Fast Fourier Transform – FFT

This post is about a good subject in many areas of engineering and informatics: the Fourier Transform. The continuous Fourier Transform is defined as:

$$F(\omega) = \int_{-\infty}^{\infty} f(t) e^{-j\omega t} dt$$

f(t) is a continuous function and *F(w)* is the Fourier Transform of *f(t)*.

But, the computers don't work with continuous functions, so we should use the discrete form of the Fourier Transform:

$$F[p] = \sum_{n=0}^{N-1} f[n] e^{-\frac{2\pi p n j}{N}}$$

f[n] is a discrete function of *N* elements, *F[p]* is a discrete and periodic function of period *N*, so we calculate just *N* (*0* to *N - 1*) elements for *F[p]*.

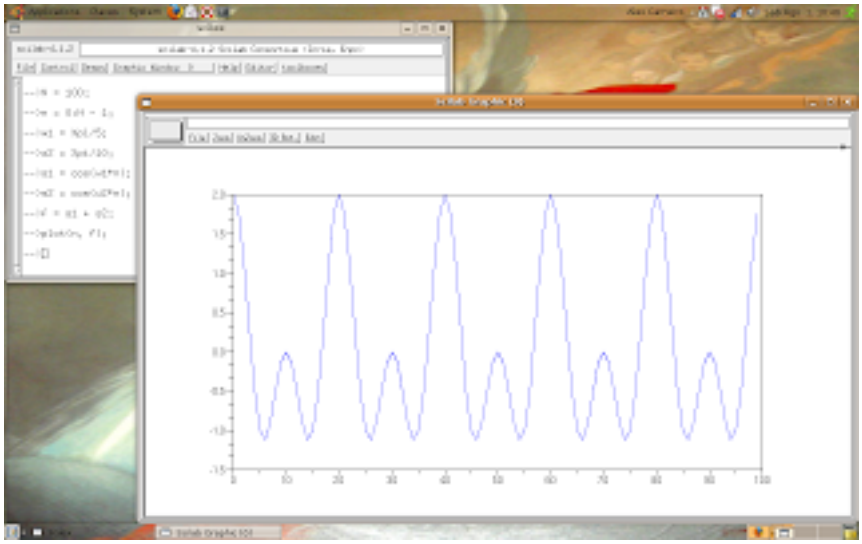
Who studies digital signal processing or instrumentation and control knows the utilities of this equation.

Now, how to use the Fourier Transform in Scilab?

If we are using large signals, like audio files, the discrete Fourier Transform is not a good idea, then we can use the fast Fourier Transform (used with discrete signals), look the script:

```
-->N = 100; // number of elements of the signal

-->n = 0:N - 1;

-->w1 = %pi/5; // 1st frequency

-->w2 = %pi/10; // 2nd frequency

-->s1 = cos(w1*n); // 1st component of the signal

-->s2 = cos(w2*n); // 2nd component of the signal

-->f = s1 + s2; // signal

-->plot(n, f);
```

The result is:



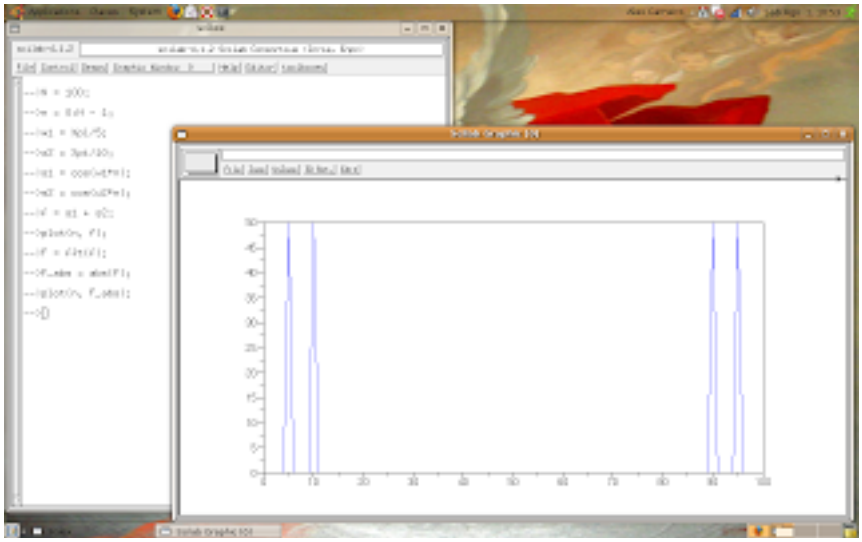
Now, let's study the Fourier Transform of our signal.

