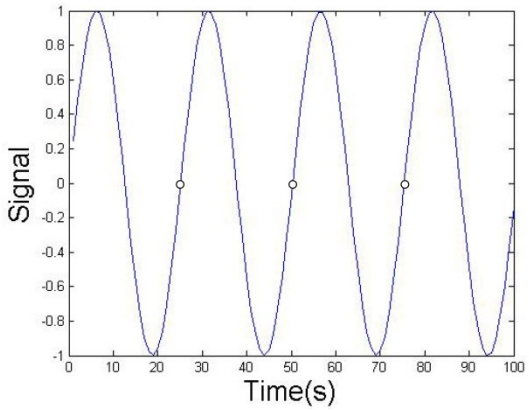
Chapter 5. Fourier Transforms and Signal Analysis

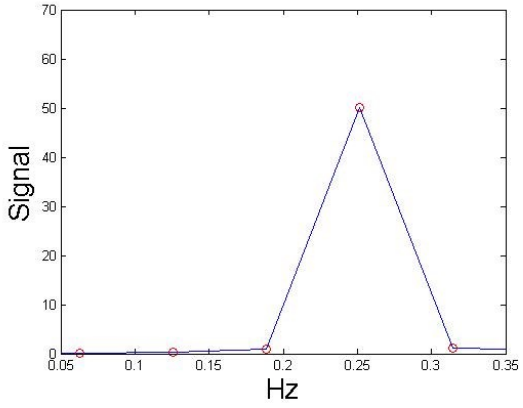
5.1 Frequencies in signals

The Fourier transform analysis is one of the most useful ever developed in Physical and Analytical chemistry. Everyone knows that FTIR is based on it, but did one ever actually calculate the signal from the raw data? What else can be done with Fourier analysis? Here you are going to actually understand at depth the true nature of the Fourier transformation technique.

Overall, the Fourier transform calculation takes a signal and returns the spectra of the component frequencies. Visually this is easier to understand; below is a spectrum of an oscillating signal that repeats itself every ~25 seconds.



You can see that the signal starts out close to 0, and then makes a full cycle every ~25 seconds as marked with the circles. There are no obvious beating patters indicative of the presence of other signals. If we make a Fourier transform of this we see the following:

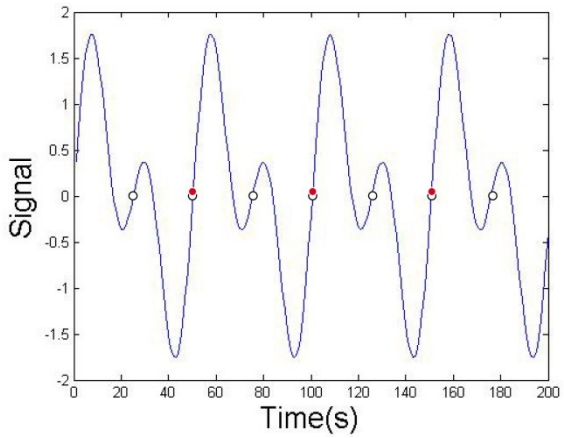


Now understanding the nature of the x-axis of a Fourier transform can be tricky. First, note that the FT spectrum has an x-axis that is the inverse of the original signal, which is seconds in this example. Thus, the FT spectrum has an x-axis of Hertz, or s^{-1}. Next, a feature at 0.25 Hz corresponds to 4 seconds. As a cycle covers 2π of ground, we multiply $4 \times 2\pi = 25.1$ s, which is how long the original signal took to repeated itself. In other words, the Fourier transform has a peak at the frequency of the original signal. Here is another bit of information; here is how I made the signal:

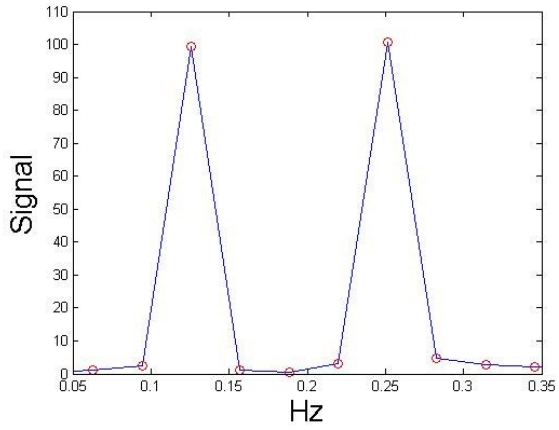
```
for i=1:100
signal(i)=sin(i*1/4);
end;
```

Now is the relationship between the frequency, sin(i***1/4**), and the peak at 0.25 Hz, more clear? Let's make a two-component signal, one with a 25.1 second frequency component and longer one twice that (50.2 s) by:

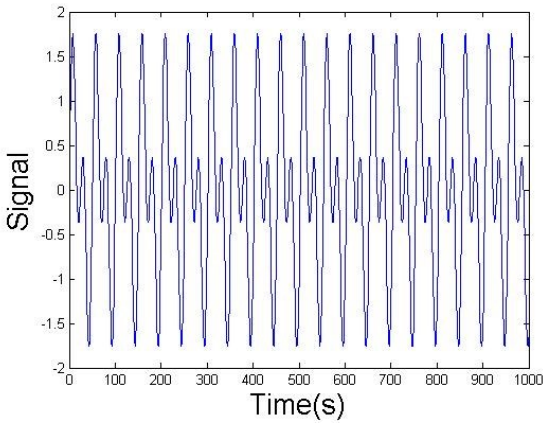
```
for i=1:100
signal(i)=sin(i/4)+sin(i/8);
end;
```

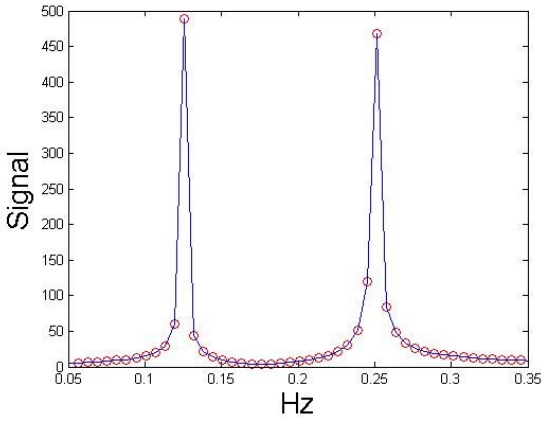


A beating pattern of two waves overlapping is clear; repeated patterns are observed every ~25 seconds (circles) and ~50 seconds (red dots). Fourier transformation of these data yield:

