It's often useful to quantify the "strength" of a frequency component of a Fourier-transformed signal. The basis of this comparison rests on *Parseval's theorem*, which states that the total power in a signal is the same regardless of whether we compute it in the time or frequency domain.

$$P_{tot} = \int_{-\infty}^{+\infty} |h(t)|^2\, dt = \int_{-\infty}^{+\infty} |H(f)|^2\, df$$

We can prove this mathematically, using the fact that:

$$\delta(a-b) = \int_{-\infty}^{+\infty} e^{-2\pi i (a-b) t}\, dt$$

The power in the frequency intervals [*f, f+df*] and [*−f, −f+df*] is:

$$P_h(f) \equiv |H(f)|^2 + |H(-f)|^2 \quad (0 \le f < \infty)$$

This is referred to as the *one-sided power spectral density* (PSD) of the function *h*. It usually includes contributions from both positive and negative frequencies with magnitude *f*, but not always, so it pays to be

careful when interpreting quoted PSD values.

back to top

Discrete Fourier Transforms (DFT)

See *NR* section 12.1

Suppose that we sample the function *h(t)* at *N* discrete, uniformly space points and attempt to Fourier transform this data. Evidently, the *N* input points will allow us to find only *N* independent output points in Fourier space. As we will discuss in more detail below, the frequencies we can acces are limited to $|f| \le f_c$ *(= 1/2Δ)*. We then have (for *N* even):

$$h_k \equiv h(t_k) \quad , \quad t_k \equiv k\Delta \quad , \quad k = 0, 1, 2, ..., N-1$$
$$H_n \approx H(f_n)/\Delta \quad , \quad f_n \equiv n/N\Delta \quad , \quad n = -N/2, ..., 0, ..., N/2$$

where,

$$H(f_n) = \int_{-\infty}^{+\infty} e^{2\pi i f_n t} h(t)\, dt \approx \sum_{k=0}^{N-1} e^{2\pi i kn/N} h_k \Delta$$
$$H_n \equiv \sum_{k=0}^{N-1} e^{2\pi i kn/N} h_k$$

The last equation is the Discrete Fourier Transform (DFT).

Note: despite the fact that there are *N+1* terms in the f_n series above, only *N* of them are independent since $H_{-N/2} = H_{N/2}$.

It's usually convenient to define the Fourier Transformed index, *n* to run from 0 to *N−1*, rather than *−N/2* to *N/2*. This can be accomplished by realizing that the DFT is periodic, with period *N*:

$$H_{-n} = H_{N-n}$$

Thus, we can remap the Fourier components as:

$H_{-N/2}$	$H_{-N/2+1}$	.....	H_{-1}	H_0	H_1	.....	$H_{N/2-1}$	$H_{N/2}$

as:

H_0	H_1	.....	$H_{N/2-1}$	$H_{N/2} = H_{-N/2}$	$H_{-N/2+1}$	.....	H_{-1}
[0]	[1]	.....	[N/2−1]	[N/2]	[N/2+1]	.....	[N−1]

and write the forward and inverse DFTs as:

$$H_n = \sum_{k=0}^{N-1} h_k\, e^{2\pi i k n/N} \approx \frac{F[h](f_n)}{\Delta}$$
$$h_k = \frac{1}{N}\sum_{n=0}^{N-1} H_n\, e^{-2\pi i k n/N} \approx F^{-1}[H](t_k)\, \Delta$$

Power Spectral Density

The discrete form of Parseval's theorem is:

$$P_{tot} = \sum_{k=0}^{N-1} |h_k|^2 = (1/N) \sum_{n=0}^{N-1} |H_n|^2$$

The Power Spectral Density of a DFT is not an entirely well-defined quantity, but we can write an approximation of the continuous case that will work for our purposes: comparing signal strength at different frequencies (see *NR* sect. 13.4 for more details)

$$P_h(f_n) = (|H_n|^2 + |H_{N-n}|^2)/N^2 \qquad (1 \le n \le N/2 - 1)$$
$$\equiv |H_0|^2/N^2 \qquad (n = 0)$$
$$\equiv |H_{N/2}|^2/N^2 \qquad (n = N/2)$$

