

ECE 3640

Lecture 9 – Numerical Computation of the FT: the DFT

Objective: To investigate the discrete Fourier transform, which is a numerical way of computing Fourier transforms.

The DFT

We have seen that the Fourier transform can be used in a variety of applications. While an important theoretical tool, there is a problem with it in practice: we must have an analytical expression for the functions we are transforming, and have to be able to compute its transform. In practice, the transforms are often computed from discrete samples of a signal, and the transform is computed for only a discrete set of frequencies. This computational approach is known as the DFT. There are fast algorithms for computing the DFT which are called the Fast Fourier Transform (FFT). The FFT is nothing more than a way of organizing the computations in the DFT to compute exactly the same thing, but with less processing for the computer. It is safe to say that the revolution in signal processing has its genesis in the discovery of the FFT algorithm.

We will begin by making the connection between continuous-time signals with the spectrum and the DFT signals in their discrete time and frequency domains. Let $f(t)$ be timelimited to τ seconds, having spectrum $F(\omega)$. Then since $f(t)$ is timelimited, it cannot be bandlimited (why?). Let $\bar{f}(t)$ denote the sampled signal with samples taken every T seconds, and let $\bar{F}(\omega)$ denote the spectrum of the sampled signal, the periodic extension of $F(\omega)$, repeating every $\mathcal{F}_s = 1/T$ Hz. Because f is not bandlimited, there must be some amount of aliasing.

Now let us imagine sampling in the frequency domain of $\bar{F}(\omega)$; this corresponds to creating a periodic repetition of the sampled signal in the time domain. We let T_0 be the period of repetition in the time domain, corresponding to samples every $1/T_0$ of the spectrum.

Let us determine how many samples are involved here. The number of samples in the time domain (of each period) is

$$N_0 = \frac{T_0}{T}$$

The number of samples in the frequency domain of each period is

$$N'_0 = \frac{F_s}{F_0} = \frac{1/T}{1/T_0} = N_0.$$

The aliasing can be reduced by increasing the sampling frequency, but can never be entirely eliminated for a timelimited signal. (If we had started out with a bandlimited signal, then it would not have been timelimited, and we would have had overlapping in the time domain.)

If, in addition, the signal $f(t)$ that we are dealing with is not, in fact timelimited, but we must for reasons of practicality deal with a time-limited version, we must perform some truncation. This truncation leads to **spectral leakage**. This also causes aliasing (creating higher spectral components than might have been in the original signal.) The spectral leakage can be reduced by increasing the window width (longer data set). This increases T_0 , which in turn reduces F_0 . (Note that F_0 determines the frequency resolution.)

Also observe that by this sampling property, we are getting only a partial view of the spectrum. A true spectral peak might not lie right on one of the sample values. This can be mitigated by sampling more finely (decrease F_0 , meaning increase T_0 , meaning increase N_0 .)

Let $f[k]$ be a discrete-time sequence, possibly obtained from a continuous-time signal by sampling, say $f[k] = f(kT)$. Note that we are using k as the discrete time index. Suppose that we have N_0 samples of the signal, where N_0 is some number that we choose. (As a point for future reference, the number N_0 is commonly chosen to be a power of 2, since this works most efficiently for most FFT routines.) Following the book's notation, let

$$f_k = T f[k] = T f(kT) = \frac{T_0}{N_0} f(kT).$$

That is, it is simply a scaled version of the time sample. Often this scaling is overlooked (or it is assumed that the sampling interval is normalized to one). Also, let

$$F_r = F(r\omega_0).$$

That is, it is a sample of the spectrum.