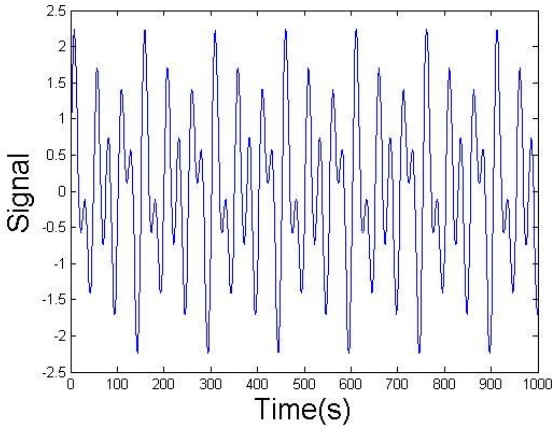


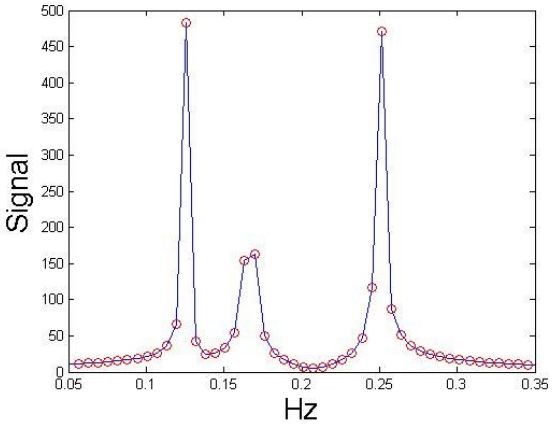
A new feature appears at a smaller frequency 0.125 s^{-1} (=1/8 s^{-1}), and note that (1/0.125 s^{-1}) × 2π = 50.2 s. This is in fact what I added to the signal. Note that I had increased the number of data points, measuring the signal out to 200 s. Let's see what happens when we go to 1000 seconds of data and transform it:





Obviously, longer signal gives a transform with more intense and sharper peaks. Let's now have three signals at once, the two above and I add a 37.5 second component that is ½ as strong as the other two as shown below (left). Right is the transform.





A new feature appears at ~0.167 s^{-1}; and note that (1/0.167 s^{-1}) × 2π = 37.5 s. Also, this feature has ½ the intensity of the other two. Thus, it seems that the Fourier transform can be used to measure the frequency spectrum of a signal as well as the relative contribution of those signals to the total.

5.2 Fourier transform mathematics

Now here are the actual mathematics; the Fourier transform is defined as:

$$Y(k) = \int_{-\infty}^{\infty} y(x) \cdot e^{-2 \cdot i \cdot \pi \cdot x \cdot k} \cdot \partial x$$

Whoa Nelly! Isn't it time to give up now! Just look at that thing!
Hold on, let's break it down into pieces. Let's get rid of the awful exponential term first by noting that:

$$e^{-2 \cdot i \cdot \pi \cdot x \cdot k} = \cos(2 \cdot \pi \cdot x \cdot k) - i \cdot \sin(2 \cdot \pi \cdot x \cdot k)$$

While the result is indeed complex (that's the sine part), overall, we can handle sines and cosines.

Next, remember that integrals are sums, especially if we have a discrete data set. This is a fancy way of saying that we are dealing with a data file with N finite numbers. Thus, we are actually dealing with an entity called the Discreet Fourier transform which looks like this:

$$Y(k) = \sum_{n=0}^{N-1} y\big(x(n)\big) \cdot e^{-2\cdot i\cdot \pi\cdot k\cdot x/N}$$

$$= \sum_{n=0}^{N-1} y(n) \cdot \left[\cos\left(2 \cdot \pi \cdot k \cdot x(n)/N\right) - i \cdot \sin\left(2 \cdot \pi \cdot k \cdot x(n)/N\right)\right]$$

Does this still look overwhelming? Let's make this easier be identifying what is what. First, the y(n) are just the data that tend to come in the 2nd column in the examples I give you. X(n) are the data in the 1st column, and there are N total data points. The little n is just the index that sums over data; in the example below, that's the "i" in the "for i=1:1000…" And you know that to sum things in Matlab you do the following:

```
thisisasum=0;
for i=1:1000
    thisisasum = thisisasum + whatever your adding together;
end;
```

Now the k is an integer that runs from k=0→N-1 (i.e. 0, 1, 2… N-1, and note that's actually N data points). So what is k really doing? K is providing the frequency information. If you look at the argument of the sine or cosine: $2 \cdot \pi \cdot k \cdot x/N$; as k goes from 0, 1, 2 etc. the frequencies are:

$$0, \frac{2\pi}{N}, \frac{4\pi}{N}, \frac{6\pi}{N}, \frac{8\pi}{N}, \ldots, \frac{2(N-1)\cdot\pi}{N}$$

This is basically your new x-axis in frequency space that you plot your Fourier transformed signal against. It also has the units of x^{-1}, since x obviously has some unit attached (usually seconds), and the argument of a sine or cosine cannot have units (i.e. $2 \cdot \pi \cdot k \cdot x/N$ must have no units, and N is just the number of data points and cannot have units. Thus, k must cancel out x). Thus, if x is in seconds, k is in s^{-1}.