-->F = fft(f); // it calculates the Fourier Transform

-->F_abs = abs(F); // F_abs is the absolute value of each element of F

-->plot(n, F_abs);

The result is:



Look at the two graph's peaks, one for the component *s1* and the other for the component *s2*.

The Fourier Transform is a linear transformation, thus it has a inverse transformation: the Inverse Fourier Transform.

Scilab has the function ifft(.) for obtain the original signal from it Fourier Transform.

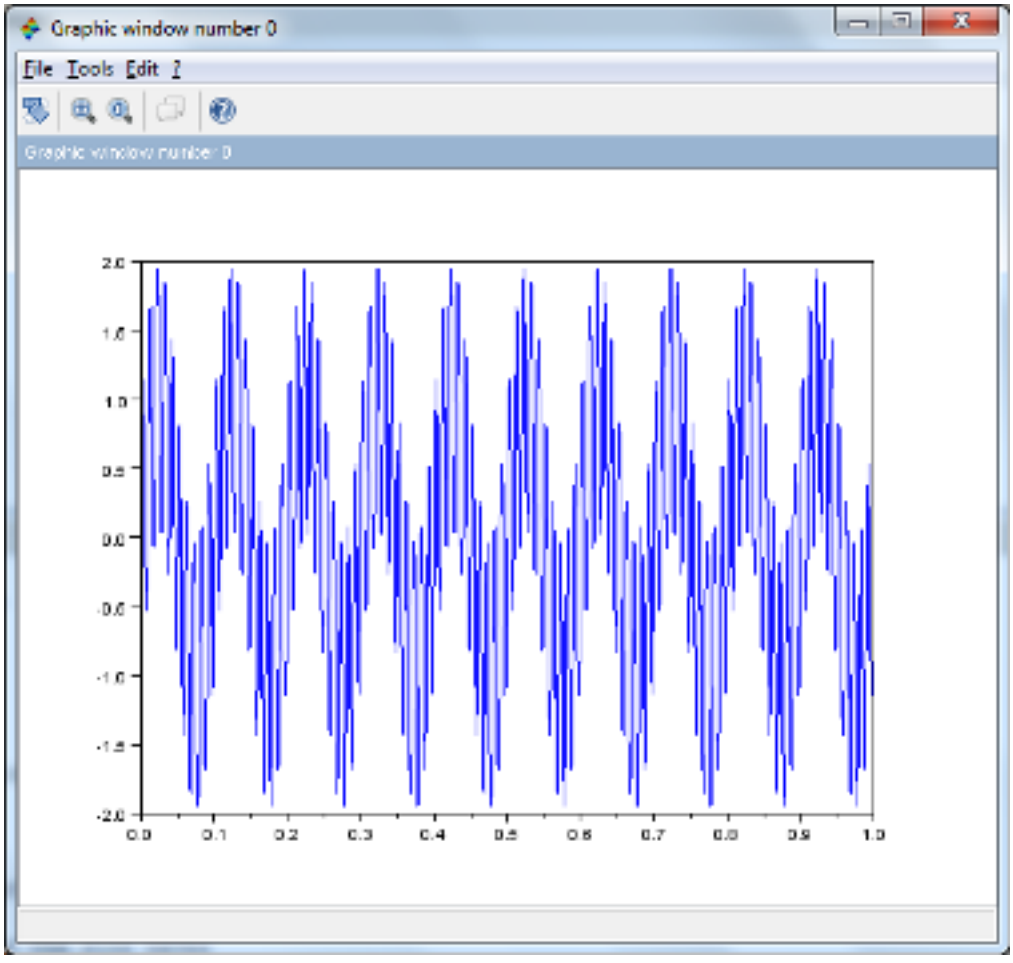
If anyone wants to know, I can make a new post about how to identify the frequencies of the original signal in the Fourier Transform.