The DFT of the set of points $\{f_0, f_1, \dots, f_{N_0-1}\}$ is given by

$$F_r = \sum_{k=0}^{N_0-1} f_k e^{-jr\Omega_0 k} = \sum_{k=0}^{N_0-1} f_k e^{-j2\pi r k / N_0} \quad (1)$$

where

$$\Omega_0 = 2\pi/N_0 = \omega_0 T.$$

In the transform formula, the number r must be an integer. It is an index to a frequency (as we will see later). The inverse DFT is

$$f_k = \frac{1}{N_0} \sum_{r=0}^{N_0-1} F_r e^{jr\Omega_0 k} = \frac{1}{N_0} \sum_{r=0}^{N_0-1} F_r e^{j2\pi r k / N_0}. \quad (2)$$

The transform pair is sometimes represented as

$$f_k \Leftrightarrow F_r$$

or

$$F_r = \mathcal{F}[f_k]$$

Note the interesting parallels between these transforms and the F.T.s we have already seen:

1. In going from the time domain (k) to the frequency domain (r), the exponential term in the summation has a negative sign.
2. In going from the frequency domain to the time domain, the exponential term has a positive sign, and the summation is multiplied by a normalizing factor, $1/N_0$ in this case.

It should be pointed out that while the sampling processing introduces possible aliasing and spectral leakage problems, the actual transform pair given above is *exact*.

Let us connect the definition given above with the conventional Fourier transform (noting where approximations are made). The sampled signal can be written as

$$\overline{f}(t) = \sum_{k=0}^{N_0-1} f(kT)\delta(t - kT)$$

(time limited). The Fourier transform of this signal is

$$\overline{F}(\omega) = \sum_{k=0}^{N_0-1} f(kT)e^{-jk\omega T}$$

If we *neglect the aliasing*, over the interval $|\omega| < \omega_s/2$ by the sampling theorem we have

$$\overline{F}(\omega) = \frac{1}{T}F(\omega).$$

Hence

$$F(\omega) = T\overline{F}(\omega) = T \sum_{k=0}^{N_0-1} f(kT)e^{-jk\omega T} \quad |\omega| < |\omega_s|/2$$

Sampling this now we obtain

$$F_r = F(r\omega_0) = T \sum_{k=0}^{N_0-1} f(kT)e^{-jkr\omega_0 T}$$

Let $\omega_0 T = \Omega_0$; then

$$\Omega_0 = \omega_0 T = 2\pi F_0 T = 2\pi/N_0$$

By our definition, $Tf(kT) = f_k$. Putting these definitions together we find

$$F_r = \sum_{k=0}^{N_0-1} f_k e^{-jr\Omega_0 k}$$

We conclude that, except for aliasing, the DFT represents a sampled version of the FT.

Taking the last expression for F_r , multiply by $e^{jm\Omega_0 r}$ and sum:

$$\sum_{r=0}^{N_0-1} F_r e^{jm\Omega_0 r} = \sum_{r=0}^{N_0-1} \sum_{k=0}^{N_0-1} f_k e^{-jkr\Omega_0} e^{jm\Omega_0 r}$$

We find (how?) that

$$\sum_{r=0}^{N_0-1} e^{j(m-k)\Omega_0 r} = \begin{cases} N_0 & k = m \pmod{N_0} \\ 0 & \text{otherwise} \end{cases}$$

Hence the sum collapses, and we get the desired result.

The value F_r is said to provide information about the r th frequency “bin”. It corresponds to a Hertz frequency of

$$f = r\Omega_0/2\pi = rf_0 = r/T_0 = r \frac{F_s}{N_0}$$

Take some examples:

$$\begin{aligned} r = 0 &\rightarrow f = 0 \\ r = 1 &\rightarrow f = F_s/N_0 = \frac{1}{T_0}. \end{aligned}$$

(This is the **frequency resolution**.)

$$r = N_0/2 \rightarrow f = F_s/2.$$

(This is the maximum representable frequency.) The number of points in the DFT can be written as

$$N_0 = \frac{T_0}{T},$$

where T_0 relates to the sampling rate in the frequency domain, and T relates to the sampling rate in the time domain. Observe that to increase the frequency resolution, the number of points **of data** N_0 must be increased. We can increase N_0 and improve the frequency resolution by increasing T_0 (taking more data), which amounts to taking samples closer together in frequency, $f_0 = 1/T_0$, or by taking samples faster in time (increasing the sampling rate).