Aliasing

As mentioned previously, the act of sampling the function *h(t)* at a finite number of points, over a finite range, can cause some problems. For a sampling interval, Δ, the *Nyquist critical frequency*:

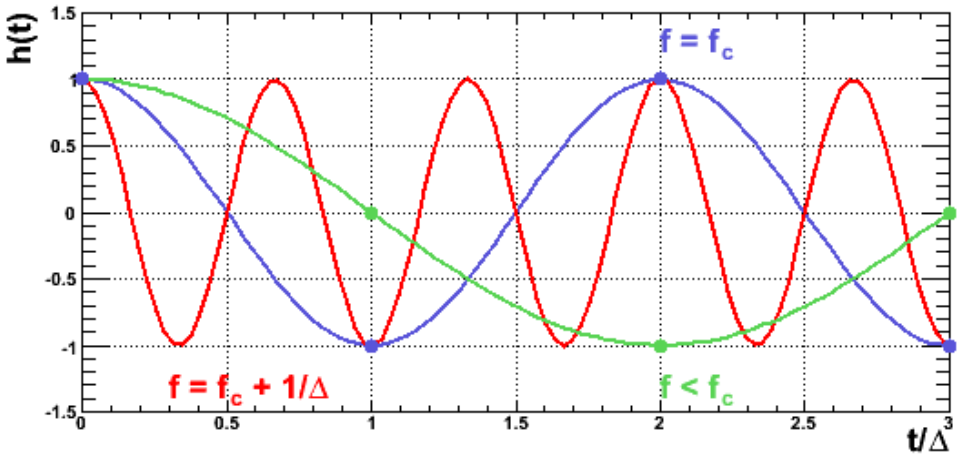
$$f_c = 1/2\Delta$$

defines the frequency range to which the DFT is sensitive:

$$-f_c \le \quad \le f_c$$

This is simply because it takes at least two sample (at the peak and at the trough) to define a sine wave.

If *h* contains frequency components with magnitude greater than f_c, their spectral power will be *aliased* back into the region, $[-f_c, f_c]$. Any frequency component greater than f_c produces identical samples to the frequency $f - m/\Delta$ as is illustrated below.



If, on the other hand, *h* does *not* contain any frequency components with magnitude greater than f_c, then *h(t)* is *completely determined* by its samples, h_n. This application of the sampling theorem suggests a good check on any DFT we do.

Always verify that the *H(f)* you find approaches zero as $|f| \to f_c$

back to top

Fast Fourier Transforms (FFT)

See *NR* section 12.2

If we re-write the discrete Fourier transform as:

$$H_n = \sum_{k=0}^{N-1} W_N^{n\,k}\, h_k$$
$$W_N \equiv e^{2\pi\, i\, /\, N}$$

then we can view the Fourier Transform as the multiplication of:

- an $N \times N$ matrix, whose elements are $W_N^{n\,k}$, by
- an N-dimensional vector, whose elements are h_k

doing this calculation requires N^2, complex number, operations.

It turns out, though, that there is a cleverer way - the *Fast Fourier Transform*. This technique became common in the 1960's because of the work of Cooley and Tukey [1], but variants of it have been available since Gauss' time. Following *NR* and *CSM*, the one we will consider is based on the Danielson and Lanczos lemma [2].

Danielson-Lanczos Algorithm

If N is even, then we can split the DFT above into k = even, and k = odd parts.

$$H_n = \sum_{k=0}^{N-1} W_N^{n\,k}\, h_k$$

$$= \sum_{k=0}^{N/2-1} e^{2\pi i\, n\,(2k)\,/\,N}\, h_{2k} + \sum_{k=0}^{N/2-1} e^{2\pi i\, n\,(2k+1)\,/\,N}\, h_{2k+1}$$

$$= \sum_{k=0}^{N/2-1} e^{2\pi i\, nk\,/\,(N/2)}\, h_{2k} + W_N^{\,n} \sum_{k=0}^{N/2-1} e^{2\pi i\, nk\,/\,(N/2)}\, h_{2k+1}$$

$$= \sum_{k=0}^{N/2-1} W_{N/2}^{n\,k}\, h_{2k} + W_N^{\,n} \sum_{k=0}^{N/2-1} W_{N/2}^{n\,k}\, h_{2k+1}$$

$$\equiv H_n^e + W_N^n\, H_n^o$$

Thus, we can find H_n from two DFTs of length $N/2$, rather than one DFT of length N. (We win here because of the N^2 operations that are required to do a single DFT.)