*Posted By **Chin Luh Tan**, Wednesday, August 10, 2011*

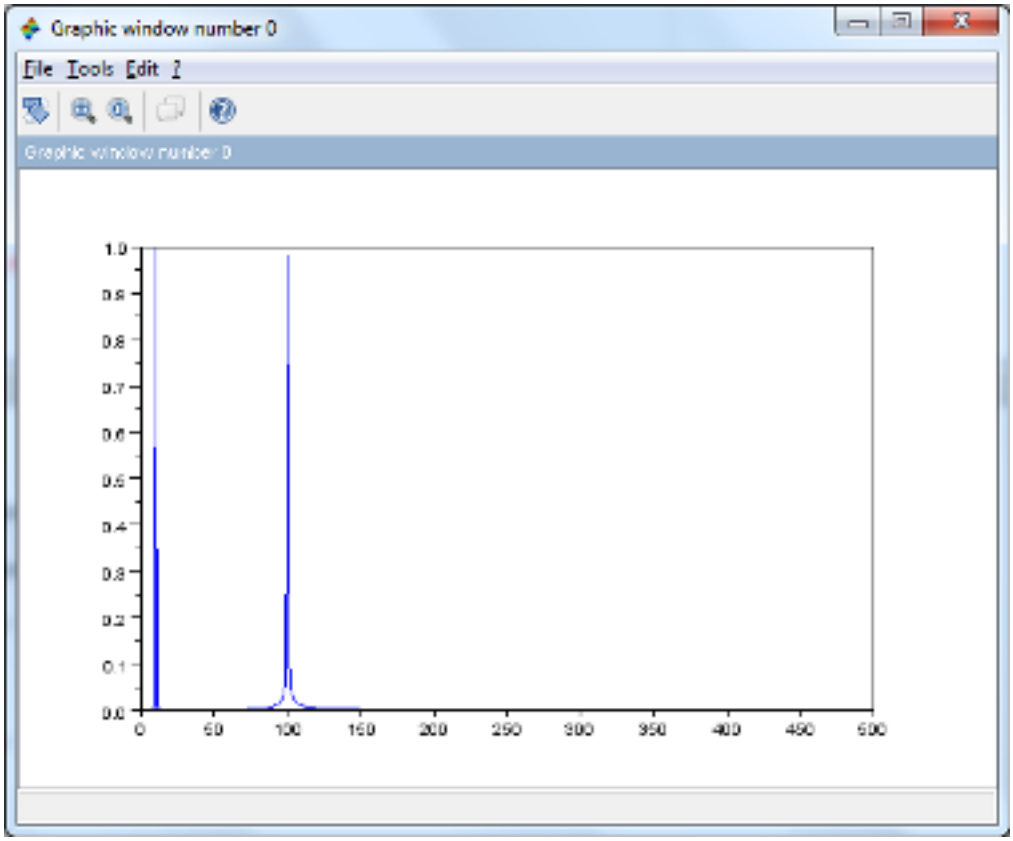
I always believe that visualization makes one understand a subject better than just looking at the equations.

Following is an example of simple filter design which shows a better picture on the signals in both time and frequency domain.

```
// Creating signals with sampling frequency of 1000 Hz

Fs = 1000;

t = 0:1/Fs:1;

n = length(t);

f = linspace(0,Fs,length(t)); // Create frequency vectors

x1 = sin(2*%pi*10*t); // 10 Hz Sine Wave

x2 = sin(2*%pi*100*t); // 100 Hz Sine Wave

x = x1 + x2; // Combination of 10 Hz and 100 Hz Sine Wave

plot(t,x); // Time Domain representation of the sine waves
```