Note: The book defined the transform in terms of $f_k = Tf(kT)$. In practice, the FFT routines simply deal with the data f_k ; they don’t worry about whether it is scaled or not. Since the effect of the scaling by T is to simply scale the transform, in applications it is not common to worry about the scaling either.

Example 1 A signal $f(t)$ has a duration of 2 ms and an essential bandwidth of 10 kHz. (What is its absolute bandwidth?) We want to have a spectral resolution of 100 Hz in the DFT. Determine N_0 , the number of points in the DFT.

By the sampling theorem, we must sample at least double the highest frequency, which is $B = 10000$. Let $F_s = 2B = 20000$ samples/sec.

The resolution is $F_0 = 100$ Hz. The signal duration is

$$T_0 = 1/F_0 = 10 \text{ ms}.$$

Here is an interesting thing: to get the spectral resolution we need, we need 10 ms of data, when in fact the signal only lasts for 2 ms. The solution is to take the 2 ms of data, then pad the rest with zeros. This is called **zero padding**. Then

$$N_0 = \frac{f_s}{f_0} = 200$$

In practice, we would probably want N_0 to be a power of 2, so $N_0 = 256$ is a better choice. This could be obtained by decreasing T , which would reduce aliasing, or increasing T_0 , which would increase the resolution. \square

Note that zero padding cannot replace any information that is not already present in the signal. It simply refines the sampling of the spectrum that is already present.

Aliasing and Leakage

There are two effects that are introduced into the computation of the DFT.

Aliasing Since when we compute we are necessarily dealing with a time-limited set of data, the signal cannot be bandlimited. The sampling process, with its incumbent spectral duplication, therefore introduces aliasing. This aliasing effect can be reduced by sampling faster.

Leakage If the function $x(t)$ is not really time limited, then we truncate it in order to obtain a finite set of samples. As viewed above, the mathematics sees the sampled signal as if it were periodic in time. There are two ways of viewing what is going on. First, if we have a function $x(t)$, we can obtain a time-truncated version of it by

$$y(t) = x(t)w(t)$$

where $y(t)$ is the truncated version and $w(t)$ is a windowing function. In the frequency domain, the effect is to smear the spectrum out,

$$Y(\omega) = \frac{1}{2\pi} X(\omega) * W(\omega)$$

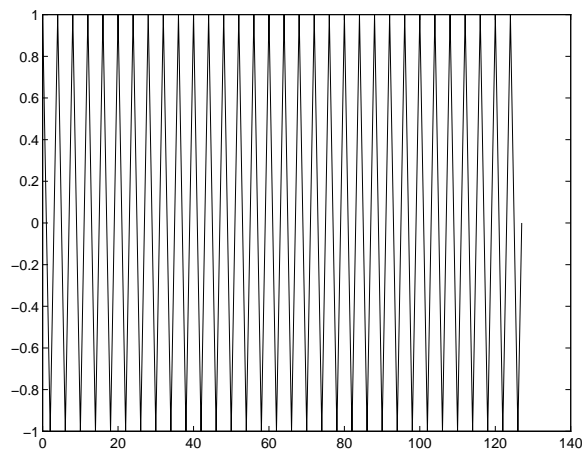
This smearing is spectral leakage. Another way of viewing the leakage is this: if we truncate a function then make it periodic, the resulting function is going to have additional frequency components in it that were not in the original function, due to the change from end to end. The only way this does not happen is if the the signal is periodic with respect to the number of samples already.

Leakage can be reduced either by taking more samples (wider windows of data), i.e. increasing N_0 . It can also be reduced by choosing a different window function. However, it can never be completely eliminated for most functions.