Note that splitting the original sum into even and odd parts is special. Simply dividing the terms arbitrarily (for example into $k_1 = 0...N/2-1$ and $k_2 = N/2...N-1$ does not end up giving us the sum of two DFTs because we don't get a factor of $N/2$ in the denominator of the argument of the exponents to match the $N/2$ terms in the two sums. (Try writing this yourself...)

We should also note that: H_n^e, H_n^o are periodic, with a period $N/2$.

$$H_{n+N/2}^{e,o} = H_{n+N/2}^{e,o}$$

$$W_N^{n+N/2} = W_N^n\, e^{2\pi i\,(N/2)\,/\,N} = -W_N^n$$

$$H_n = H_n^e + W_N^n\, H_n^o$$

$$H_{n+N/2} = H_n^e - W_N^n\, H_n^o$$

which buys us another factor of approximately 2 compared to the naive DFT. But the Danielson-Lanczos lemma allows us to do even better than that if N is a power of two.

$$N = 2^m \quad (m = \text{integer})$$

In that case, we can continue dividing the sums by two until we reach sums of length one. This requires $\log_2 N$ decimations.

As an example, division of the sums H_n^e and H_n^o into even and odd components of length $N/4$ gives:

$$H_n = H_n^{ee} + W_{N/2}^n\, H_n^{eo} + W_N^n\,[\, H_n^{oe} + W_{N/2}^n\, H_n^{oo}\,]$$

$$H_n^{ee} = \sum_{k=0}^{N/4-1} e^{2\pi i\, nk\,/\,(N/4)}\, h_{4k}$$

$$H_n^{eo} = \sum_{k=0}^{N/4-1} e^{2\pi i\, nk\,/\,(N/4)}\, h_{4k+2}$$

$$H_n^{oe} = \sum_{k=0}^{N/4-1} e^{2\pi i\, nk\,/\,(N/4)}\, h_{4k+1}$$

$$H_n^{oo} = \sum_{k=0}^{N/4-1} e^{2\pi i nk / (N/4)} h_{4k+3}$$

Continuing on to sums of length one leaves us with N terms:

$$H_n^{eee...ee} = h_0$$
$$H_n^{oee...ee} = h_1$$
$$.....$$
$$H_n^{ooo...oo} = h_{N-1}$$

where the left-most element in the *eeoe...oee* pattern corresponds to the $N \rightarrow N/2$ split, the next element to the $N/2 \rightarrow N/4$ split, etc. Note that these 1-point transforms are *independent* of n. So they are the same for all points in the Fourier-Transformed space.

Which pattern of *e*'s and *o*'s correspond to which h_k can be obtained by the following neat trick.

1. Represent the pattern *eoeeeo...oee* as a binary number using *e=0, o=1*.
2. Bit-reverse the number.
3. k is just this bit-reversed number

This works because division a splits the numbers in the sample being divided into parts where bit a of the numbers being considered is:

0 : even division
1 : odd division

Since the pattern of *e*'s and *o*'s starts on the left, we have to reverse that order to get the original number.

The total number of operations required to do the Fourier Transform then scales as $N \log_2 N$. For large N, this is clearly a big improvement over the N^2 behavior of the basic DFT implementation.

N = 8 Example [3]

The decimation process is shown below, which illustrates the requirement of bit-reversing.