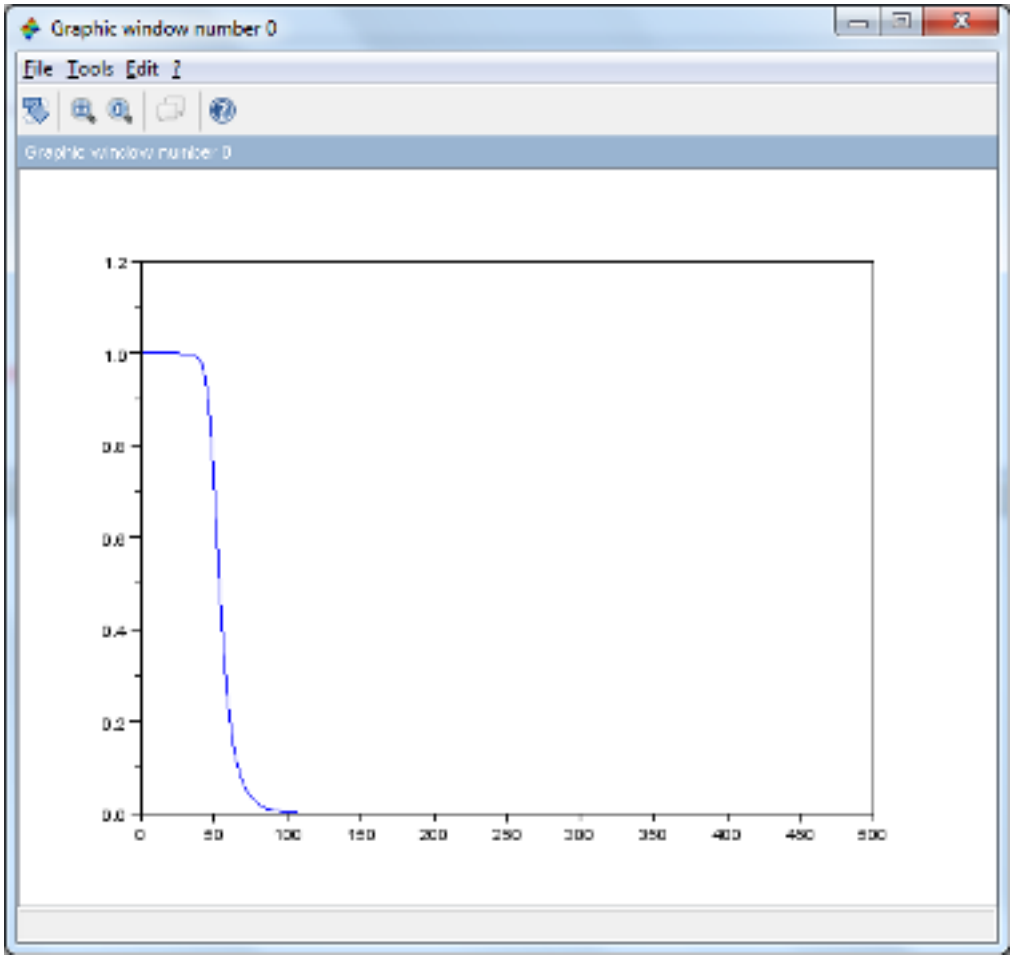



```
X = fft(x)./(length(x)/2); // Creating frequency response of the signal

plot(f(1:n/2),abs(X(1:n/2))); // Frequency Domain representation.
```



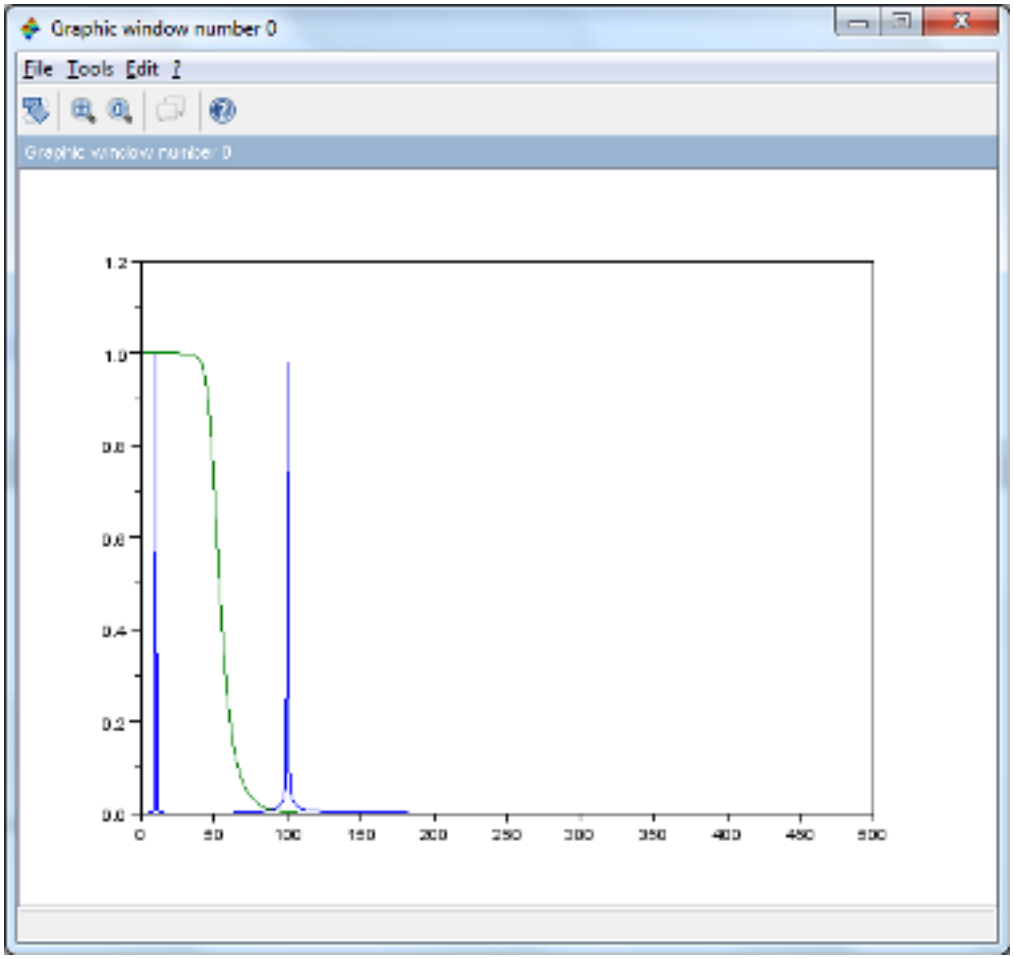
The frequency domain plot shows the contents of the signal, which consists of 10 Hz and 100 Hz components. Now we are going to design a filter to eliminates the 100 Hz component.

```
// Design of a low-pass butterworth filter with 50 Hz cut off frequency

hz = iir(8,'lp','butt',50/Fs,[]);

[hzm,fr]=frmag(hz,256);

fr2 = fr.*Fs;

plot(fr2,hzm)
```



To make it clearer, I overlap 2 graphs together:

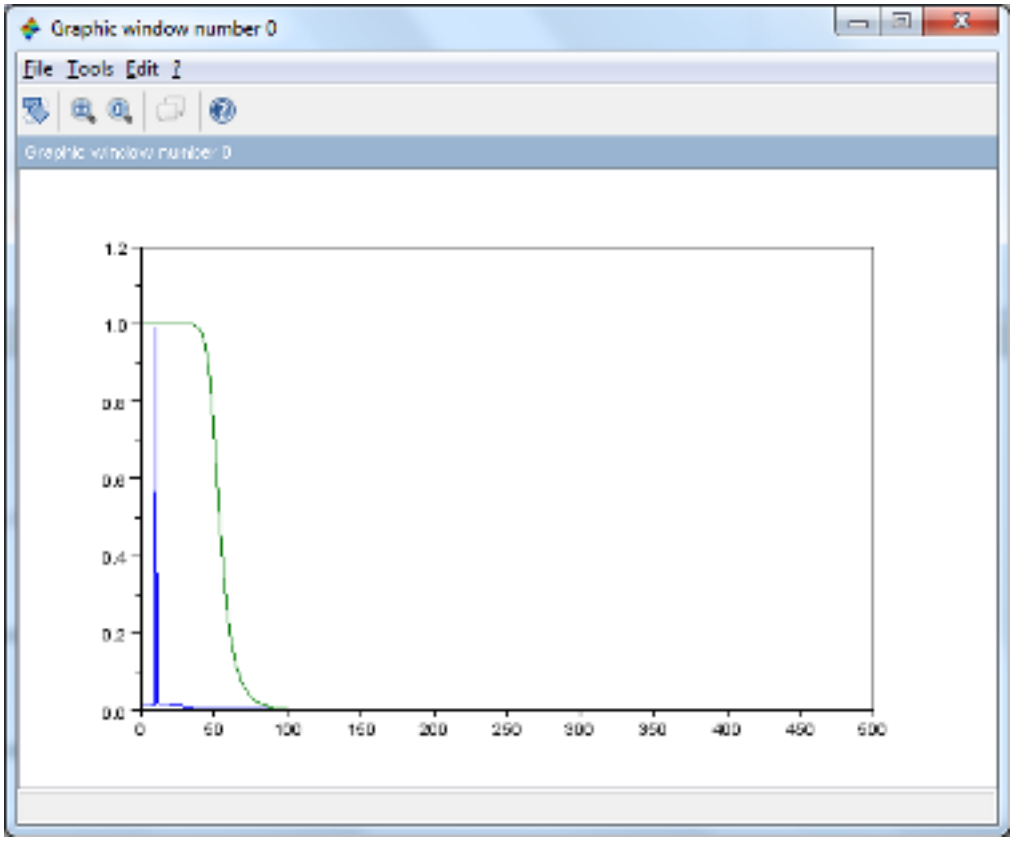
plot(f(1:n/2),abs(X(1:n/2)),fr2,hzm);