Some examples

In this section some examples of FFT usage are presented by means of MATLAB.

```
f = 100;      % frequency of the signal
fs = 400;     % sampling rate
N0 = 128;     % choose the number of points in the FFT
f0 = fs/N0;  % compute the frequency resolution, f0 = 3.125 Hz
k = (0:N0-1); % range of index values
fin = cos(2*pi*f/fs*k); % input function
plot(k,fin); % plot the function
```



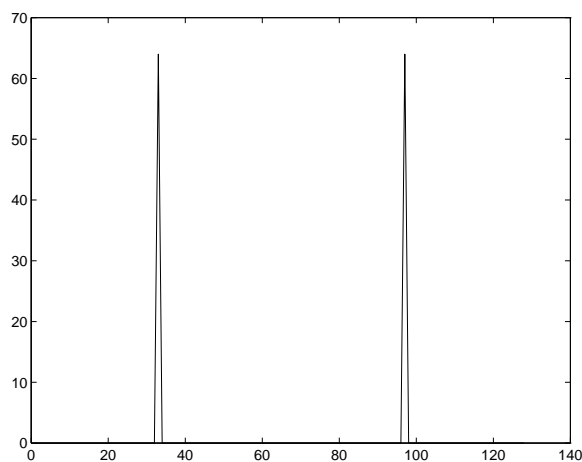
% Note that the function ends at the end of a cycle: its periodic
% extension is the same as the function.

%

fout = fft(fin,N0); % compute the FFT

plot(k,abs(fout));

% we have to plot abs(fout), since fout is a complex vector.



% note that there are exactly two peaks. This is because the
% frequency was a multiple of the resolution. The bin with the
% frequency peaks is bin 32 and bin 96 (128-32=96)
% since $f/f_0 = 32$.

% Note: the indexing in Matlab starts from index=1. This means that
% index 1 corresponds to bin 0. Hence, bin 32 is in fout(33):

% fout(33) = 64 + 0i

% fout(97) = 64 + 0i

%

% Now let's change the frequency and do it again.

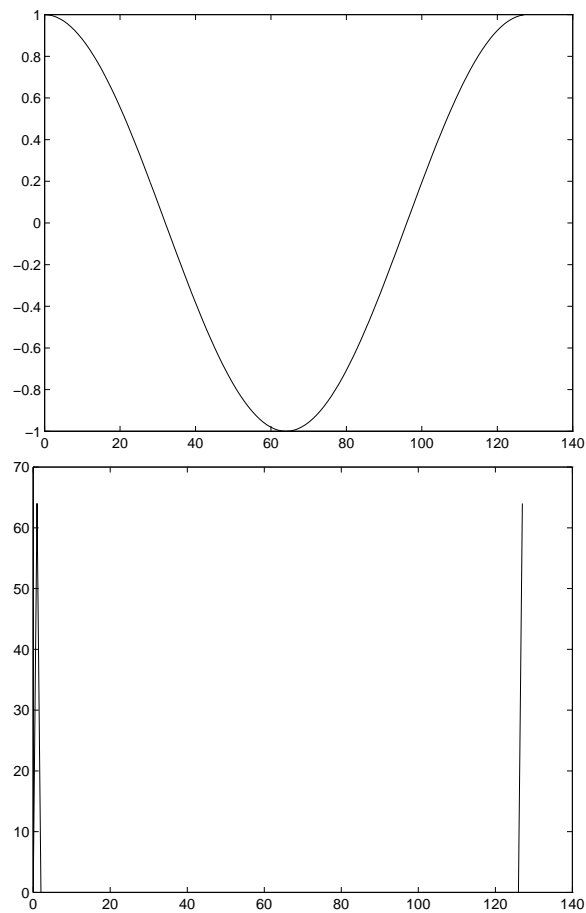
f = f0; % set the frequency to the lowest resolvable frequency