If this is still daunting, let's start practicing. Download the dataset5.txt data, which is for the last example provided above. Now write your own code for calculating Fourier transforms, or, use this one here. Note that I left out a few lines of code; I want you to be able to figure out how to program so I'm not babying you fully anymore, and I have made intentional mistakes as well. The heart of the code works regardless. Please try to understand how each line works as well (note that if you are completely stuck I included the proper code as part of this packet).

```
mydata5=load('Dataset5_1.txt');

%INITIALIZE VARIABLES HERE! ERROR MESSAGES TELL YOU WHAT TO FIX

for i=1:length
    freq(i)=2*pi*(i-1)/length;
    for j=1:length
        ftreal(i)=ftreal(i)+dataset5(j)*cos(freq(i)*j);
```

```
        ftimag(i)=ftimag(i)-dataset5(j)*sin(freq(i)*j);
    end;
    fttot(i)=sqrt(ftreal(i)^2+ftimag(i)^2);
end;
```

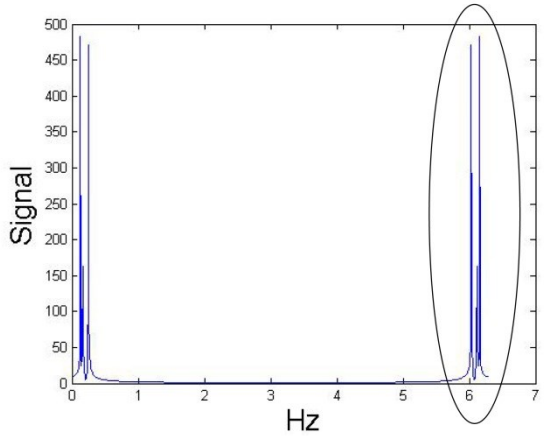
Note that we have real and imaginary components; the proper way to deal with this is to calculate them separately as above. Sometimes, depending on what kind of data you are taking Fourier transforms of, the real and imaginary components may mean something different. However, this is uncommon and as such one is generally interested in the "net" transform, or the absolute magnitude which is:

$$\text{sqrt}(\text{real}^2 + \text{imaginary}^2)$$

You can see where that is calculated on the next to last line of code. Now plot the data:

>>plot(freq,fftot)

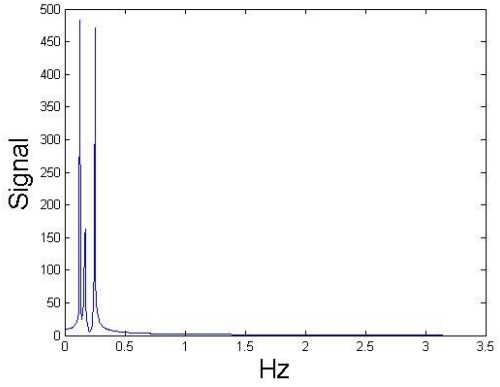
Here is what you see:

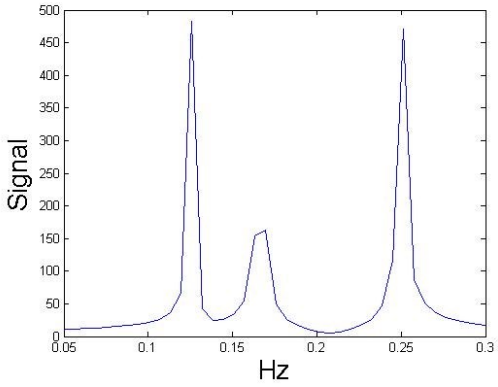


While we expect the peaks on the left side (0.1→0.3 Hz), as seen above, the right handed high frequency peaks (circled) may be confusing. These are actually mirror images of the real peaks from 0.1→0.3 Hz. They form because the real data on the left side are calculated with "right" travelling waves while the data on the left side are the same calculated with "left" travelling waves. This means that the frequency is determined by the distance the x-point is from very middle of the spectrum (point 501 here). There is really nothing wrong with this, but to help you visualize the data, the way to deal with this is to redefine the x-axis as follows:

```
for i=1:500
xax2(i)=(i-1)*2*pi/1000;
end;
for i=501:1000
xax2(i)=(1001-i)*2*pi/1000;
end;
```

Replot and zoom in as well:





Everything looks normal. Try to make up your own signal and analyze it:

```
for i=1:1000
analyzethis(i)=cos(i/5)+sin(i/10);
end;
```

and then Fourier transform it. This actually can be kind of fun.

5.3 Allowed frequencies and Aliasing errors

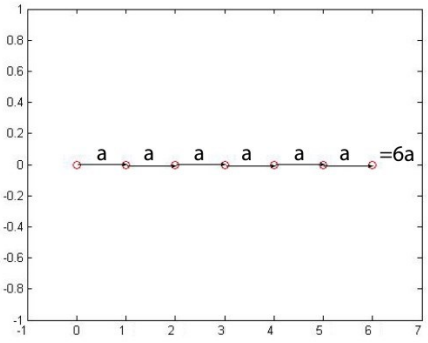
One of the biggest mistakes a person makes with discrete Fourier transforms is that they do not know what are the allowed frequencies are. As stated above, the allowed frequencies are:

$$0, \frac{2\pi}{N}, \frac{4\pi}{N}, \frac{6\pi}{N}, \frac{8\pi}{N}, \ldots, \frac{2(N-1)\cdot\pi}{N}$$

This is actually wrong; I simplified it earlier to make getting the general idea across easier. So what is missing is the unit of inverse seconds as N and π have no units. They are missing because I left something out- if the spacing between data points on the x-axis is "a", then the real frequencies are:

$$0, \frac{2\pi}{a(N-1)}, \frac{4\pi}{a(N-1)}, \frac{6\pi}{a(N-1)}, \frac{8\pi}{a(N-1)}, \ldots, \frac{2(N-1)\pi}{a(N-1)}$$

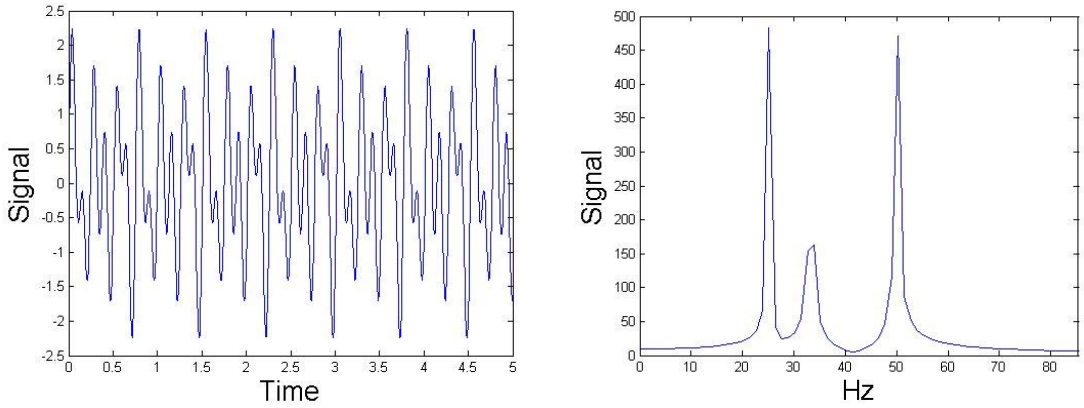
First, note that (N-1)·a is actually the total length of the data set. Imagine a data set of **7** points each set "a" amount apart as shown below:



See? The length is actually 6·a = (7-1)·a due to the fact that the first point doesn't advance the line, rather starts it. So how did I get away before with using N vs. N-1 in my Fourier transform program above? It's a stupid Matlab artifact that an array of numbers cannot begin with a "0" index rather a "1". Thus, when I made a signal of 1000 points:
for x=1:1000 signal(x)= …

the algorithm thinks I am actually dealing with 1001 points as the Fourier transform algorithm assumes that the first point actually occurs at x=0. Thus, everything is fine when I used (N-1) = 1000 because 1000 is N-1 from the point of view of the Fourier transform algorithm.