The green color line indicates the "passband", or the allowed zone for the signal, and looks like it will pass through the 10 Hz component and eliminates the 100 Hz component!

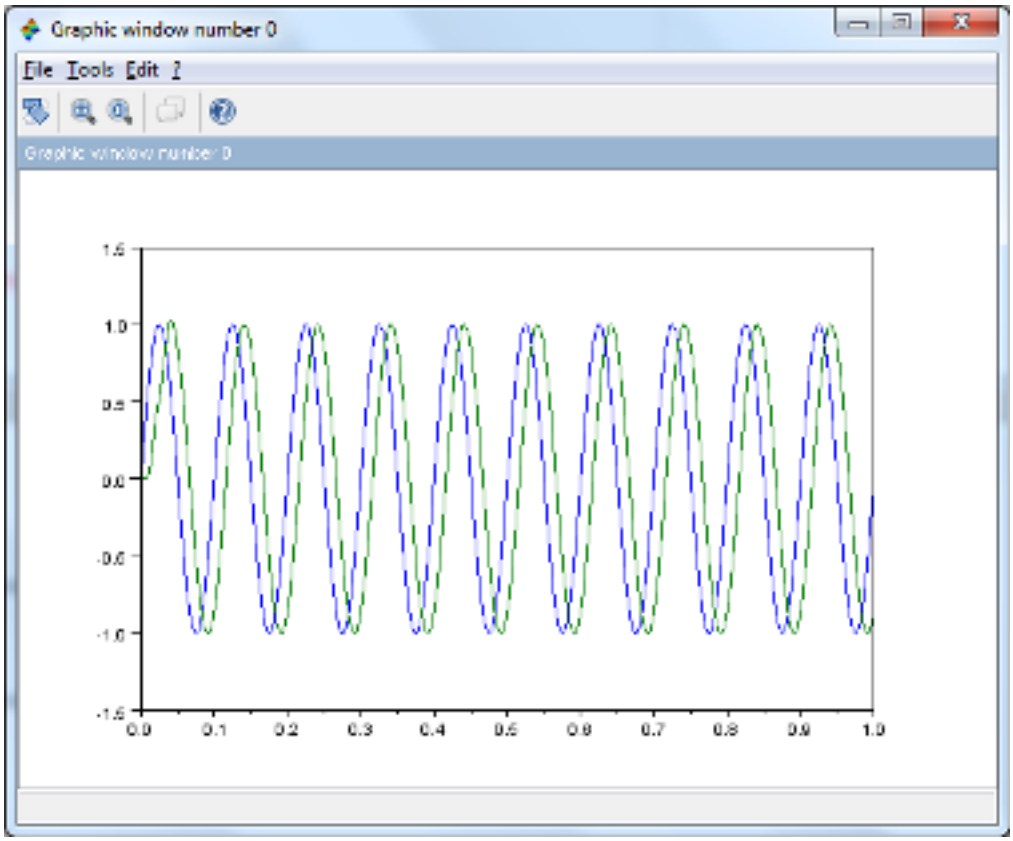
Let's prove it!