fin = cos(2*pi*f/fs*k); % function

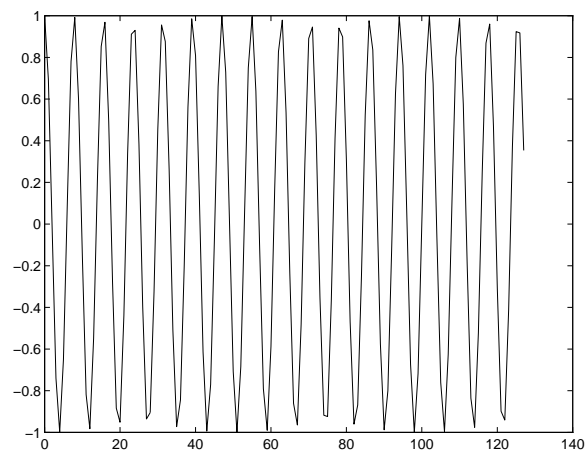
plot(k,fin); % plot the function

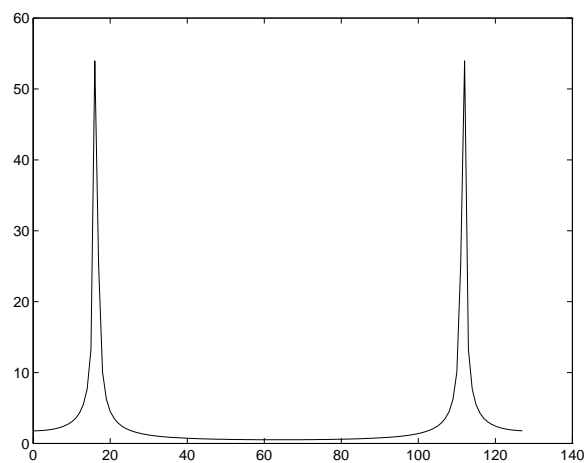
fout = fft(fin,N0); % compute the FFT

plot(k,abs(fout));

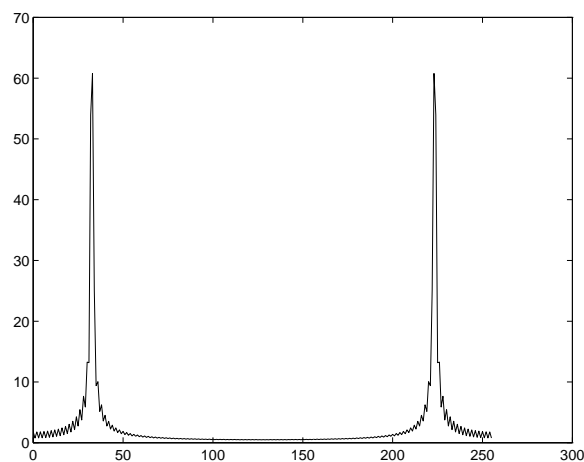


```
%
% Now let's try a frequency that is not a multiple of the
% frequency resolution
f = 51;
% Note that f/f0 = 16.32, which is not an integer bin number
%
fin = cos(2*pi*f/fs*k); % function
plot(k,fin); % plot the function
fout = fft(fin,N0); % compute the FFT
plot(k,abs(fout));
```