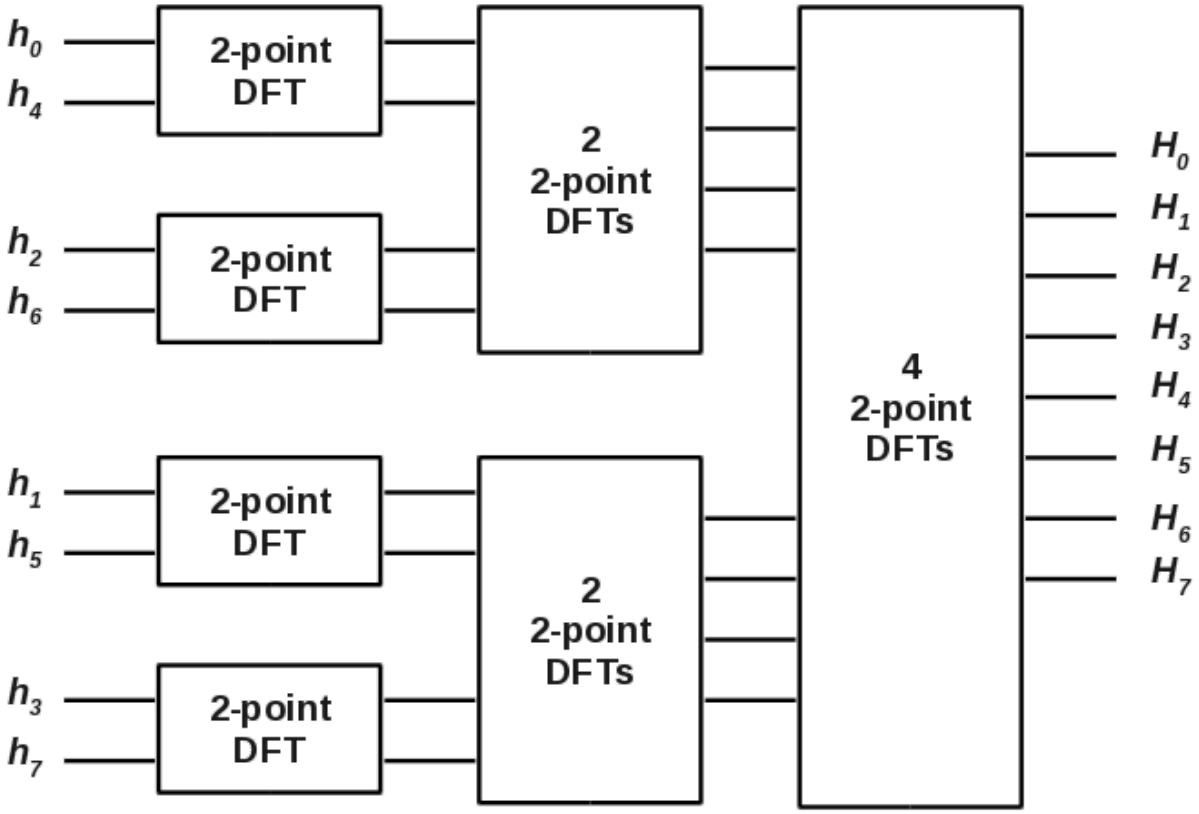
Orig	Step 0	Step 1	Step 2
0 (000)	0 (000)	0 (000)	0 (000)
1 (001)	2 (010)	4 (100)	4 (100)
2 (010)	4 (100)	2 (010)	2 (010)
3 (011)	6 (110)	6 (110)	6 (110)
4 (100)	1 (001)	1 (001)	1 (001)
5 (101)	3 (011)	5 (101)	5 (101)
6 (110)	5 (101)	3 (011)	3 (011)
7 (111)	7 (111)	7 (111)	7 (111)

Bit-reversal is shown below as are the two swaps that are required to re-order the initial h's to get the bit-reversed ordering.

eee =	000	→	000	(h_0)	000	
eeo =	001	→	100	(h_4)	001	↖
eoe =	010	→	010	(h_2)	010	|
eoo =	011	→	110	(h_6)	011	| ↖

oee	=	100	→	001	(h_1)	100	↙ |
oeo	=	101	→	101	(h_5)	101	|
ooe	=	110	→	011	(h_3)	110	↙
ooo	=	111	→	111	(h_7)	111	

The stages in the construction of the Fourier coefficients using the Danielson-Lanczos lemma is shown schematically below.



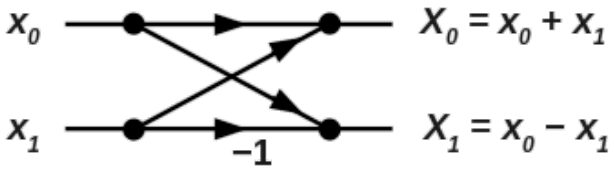
Using the general form of the DFT:

$$X_n = \sum_{k=0}^{N-1} e^{2\pi i nk / N} x_k$$

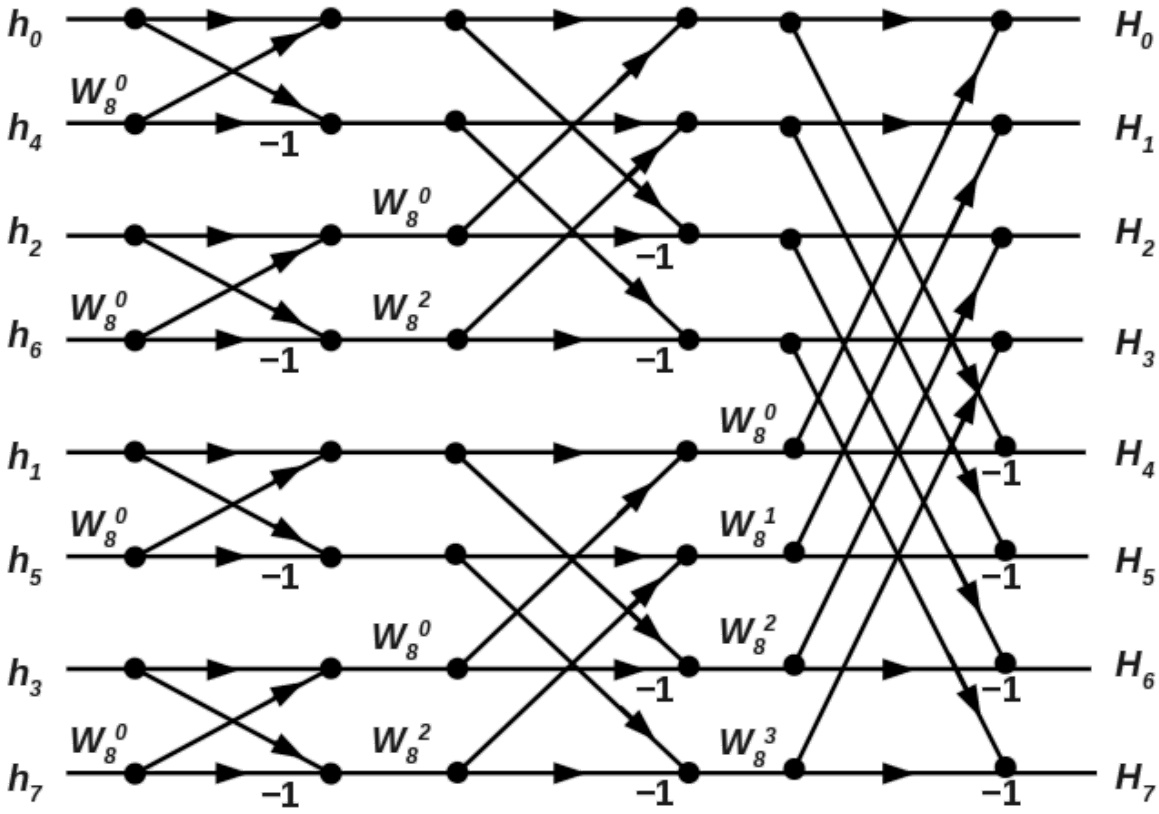
each 2-point DFT in the above diagram can be written as:

$$X_0 = x_0 + x_1$$
$$X_1 = x_0 - x_1$$

This operation is often referred to as a *butterfly* as can be seen from the illustration below.



The $N = 8$ case then becomes.



For example,

$$H_1 = (h_0 - h_4) + W_8^2 (h_2 - h_6) + W_8^1 [(h_1 - h_5) + W_8^2 (h_3 - h_7)]$$

Implementation in Code

The example above illustrates how the FFT can be implemented efficiently in code.

1. Put indices of original input arrays (real and imaginary components of *h*) into bit-reversed order
   a. choose each pair to swap
2. Loop over $\log_2 N$ stages of FFT
   a. determine pairs of indices for butterfly for this stage
   b. calculate $W_{N/d}$, where *d* = number of decimations that have been done at this stage
   c. Loop over groups of butterflies with different powers of $W_{N/d}$ multiplying the 2nd element
      i. Loop over individual butterflies in each group
         - calculate 1st element of butterfly: $x_0 = x_0 + W^i x_1$
         - calculate 2nd element of butterfly: $x_1 = x_0 - W^i x_1$
   d. Update to next power of *W*

This algorithm requires of order $N \log_2 N$ operations to complete. Code for it can be found in

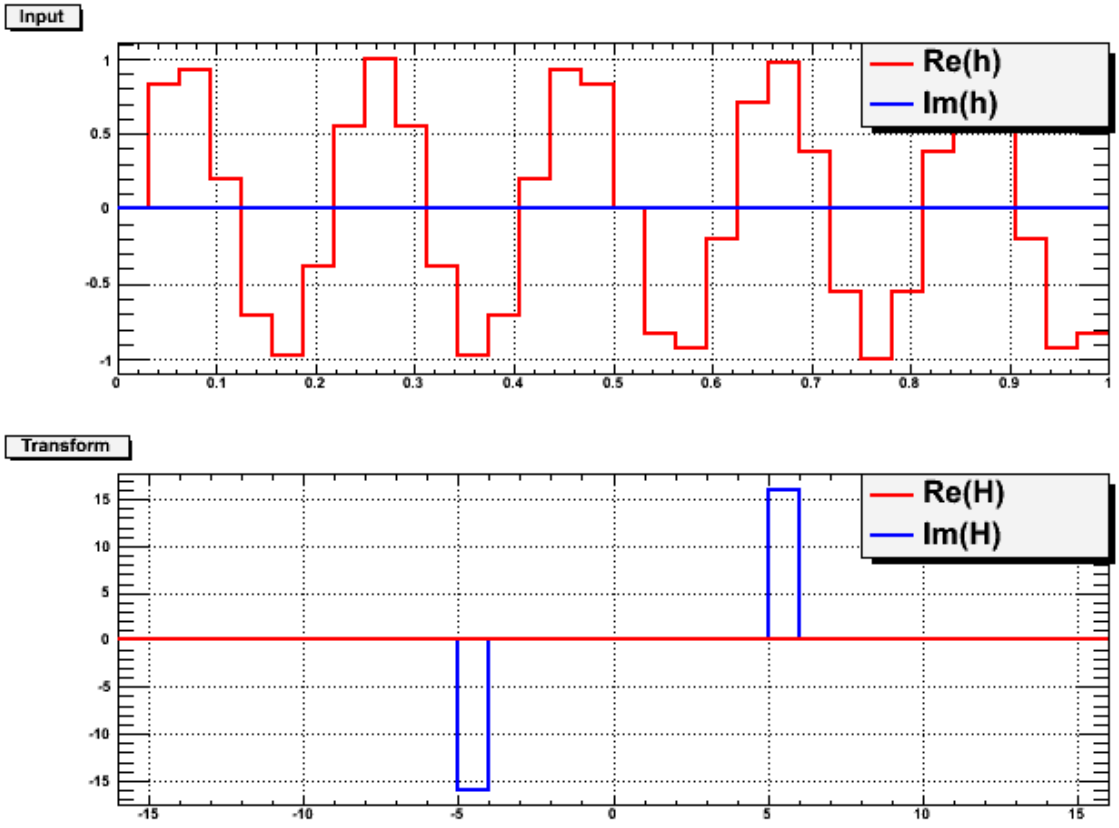
```
$CPCOMMON/.../cpFourier
```

The code in `cpFourier` is, by no means, as optimal as it could be. If you plan to use Fourier Transforms as part of your research, then you should use an optimized FFT package like FFTW [4].

Testing the Code

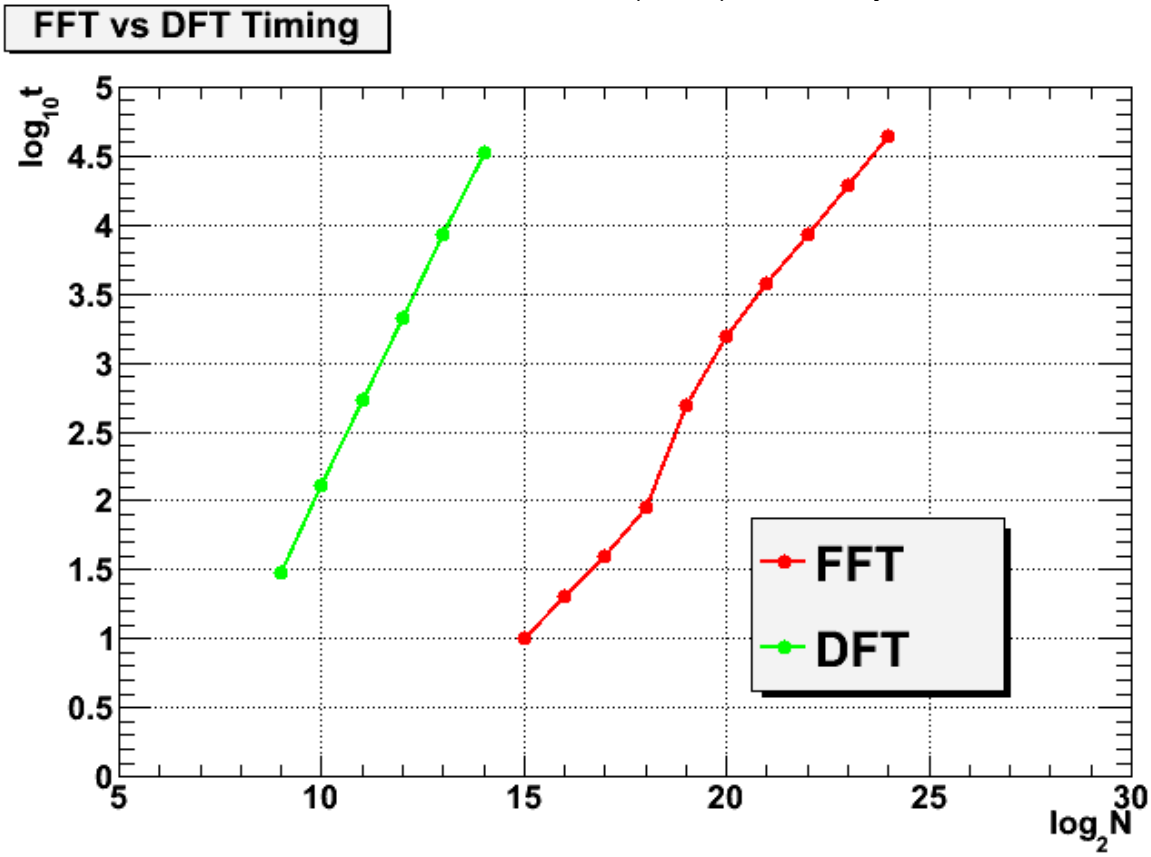
We'll look at a couple of basic tests of the FFT algorithm that's coded in *cpFourier*.

1. First, let's just verify that it works, by doing a transform of the function:
   $h(t) = \sin(2\pi f t)$
   The result, for $f = 5$, is given below.



2. We should also check that: $F^{-1}[\,F[h](t)\,] = h(t)$

3. Finally, let's see how much we gain by using the FFT rather than a normal DFT. Below is a plot of the amount of CPU time taken to do FFTs and DFTs of $sin(\,2\pi\,f\,t\,)$ for a variety of values of N.



Obviously, we gain a huge amount by using the FFT.

Running the Code

Code for the above tests can be found in: `$CPHOME/examples/PDE/`

```
# h(t) to H(f) Transform
[evans@grendel PDE]$ ./ft 0 32 1 5 0 | tee fft.out
root [0] .x ft_dist.C
root [1] canv->Print("FFT_f5.png")

# Timing Studies
[evans@grendel PDE]$ ./ft 0 32768 10 5 0 | tee fft_time.out
[evans@grendel PDE]$ ./ft 1 512    6 5 0 | tee dft_time.out
root [0] .x ft_time.C
root [1] canv->Print("FFTvDFT.png")
```

References

1. Cooley, James W., and John W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.* 19, 297–301 (1965). doi:10.2307/2003354
2. Danielson, G. C., and C. Lanczos, "Some improvements in practical Fourier analysis and their application to X-ray scattering from liquids," *J. Franklin Inst.* 233, 365–380 and 435–452 (1942).
3. You can find a nice presentation of various implementations of the Danielson-Lanczos lemma at a site from National Taiwan University's CMLab: here [3].
4. Fastest Fourier Transform in the West, Matteo Frigo and Steven G. Johnson

back to top

return to Lectures

Last Modification: 31-Jan-11