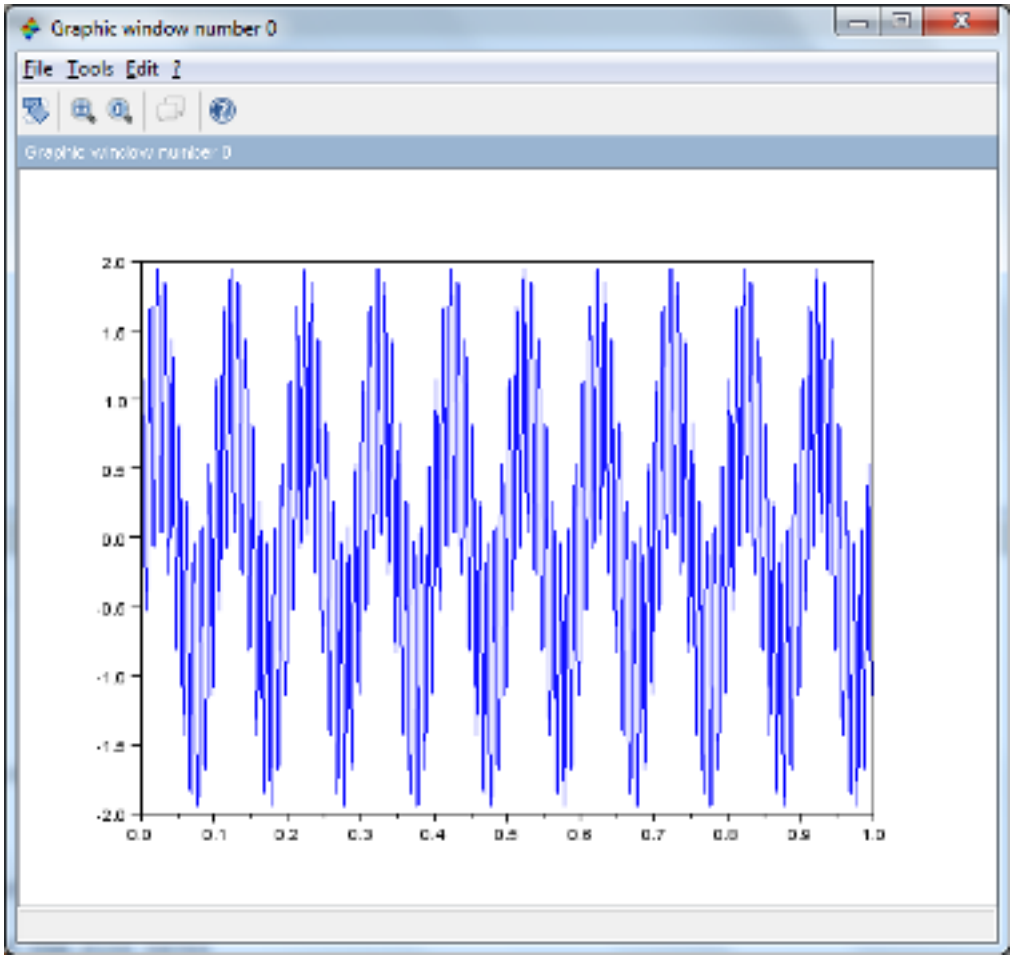
```
// Applying filter to the signal

y = flts(x,hz);

Y = fft(y)./(length(x)/2);

// Compare the frequency domain of filtered signal with filter response

plot(f(1:n/2),abs(Y(1:n/2)),fr2,hzm);
```



Finally, compare the original 10Hz signal with the filtered signal.

plot(t,x1,t,y);




Graphic window number 0
File Tools Edit ?
Graphic window number 0
2.0
1.5
1.0
0.5
0.0
-0.5
-1.0
-1.5
-2.0
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0

Fourier Theory is the science of additive synthesis.

1. The Forward Transformation: Take a waveform and find one of its frequencies.

2. The Reverse Transformation: Add all the frequencies to get the waveform back.

The FFT program does both the forward and reverse transformation for you. With the Scilab program you take your sound wave data, say the array called y, and plug it into the FFT function:

```
fy  =  fft(y, -1);
```

which gives you all the frequencies that make up your sound wave. The reverse transformation is done in Scilab with the following command:

```
y  =  fft(fy, 1);
```

which gives you the original waveform back, given the frequencies it contains.