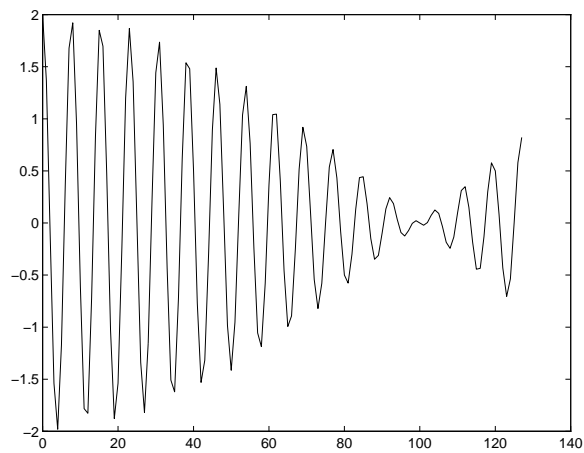


```
% Note that the time function does not end at the end of a period
% The transform does not occupy just one bin -- there is spectral
% leakage. The peak occurs at bin 16 -- the nearest integer to the
% true (real) frequency bin.
%
% Now let's examine the effect of zero padding on this problem
% (increasing N0). Using the same frequency, add on 128 zeros to the
% set of samples:
fin1 = [fin zeros(1,128)];
fout1 = fft(fin1,256);
plot(0:255,abs(fout1));
```

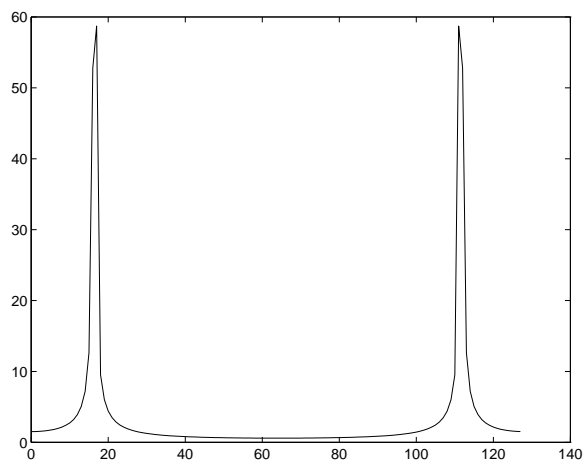


```
% Note that the zero padding did nothing to improve the spectral
% leakage.
```

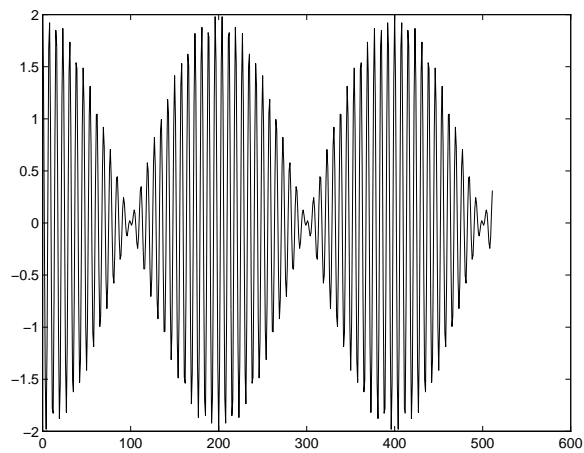
```
%
% Now let's examine the effect of the number of samples on the spectral
% resolution. Suppose that we have a signal that consists of two
% sinusoids, f1 = 51 Hz and f2 = 53 Hz. We will keep the same
% sampling rate. To begin with, we will take 128 points of data:
fin = cos(2*pi*k*f1/fs) + cos(2*pi*k*f2/fs);
plot(k,fin)
```

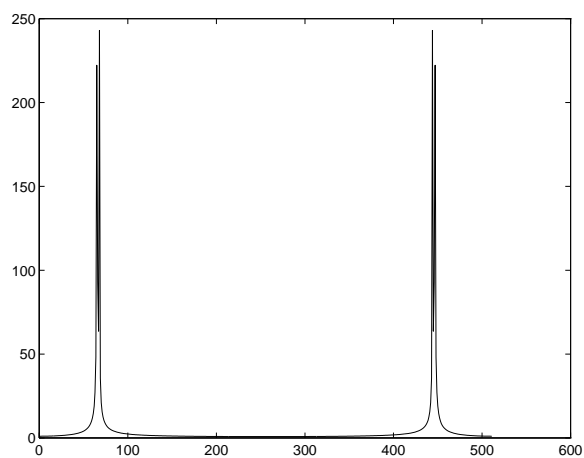
```
fout = fft(fin,128);
plot(k,abs(fout));
```



```
% The frequency resolution is f0=3.125 Hz. The frequencies we wish to
% distinguish are too close to both show up.
%
% Now let's increase the number of sample points up to 512. This gives
% a frequency resolution of f0 = 0.7813
k2 = (0:511)
fin2 = cos(2*pi*k2*f1/fs) + cos(2*pi*k2*f2/fs);
plot(k2,fin2);
```