Also, I left out the "a" term because I made it 1 second in the previous examples; I did this so that all of this at once would not overwhelm you. Thus, if we re-scaled the example we have been using thus far such that 1000 seconds was actually 5 seconds (and did the transform):

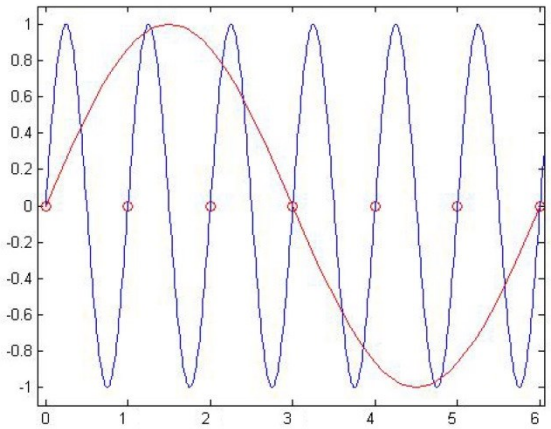


The frequencies have increased by a factor of 200, or 1000/5, which is correct.

While it seems that we have opened a can of worms when we change the length of our data sets and the spacing between each point, actually, note that Matlab doesn't care what your x-axis is. It is going to do Fourier transforms with the idea that each datastep is "1" unit long. As a result, you can keep using the code I gave you above, and re-calculate the real x-axis after the fact. Here is an example for the above:

```
>>length=5;
>>for i=1:1000 xaxis(i)=2*pi*(i-1)/length;  end;
>>plot(xaxis,fttot);
```

Thus, in my algorithm, make the length equal to the number of data points. Calculate the Fourier transform, and then recalculate the correct frequency axis based on the length of the signal given its proper units.



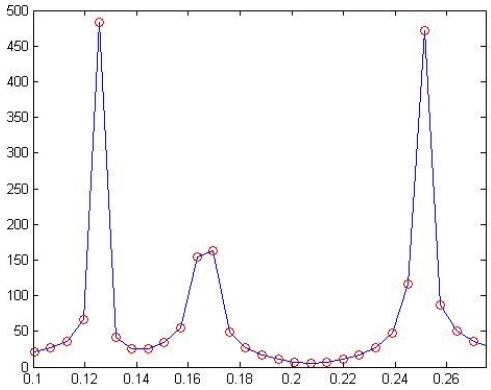
Why do the frequencies work this way? It's because there is a fundamental limit on the frequencies that can be explored. The highest frequency has a wavelength equal to the spacing between points as shown here in blue. While 0 is the lowest allowed frequency, the next lowest is the length of the data set itself as shown in red.

Now remember how when we increased the number of data points that the spectrum became much sharper? While that was because we used more data points in time, but what if we had sampled more frequencies, in other words, what if we inserted frequencies such as:

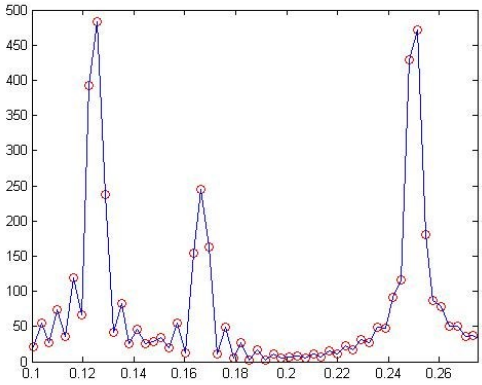
$0, \frac{\pi}{\text{length}}, \frac{2\pi}{\text{length}}, \frac{3\pi}{\text{length}}, \frac{4\pi}{\text{length}}, \ldots$

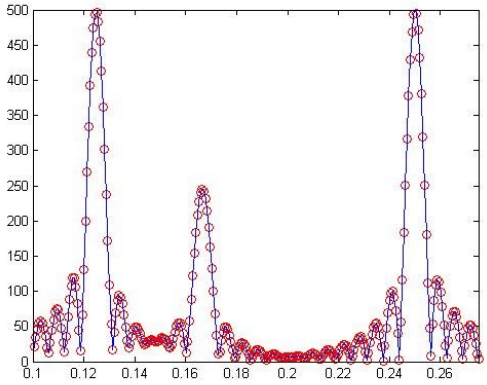
where $\frac{\pi}{\text{length}}, \frac{3\pi}{\text{length}}$, etc. are not normally allowed? Wouldn't that add data that would make the peaks narrower?

Let's just try it with the usual dataset. Below are the peaks with data points included calculated using the first algorithm.



Next, I doubled the frequencies examined on the left, and up a factor of 10 on the right:





Note that you're not helping- the peaks are not more narrow and if anything you're adding a lot of noise to the spectrum. You're also adding small peaks near the main ones- what if you interpreted these peaks are real? They aren't- this is called an aliasing error.

5.4 Inverse Fourier Transform.

If you can Fourier transform a signal, you should be able to Fourier transform the Fourier transform back into the original; this is called an inverse Fourier transform (often referred to as a backward or back transform). There is a slight difference; instead of this expression for the forward transform:

$$Y(k) = \int_{-\infty}^{\infty} y(x) \cdot e^{-2 \cdot i \cdot \pi \cdot x \cdot k} \cdot \partial x$$

The backward one is:

$$y(x) = \int_{-\infty}^{\infty} Y(k) \cdot e^{2 \cdot i \cdot \pi \cdot x \cdot k} \cdot \partial x$$

It's mostly different due to the lack of a negative sign in the exponential.