Example 1:

Let's say you have a pure cosine waveform and you want to find its frequencies. Well, there should only be 1 frequency because it is a pure sinusoid. Let's create an arbitrary cosine waveform with Scilab. Make it simple so that it's easy to see on a plot. Let's say the frequency is 1 hz, and let's pretend the sampling rate is only 128 samples per second (WAV files are sampled at 44100 hz). We will make a waveform that has a duration of 1 second for simplicity. Then enter the following Scilab commands:

```
    Fs  =  128;             // the sample rate     f   =  1;
// the frequency     N   =  1 : 128;         // an array of numbers
counting from 1 to 128     x   =  2 * %pi * (f / Fs) * N;      // the
input to the cosine     y   =  cos(x);
```

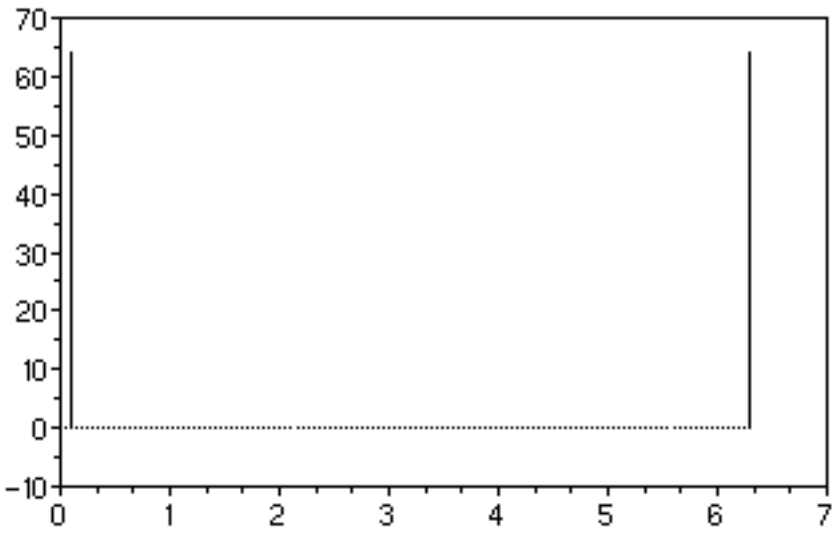
Recall from the earlier tutorial, creating a sound file, why we must write the cosine argument *x* with the sampling rate as above. Now the array called *y* has one complete cycle of a cosine wave. The frequencies can be calculated by putting the cosine wave, *y* into the FFT function. Enter the following Scilab command:

```
fy   =  fft(y, -1);
```

Now the array called *fy* contains the frequencies. These are complex numbers, so you have to plot the real part and the imaginary part separately.

```
plot2d3(x,  real(fy))
```

which will look like the following plot



In the above plot of the FFT of *cos(x)* there are two vertical bars, one at the beginning and one at the end. These are the frequencies. There is really only one frequency: the one at the beginning. The frequency bar at the end is its mirror image, and this will be explained below.

The above plot is just the real part of the complex number array. To plot the imaginary part of the array we would use the same command as above but replace real(fy) with imag(fy). Since this was a cos(x) function there is no imaginary part. If we had used sin(x) for our test waveform instead of cos(x) there would only be imaginary numbers, and the real part of the array would be all zeros. Be forewarned that the "zero" elements of this array will usually not be exactly zero. The FFT calculation always results in some *numerical noise*. This is especially important when comparing the real components to the imaginary components. If we examine the numbers in the *fy* array we will be surprised to find that numbers that should be zero are very small, but not zero. For example, look at the frequency element for the above plot, which is fy(2). Just type fy(2) at the Scilab command line. We get the following result:

```
ans  =          63.922909 + 3.1403312i
```

The imaginary part of this spectrum is theoretically all zeros, but this FFT calculation came out with 3.1403312 for the imaginary part of this frequency. That is numerical noise. You will need to watch for this as you apply your frequency analysis to music synthesis.