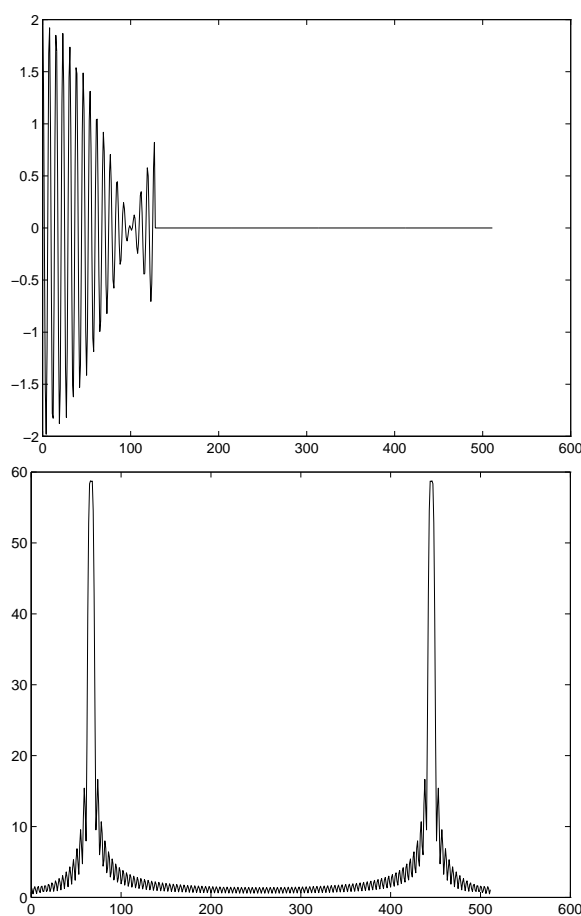
```
fout2 = fft(fin2,512);
plot(k2,abs(fout2));
```



```
% In this case, we are able to distinguish between the two
% frequencies.
```

```
%
% Now, suppose that instead of taking more data, we had simply decided
% to zeropad out to 512 points.
```

```
%
fin1 = [fin zeros(1,384)];    % 512 points
plot(k1,fin1)
fout1 = fft(fin1,512);
plot(k1,fout1);
```



% Note that we have not increased the spectral resolution. We have
 % only provided a more finely-sampled version of the spectrum we saw
 % with 128 points.

The FFT

The FFT is a way of organizing the computations necessary to compute the DFT in such a way that the same work is done, but with less effort. I like to think of this as a bicycle analogy. If I ride my bike from home to work, I use the same engine (me!) as if I walk in. But, by organizing my efforts better (by means of wheels and gears), the engine is able to work more efficiently, and I get here faster.

To gauge the improvement, let's first approximate how many computations are necessary to compute the DFT. If I have N_0 data points, I can compute one of the DFT outputs by the formula

$$F_r = \sum_{k=0}^{N_0-1} f_k e^{-j2\pi kr/N_0}.$$

There are N_0 terms in the summation, each of which requires a multiply (naturally, in the name of efficiency we would precompute and store all the complex exponential factors). So each point requires N_0 multiply-accumulate operations. There are N_0 points in the transform, so the overall DFT takes about N_0^2 multiply accumulates. If $N_0 = 1024$, this turns out to be a whopping 1,048,576 multiply-accumulates. We

would say the complexity is $O(N_0^2)$, where the O means “order of”, which is to say, “in the ballpark of.”