So, let's start over from the beginning by first writing a significantly better forward Fourier transform code:

```
length=1000;
i=sqrt(-1);
for j=1:length
    forward(j)=0;
    for a=1:length
        a2=2*pi*a/length;
        forward(j)=forward(j)+sig(a)*exp(-i*a2*(j-1));
    end;
end;
```

Note that I am making use of the fact that Matlab can understand how to deal with $e^{-2 \cdot i \cdot \pi \cdot x \cdot k}$ terms. This is especially useful for the backward transform, because the new signal (the transform) is imaginary. Here is a code for the back transform:

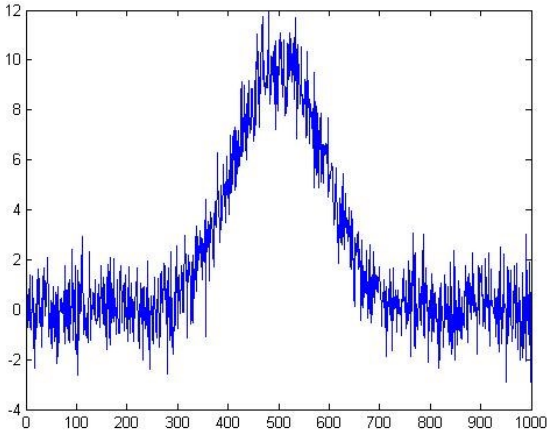
```
length=1000;
i=sqrt(-1);
for j=1:length
    back(j)=0;
    for a=1:length
        a2=2*pi*(a-1)/length;
        back(j)=back(j)+1/length*forward(a)*exp(i*a2*j);
    end;
end;
```

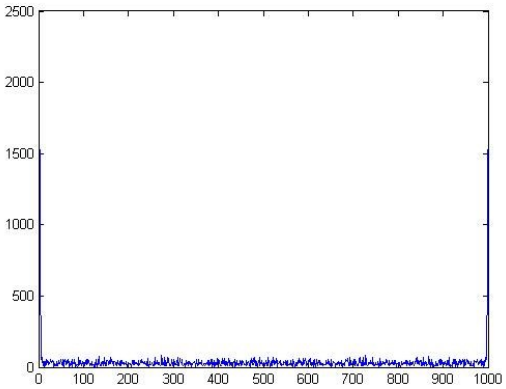
So, you can go forward and backward all day. So what? There is a method for removing noise from a spectrum called optimal windowing that uses forward and backward transforms; here is how it works. Let's make a noisy signal:

```
for i=1:1000
sig(i)=10*exp(-(i-500)^2/15000)+randn;
end;
plot(sig);
```

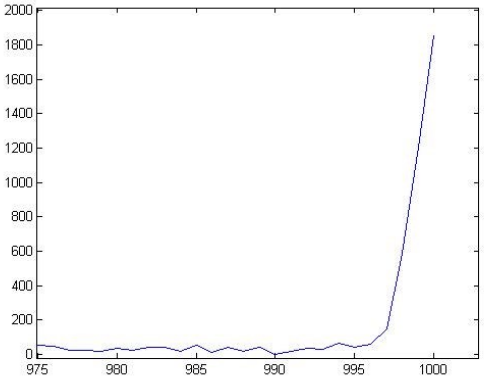
And now make the Fourier transform and plot it:

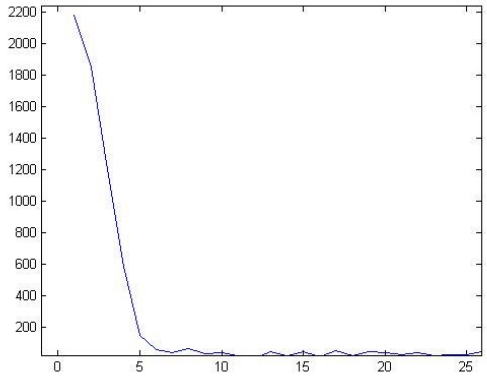
```
plot(abs(forward));
```





I don't see a lot here, if I zoom in the left and right sides I see:

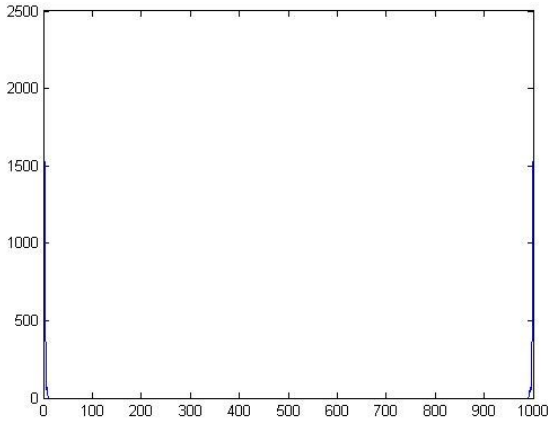




What you are seeing is that there is a significant signal at the lower frequencies (0→5 and 995→1000), while the higher frequencies (6→995) appear like noise. This is ironic- look closely at the noisy spectrum; imagine what kind of frequency signals would make that noise. Are they high frequencies? You bet, so we are going to remove all the high frequency components from the Fourier transformed spectra:

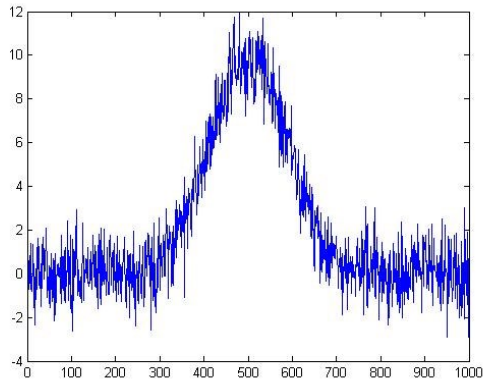
```
for i=5:995
forward(i)=0;
end;
```

Here is the new, cleaned Fourier spectrum:

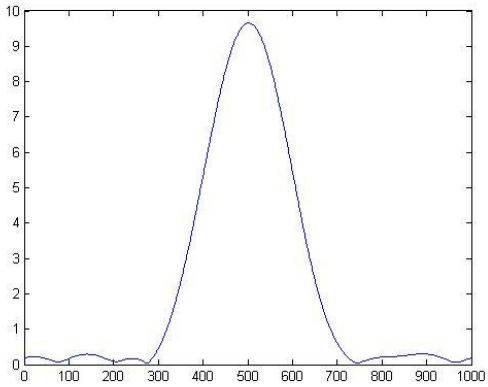


Now when I do the backward transform I get this (the old on is on the left for comparison):

Old



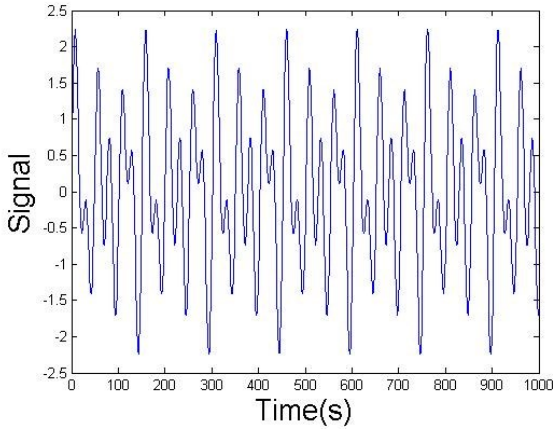
New

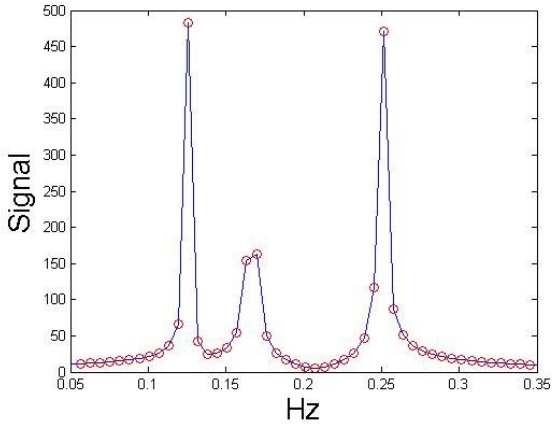


Quite a trick, isn't it?

5.5 Windowing a Fourier Transform.

There is just one last thing to learn about Fourier transforms, which is the signal windowing technique. Let's go back to dataset5.

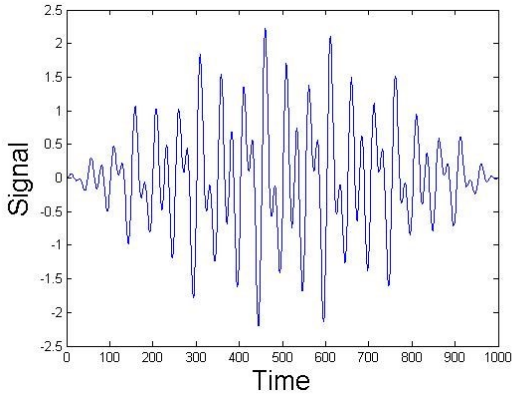


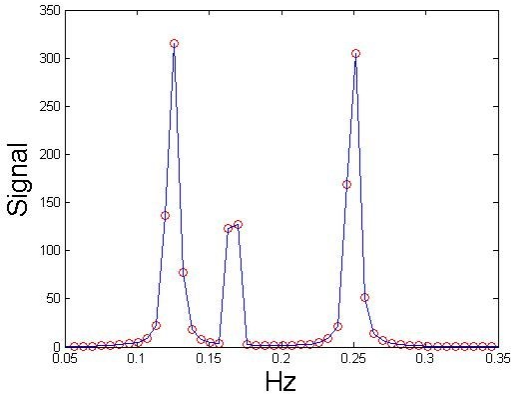


Note that the Fourier transform signal on the left has a somewhat uneven baseline. This is a result of the way the transform calculates data on the left and right sides of the total signal; there is some "badness" as the data simply come to an abrupt end. As such, you can smooth the baseline by multiplying the data by a window (in a process called windowing). Here I multiply the signal by a half sine wave:

$$w(n) = \sin\left(\frac{\pi n}{N-1}\right)$$

and transform the data:

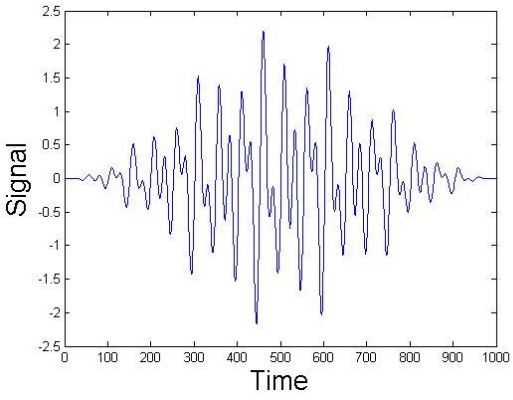




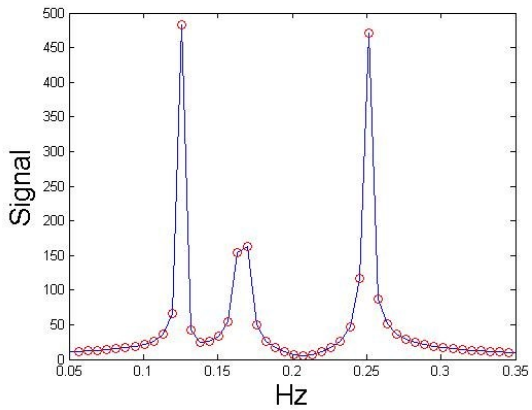
See how more smooth and flat the baseline is now? Especially for the middle peak. This is called a cosine power window. Over the years, mathematicians have developed a large number of window functions, see:
http://en.wikipedia.org/wiki/Window_function
I will provide one more example here, the Hanning window:

$$w(n) = 0.5\left(1 - \cos\left(\frac{2\pi n}{N-1}\right)\right)$$

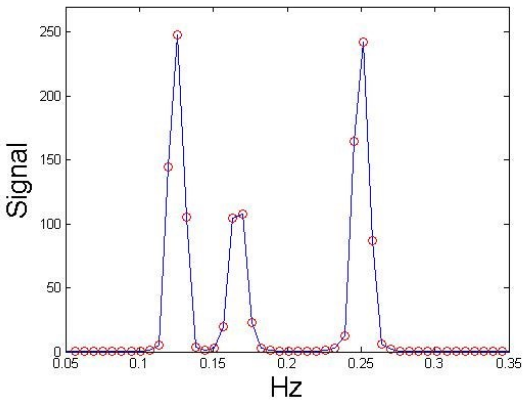
Which make the data look like this (unfiltered transform is on the left for comparison):