The FFT is able to reduce the computational requirements to about $O(N_0 \log_2 N_0)$. For the case of $N_0 = 1024$ this turns is 10240 operations, which is more than 100 times faster! And it provides exactly the same result. This speedup has enabled some algorithms to be developed and successfully deployed that never could have happened. Note that this speedup is entirely *independent* of the technology. It was 100 times faster 20 years ago, and will be 100 times faster 20 years from now. The discovery of the FFT laid the foundation for modern DSP, as well as a variety of computational disciplines. One of the morals of this story is that it pays to pay attention to how the computations are done!

We will not go into too many details of how the FFT works — we need to leave something to ECE 5630. We can, however, give a brief summary. The basic technique is divide and conquer: we take a problem with N_0 points, and divide it into two problems of $N_0/2$ points. Since the complexity goes up as the square of the number of points, each of these $N_0/2$ problems is four times as easy as the N_0 problem. If we can put these two problems together to give the overall answer, then we have saved computations. We then take each of these $N_0/2$ problems and split them into $N_0/4$ problems, each of which is easier, and so on. At every level, we split the number of points that go into the computation, until we get down to a DFT in just two points. This is easy to compute. This subdivision shows why the number of points must be a power of two: we split by two at every stage in the process. This also shows why the complexity has the factor of $\log_2 N_0$ in it: we can split N_0 by two that many times.

All good routines to numerically compute the DFT are based upon some form of FFT.

Convolution using the DFT

We have seen the convolution theorem over and over: convolution in time is the transform of multiplication in the frequency domain. By numerically computing the transform using the DFT, we can compute convolutions of sequences. There are some issues to be very careful of, however, since the DFT imposes certain requirements on the signals. We will worry later about this. For the moment, consider the computational complexity. Suppose I have a sequence x of N points and another sequence y also of N points. We know that the convolution will have $2N - 1$ points. Computation of each of the outputs requires approximately N^2 computations. The overall complexity for computing convolution is therefore $O(N^2)$.

But here is the neat thing: We compute the DFT of x and the DFT of y , multiply the points in the frequency domain, then transform back:

$$X_r = \text{DFT}(x_k)$$

$$Y_r = \text{DFT}(y_k)$$

$$Z_r = X_r Y_r$$

$$z_k = \text{IDFT}(Z_r)$$

This sure seems like the long way around the barn! But, consider the number of computations if we do the DFTs using FFTs: Each transform requires $O(N \log_2 N)$ operations, the multiplication is $O(N)$, and the inverse transform is $O(N \log_2 N)$. The overall computation is $O(N \log_2 N)$. (That’s how orders are computed!). So it requires fewer computations (by far!) than straightforward convolution, at least if N is large enough. This is, in fact, the way MATLAB computes its convolutions.

This is also the way that symbolic packages (such as Mathematica) compute their multiplications of large integers.

Now the issues regarding use of the DFT for convolution. Recall that the DFT always assumes that the signal is *periodic*. This is the key to understanding the convolution. The convolution is done on *periodic* signals, where the period is the number of points in the DFT, and the result of the convolution is periodic also. Suppose that we are dealing with N -point DFTs. We can define a convolution which is periodic with period N by

$$y_k = f_k * g_k = \sum_{n=0}^{N-1} f_n g_{((k-n))}$$

The notation $((k-n))$ is used to indicate that the difference is taken modulo N . Suppose $N = 10$, and f_k is a sequence 6 points long and g_k is a sequence 10 points long. Draw sequences, and their 10-periodic extensions. Show graphically what the periodic convolution is. **The periodic convolution is the convolution computed when DFTs are used.**

Suppose we don't want the effects of the circular convolution; we just want regular linear convolution. What we need to do is zero pad. If f_k is a sequence of length N and g_k is a sequence of length M , then their convolution will have $N + M - 1$ points. If we take a DFT with at least that many points, there will be no wrap around. Show pictures.