Old



Windowed



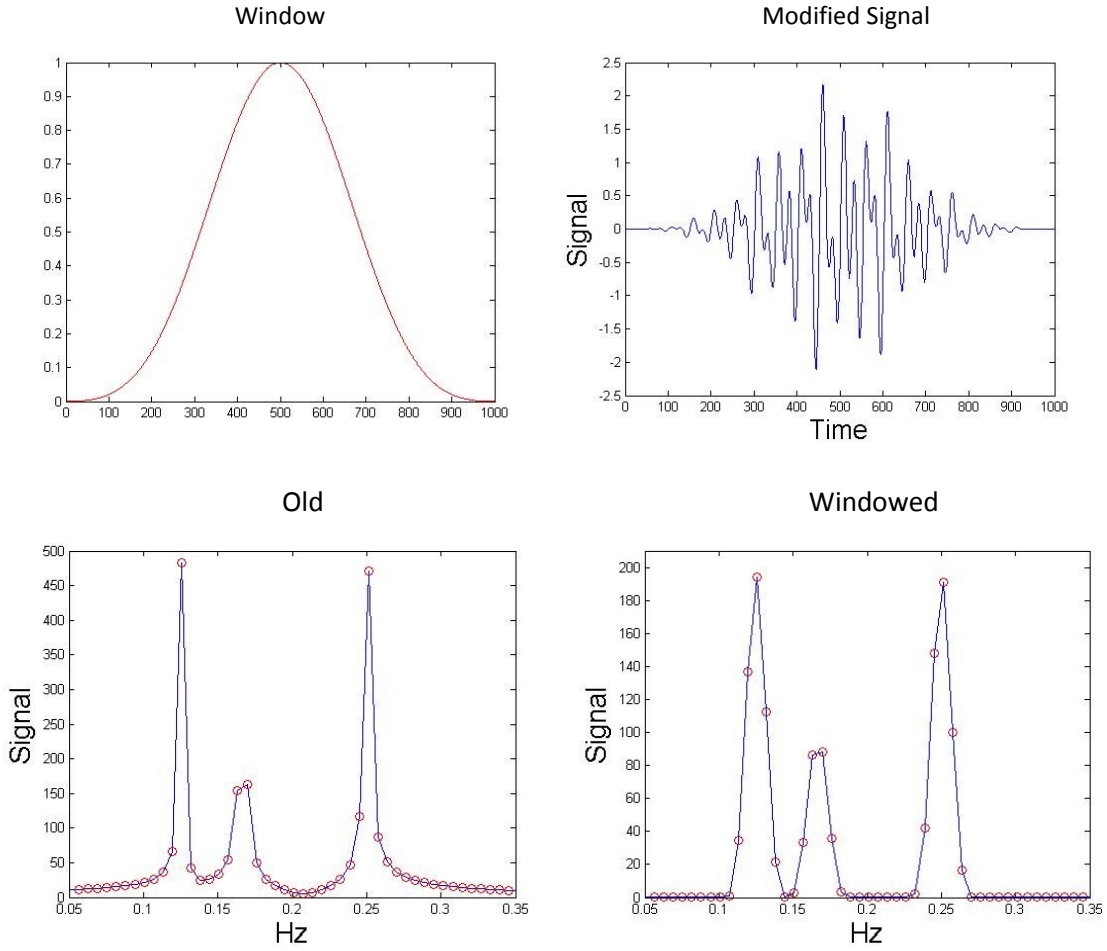
Almost all Fourier transform instruments window their data. One last example of a very common window is the Kaiser-Bessel function:

$$w(n) = \frac{I_0\left(\pi\alpha\sqrt{1-\left(\frac{2n}{N-1}-1\right)^2}\right)}{I_0(\pi\alpha)}$$

Here I_0 is a Bessel function, and α is usually the number 2 or 3. Here is how to program it into Matlab, assuming 1000 points as in dataset5:

```
for i=1:1000
win(i)=besseli(3,pi*3*sqrt(1-(2*(i-1)/(1000-1)-1)^2))/besseli(3,pi*3);
end;
```

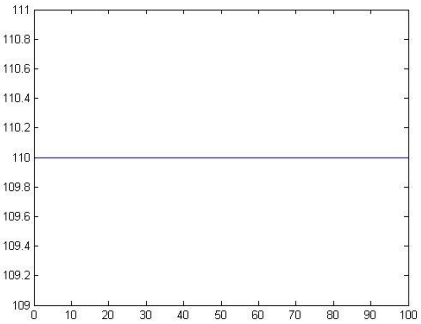
The results are:



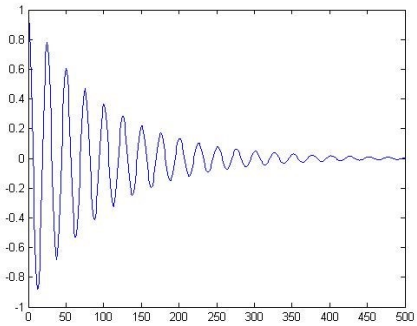
You can see that this window is very popular due to its ability to completely remove a baseline. Note that the overall signal does decrease, a drawback to this method.

Matlab Assignment

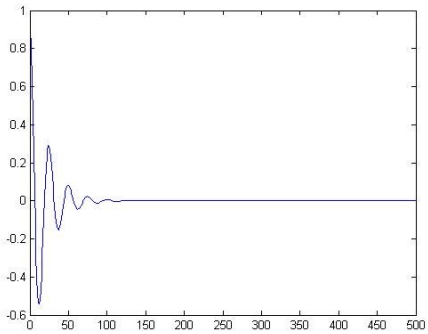
1. a. Fourier Transforms. Calculate the absolute value of the Fourier transform for the Problem5_1.txt data set shown below.



b. Calculate the absolute value of the transform for the Problem5_2.txt data set as well as the characteristic frequency.

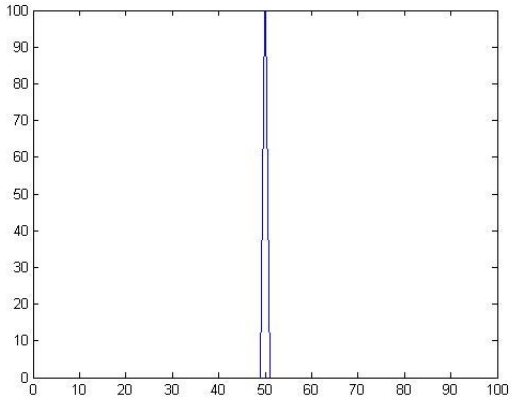


c. Calculate the absolute value of the transform for the Problem5_3.txt data set as well as the characteristic frequency. Compare and contrast the results from pt. b.

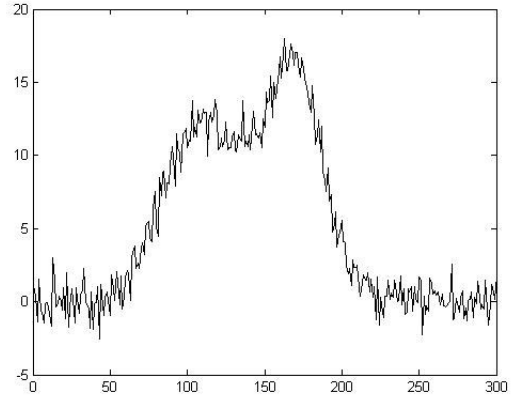


d. Apply a sine window (or any window) to the data in pt. c (Problem5_3.txt).

e. Calculate the real and imaginary Fourier transforms for the data in Problem5_4.txt shown below.

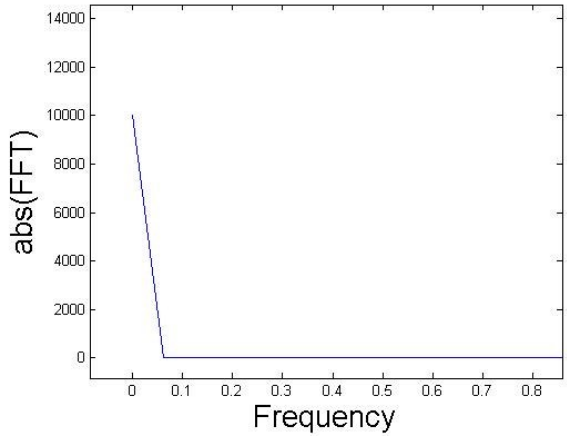


f. Use the optimal filter technique to "clean" up the following data in Problem5_5.txt.

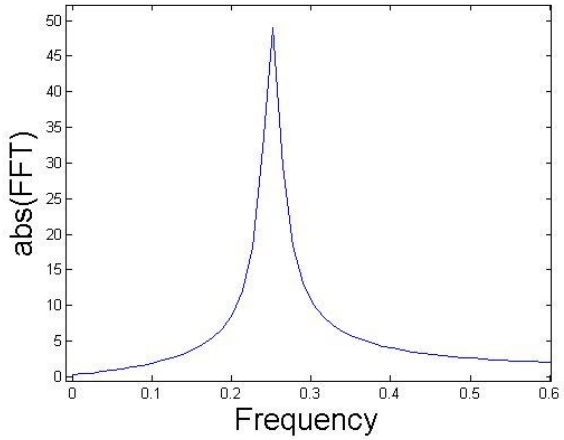


Answers:

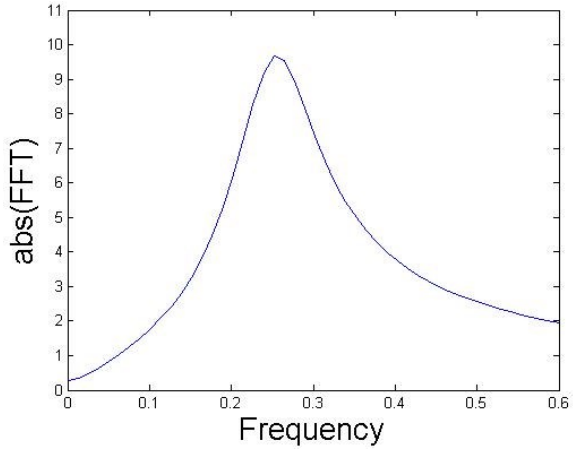
a. As there is no real signal, the FFT is singularly peaked at 0 frequency:



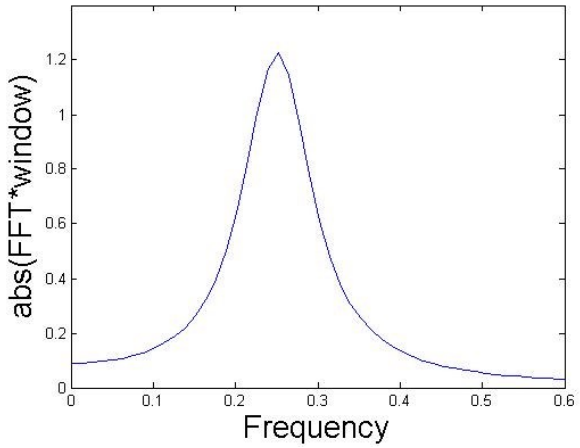
b. The frequency is 0.252 Hz.



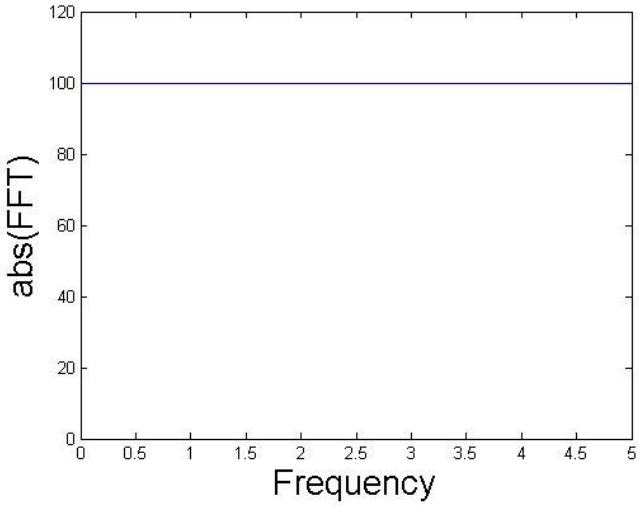
c. Basically the same, but the fact that the signal is shorter broadens the signal.



d. With a sine window, the data are a bit more clear.



e. The point of this question is for you to see that a single sharp point has a FFT that is basically a flat line.



f. At this point this should be fairly easy to do: