Chapter 1: Matlab Basics

1.1 Entering data into memory with Matlab

```
>>x=3 <enter>
x =
    3
>>
```

A ; (semicolon) in matlab is the same thing as pressing <enter> inside a program; its use is necessary as when you execute a program you are not expected to hit <enter> a bunch of times (we will cover more of this later). When you use ; in the consul, it doesn't do much:

```
>>x=3;
>>
```

It simply causes the program not to repeat what your assignment is like in the first example. If you forget what x is equal to later on, do this:

```
>>x <enter>
x =
    3
>>
```

Vectors: Now we will make x a vector as follows.

```
>>x(2)=4
x =
    3    4
>>
```

Or try this:

```
>>x(2)=4;
>>
```

To see what x is now equal to:

```
>x
x =
    3    4
>>x(1)
ans =
    3
>>x(2)
ans =
    4
>>
```

Now let's take a line of data from a .txt file, you should highlight the data (skip the header Wavelength Absorption) and copy it (ctrl c), here is just an illustration:

Wavelength Absorbance (***SEE Dataset1_1.txt***)
500 0.1
501 0.2
502 0.3
503 0.2
504 0.1
<ctrl c>

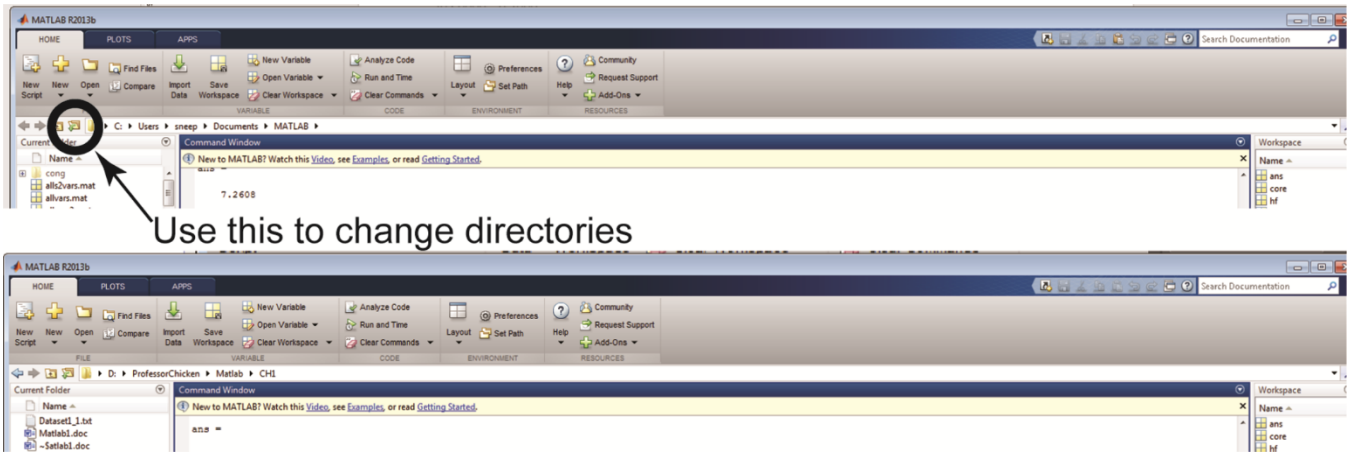
Now lets go back to the matlab window and type the following:

>>spectra=[(now hit <ctrl v> and this becomes)
>>spectra=[500 0.1
501 0.2
502 0.3
503 0.2
504 0.1
(now type in] and hit enter, making it look like this):

>>spectra=[500 0.1
501 0.2
502 0.3
503 0.2
504 0.1
] <enter>

spectra =
500.0000 0.1000
501.0000 0.2000
502.0000 0.3000
503.0000 0.2000
504.0000 0.1000
>>

Congratulations! You have now loaded in a spectrum from your data. Alternatively, you can change the directory that Matlab is in as shown here:



Move to the one that holds Dataset1_1.txt and type in the following:

```
>> spectra=load('Dataset1_1.txt');
```

Regardless of how you load the data, the x-axis can is now the first column of the Matlab variable spectra:

```
>>spectra(:,1)
ans =
  500
  501
  502
  503
  504
```

And the y-axis is

```
>>spectra(:,2)
ans =
   0.1000
   0.2000
   0.3000
   0.2000
   0.1000
```

Notice what : does? In the above examples, it means “show me all the numbers in this column.” In short, : means “everything”. Here is another use:

```
>>spectra(:,2)=spectra(:,2)-0.1
spectra =
  500.0000        0
  501.0000    0.1000
  502.0000    0.2000
  503.0000    0.1000
  504.0000        0
>>
```

See! You just baseline corrected the data! Here is another example:

```
>>spectra(:,2)=spectra(:,2)*10
spectra =
  500.0000        0
  501.0000    1.0000
  502.0000    2.0000
  503.0000    1.0000
  504.0000        0
```

Great if you want to scale your data for visual purposes. Here is another example:

```
>>spectra(:,2)=1
```

```
spectra =
   500     1
   501     1
   502     1
   503     1
   504     1
```

Notice you just make a mass-assignment of the second column in spectra. Useful, but you can wipe out all your work fairly easily. Now cut and paste the original spectra data from above back into Matlab:

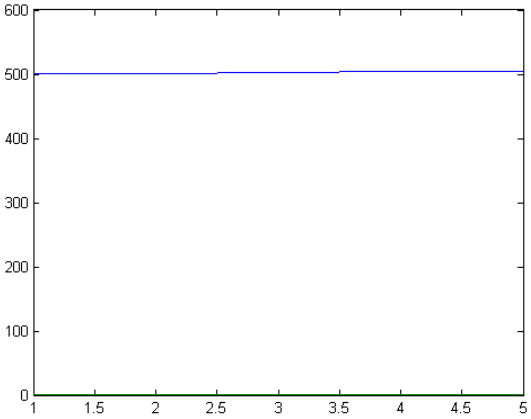
```
>>spectra=[500          0.1
501       0.2
502       0.3
503       0.2
504       0.1
] <enter>
```

1.2. Figures

Let's make a figure!

```
>>plot(spectra)
```

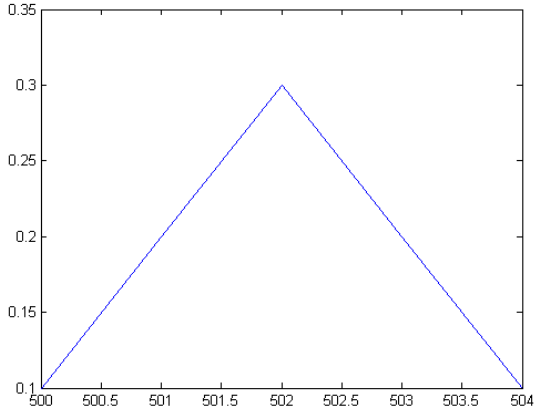
and you get this:



The reason this looks like cat vomit is because matlab doesn't know what you mean to use for the x-axis and the y-axis. The format for the plot command is plot(x-axis,y-axis) so type in:

```
>>plot(spectra(:,1),spectra(:,2))
```

and you get:



Try these commands in this order, and watch what happens after you type each one in:

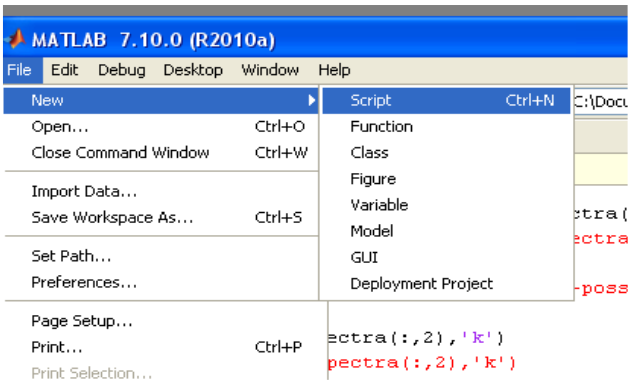
```
>> plot(spectra(:,1),spectra(:,2), 'ro')
>> hold on
>> plot(spectra(:,1),spectra(:,2),'g')
>> hold off
>> plot(spectra(:,1),spectra(:,2),'k')
>> xlabel('Wavelength')
>> ylabel('Emission')
```

Already better than Excel, isn't it! To save this kind of figure for your report, type the following:

```
>>print -djpeg figure1.jpg
```

1.3. Now let's make a script

A script is a series of commands stored in a file which will be executed all at once. This is much better for when you have to make a series of analysis, but if you screw up, you don't have to retype anything. It also makes it easier to spot an error before you have executed a command.



To make a script, go to file, and select new M file. A window like notepad will open. In the is window, type the following commands (i.e. don't be an idiot- cut and paste the following):

```
clear all;
close all;
spectra=[500    0.1
501     0.2
502     0.3
503     0.2
504     0.1];
spectra(:,2)=spectra(:,2)-0.1;
plot(spectra(:,1),spectra(:,2));
hold on;
plot(spectra(:,1),spectra(:,2),'ko');
```

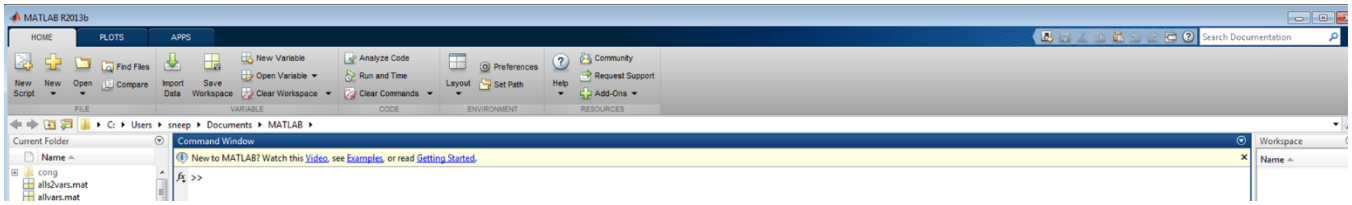
Note the use of the semicolon on every line!!!! Go to file and save. You have to make up a name, I will use snarf. Now, go back to the Matlab consule and type:

```
>>snarf  <enter>
```

And what happens? The data are created, baseline corrected, and a figure is plotted with the data! All at once, so you can see, if you make some kind of mistake you can just easily edit the .M file without having to type everything all over again.

1.4 Loading data

The cut and paste thing is the easiest way to load data, but what if you have a lot of data and that's just impractical? Matlab can load data directly, but there are just two issues you have to address. First, notice that the status bar ("Current Folder:") at the top of the matlab window had your present working directory:



Generally, it is something like: C:\Documents and Settings\My Documents\MATLAB. Problem- the data file has to be in the same directory, or you have to move the present working directory to the one with the data. To move directories, hit the […] button next to "Current Folder:" which will allow you to move into another directory. Next issue, your data must be a table of numbers with no other extraneous information. For example, let's look at the data set (Dataset1_2.txt) that is provided in this packet:

```
Wavelength        Absorbance
  5.0000000e+02   2.6837674e-02
  5.0100000e+02  -1.9768694e-02
  5.0200000e+02   3.6358854e-02
…
```

Now type this to load it in (and note the response after):

```
>>mydata1=load('Dataset1_2.txt');
Error using load
Unknown text on line number 1 of ASCII file Dataset1_2.txt
"Wavelength".
```

Whups! Matlab doesn't know what to do about the text in the first line, and how that meshes with the numbers that come after. Here is what you do- you cut out the first line in notepad, and then save it into your present matlab working directory. With that part out of the way, do the following in the consul or in your script:

```
>>mydata1=load('Dataset1_2.txt');
```

Did it work? Hard to tell because matlab hasn't acknowledged anything yet. There are two ways to know: first, do what I already showed you, type in the variable name and hit enter:

```
>>mydata1 <enter>
mydata1 =
  5.0000000e+02   2.6837674e-02
  5.0100000e+02  -1.9768694e-02
  5.0200000e+02   3.6358854e-02
  …
>>
```

Yipee! Here is another way: type who

```
>>who <enter>
Your variables are:
ans     mydata1   spectra
```

It lists all the variables you presently have saved to memory. You can use this at anytime to remember what you are working with and what you are not.

1.5 Programming commands

Let's write our first simple program on the consul. This is a for loop which will repeat a command several times (here, 5 times). The variable i is equal to 1 on the first loop and 5 on the last. Let's use this:

```
>>for i=1:5 kitty(i)=i/10; end; <enter>
>>
```

Again, you don't know what matlab did. Ask it!

```
>>kitty <enter>
kitty =
   0.1000    0.2000    0.3000    0.4000    0.5000
>>
```

Now lets say you meant to create a vector from 0 to 0.4. There are two ways to fix it:

```
>>kitty=kitty-0.1;
```

Here is another: hit the up arrow until the original program appears:

```
>>for i=1:5 kitty(i)=i/10; end;
```

Now move the cursor left and alter the command as so:

```
>>for i=1:5 kitty(i)=(i-1)/10; end; <enter>
>>kitty
kitty =
       0    0.1000    0.2000    0.3000    0.4000
>>
```

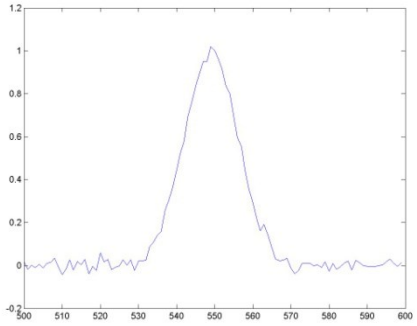
See? You can redefine the variable i within the loop to suit your needs. Note, you cannot have fixed this problem by using:

```
>>for i=0:4
```

Just try it! Now you can imagine that this can get complicated, which is why you normally put programming instructions in an M file because your going to make a lot of mistakes when your typing this stuff in the consul (and you will screw up the data often).

Here is the best part- the for loop is pretty much the only programming function you need to use!

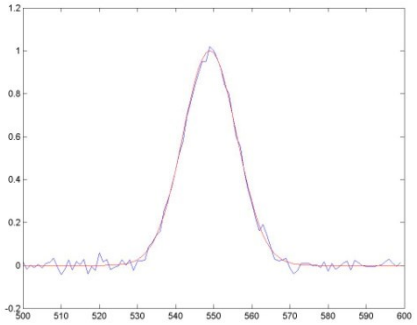
1.6 Error Analysis with Matlab; calculating R^2



Let's look at the set of data we just loaded in (it's an emission spectrum) and fit it. While we can do a good job doing that, I want to know how good with numbers. In this case, we want to calculate the "goodness of the fit", aka R^2. Let's use matlab to calculate this quantity on the data set that you have already loaded (mydata1). First, plot mydata1 so that you have an idea what your data look like:

```
>>plot(mydata1(:,1),mydata1(:,2))
```

Looks like some sort of emission spectrum, fairly Gaussian (bell-shaped). Now our job is to calculate R^2 of our fit to this data. First, I will define my best fit to the data:



```
>> for i=1:100 fit(i)=exp(-(i-50)^2/100); end; <enter>
```

Where did I get the parameters for this fit (50, 100)? We will be doing that part in the next rotation. For now, let's see if I did a good job as evident from a high R^2 value:

```
>> hold on
>> plot(mydata1(:,1),fit,'r') <enter>
```

Looks good! But how good? R^2 is calculated from the sum of the squares of the differences of the data from the fit: $\sum(\text{data} - \text{fit})^2$. This result is divided by the sum of the square of the data points minus the average value of all those same data points: $\sum(\text{data} - \text{mean}(\text{data}))^2$. Finally, R^2 is calculated as:

$$R^2 = 1 - \frac{\sum(\text{data} - \text{fit})^2}{\sum(\text{data} - \text{mean}(\text{data}))^2}$$

I will show you how to translate this into Matlab-ese below.

I first note that I need to do a summation. A for loop can be used for this purpose. Here is an example:

```
>> part1=0;
>> for i=1:100 part1=part1+(mydata1(i,2)-fit(i))^2; end;
```

This calculates the numerator. As for the denominator:

```
>>part2=0;
>> meandata=mean(mydata1(:,2));
>> for i=1:100 part2=part2+(mydata1(i,2)-meandata)^2; end;
```

See your done!

```
>>r2=1-part1/part2
r2 =
   0.9957
```
And there is your answer! An R^2 of .99 or greater represents a very good fit.

1.7 Conclusion: Other useful commands

Now we have done some processing of the variable spectra (we baseline corrected it), you don't want to have to this over and over again. To save the data, close matlab and come back to it later, just type the following:

>>save mydata1 mydata1 <enter>

Now you can load it in later. DO NOT FORGET TO TYPE THE VARIABLE NAME TWICE! It's a long story, but that can cause some major problems later. If you do it accidentally, just retype

>> save mydata1 mydata1 <enter>

again, and it will be fixed. More on what is happening with this later on..

ls or dir -lists all files in your present working directory

```
>>ls
.          Thumbs.db     snarf.m       figure1.jpg   mydata1.txt
..         temp.txt          mydata1.mat
>>
```

Don't forget we have a .M file present when we wrote the "snarf.m" script and we have a .MAT file present as just saved the variable spectra. Thus, scripts are labeled .m and variables .mat.

cd .. -change directory one level up

cd (name) -move into the directory named name.

pi -π, or 3.1416

sin(pi) - the sine of pi. Anything can be given to sin(?), let's try this:

>>sin(mydata1)

```
ans =
  -0.4678    0.0268
  -0.9965   -0.0198
  -0.6090    0.0364
   …
```

See? The sine of every element of mydata1. Other trig functions are cos, tan, etc. See, its all fairly intuitive.

abs -absolute value

exit -quit the program

Chapter 2. Matlab Functions

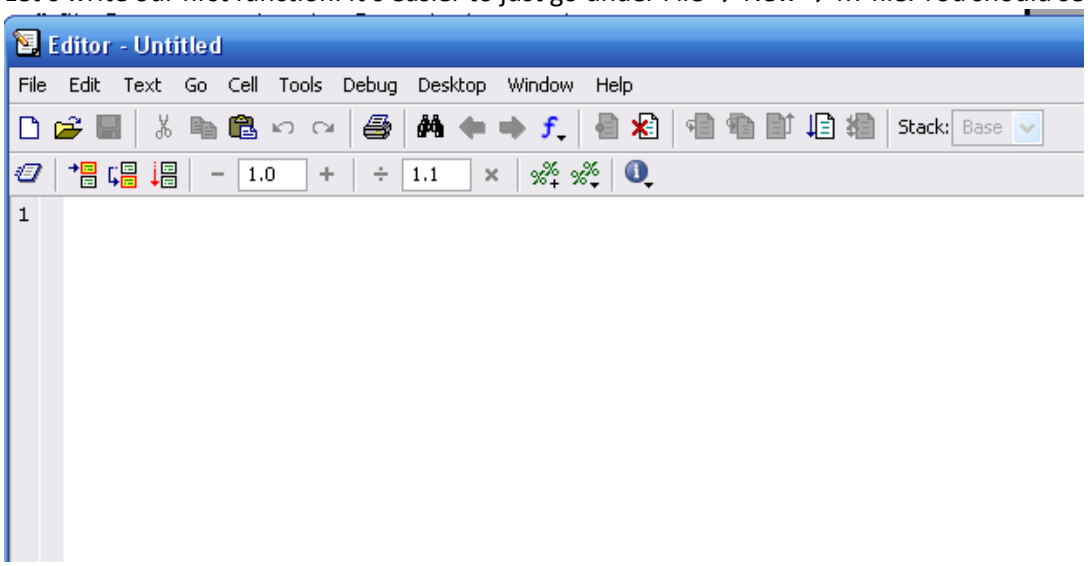
1.2 Writing functions in Matlab

In Matlab, every mathematical function (such as sin) is actually a series of instructions in a "function_name.m" file. In my example, when I type in the consul:

```
>>sin(3.14)
ans =
   0.0016
```

What is happening is the number 3.14 is being sent to file sin.m for processing such the function returns the value 0.0016. (in my computer, the file sin.m is located at: C:\ProgramFiles\MATLAB\R2006b\toolbox\eml\lib\matlab). The interesting thing is you can write your own functions that can do whatever you want, like calculate the error in a fit to your data.

Let's write our first function. It's easier to just go under File → New → M-file. You should see this:



On the first line type:

```
function [return_val]=fitter(x)
```

Now here is what these things do:

function → This tells Matlab that your writing a function. What this means is that the function is "blind" to the consul (the thing you're typing commands in). If dataset has been loaded into the consul, the function still cannot use it. Likewise, if dataset is altered in the function, it is not altered in the consul- several examples of what this means are given below.

[return_val] → This is the value that the function will return, in our example above for sin(3.14), it was 0.0016.

fitter → The name of the .m file. When you save it, make sure you save the name of the file as "fitter".

(x) → This is what you pass to the function, in our previous example of sin(3.14), x is equal to 3.14 inside the function. It can also be a vector, in other words, it can have two values; x(1) and x(2) for example.

Now let's write a function that multiplies two numbers together. In your .m file, write the following:

```
function [return_val]=fitter(x)
kitty=x(1)*x(2);
return_val=kitty;
```

Now save the .m file as fitter.m. In the consul, type:

```
>>x(1)=5; x(2)=6;
>>fitter(x)
ans =
   30
```

See? It's really easy. Note the following: change your .m file as:

```
function [return_val]=fitter(x)
i=444;
kitty=x(1)*x(2);
return_val=kitty;
```

Save it and type the following:

```
>>x(1)=5; x(2)=6; i=666;
>>i
i =
  666
>> fitter(x)
ans =
   30
>> i
i =
  666
```

Note that the value of i appears to be redefined in fitter.m from the value 666 to 444; however, it is actually unchanged from your assignment of 666 in the consul. This is what I mean when I say that the consul is "blind" to the actions of the function and vise versa.

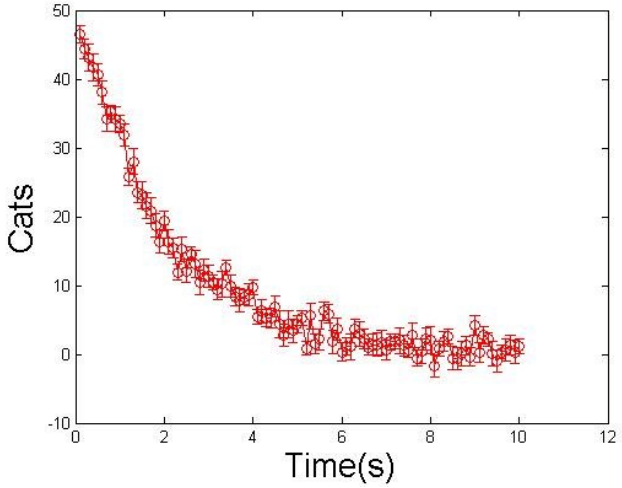
Now let's write something relevant- a function that can calculate the error of a fit. By error, I mean χ^2 (chi squared). From your laboratory manual, χ^2 is:

$$\chi^2 = \sum_{i=1}^{N} \left(\frac{1}{\sigma(i)^2}\right) [\text{data}(i) - \text{fit}(i)]^2$$

where the summation is over the total number (N) of data points and $\sigma(i)$ are the error(s) of each data point. Let's write a function that calculates χ^2 for a dataset that represents the decay of cats over time in a grinder. The dataset, called "Dataset2_1.txt" which a part of this packet, has time in the first row, number of cats in the second, and σ of each data point in the third row. First let's plot the data with error:

```
>> mydata2=load('Dataset2_1.txt')
>> errorbar(mydata2(:,1),mydata2(:,2),mydata2(:,3),'ro-')
>> d=xlabel('Time(s)');
>> set(d,'FontSize',20)
>> f=ylabel('Cats');
>> set(f,'FontSize',20)
```

The output is:



We wish to model the experimental data with an exponential decay:

$$\text{Cats} \cdot e^{-t/\tau}$$

and calculate χ^2 of that fit. The following .m file does just that:

```
function [return_val]=fitter(x)
mydata2=load('Dataset2_1.txt');

chisq=0;
for i=1:100
    fit(i)=x(1)*exp(-mydata2(i,1)/x(2));
    chisq=chisq+(1/mydata2(i,3)^2)*(mydata2(i,2)-fit(i))^2;
end;
return_val=chisq;
```

Here the number of cats is represented by x(1), the decay time constant τ is represented by x(2), the time variable t is the first column of mydata2, the experimental results are in the second column, and the error in the measured number of cats per unit time σ (as they tend to get mashed up together) is in the third column. Note that there are 100 data points. Let's look at some salient features of this file. You have to load the dataset mydata2.txt (and make sure your consul, your

fitter.m file, and mydata2.txt are all in the same directory!); this is a result of the "disconnect" between the consul and the function. Just because the dataset mydata2.txt may be loaded into the consul means nothing to the function! You must also pass to the function the vector x that has the fit parameters. The function then returns the value of chi squared.

To solidify this, let's use it and see that happens! I will first make a guess at what the parameters are that describe this horrible decay of cats. It seems that we started with 50 cats. Also, they are mostly gone within 3 seconds. As such, my guess is:

```
>>a(1)=50;
>>a(2)=4.0;
>>fitter(a)
ans =
  3.9663e+03
```

Where the ans is χ^2 of the exponential decay fit described the parameters a(1) and a(2). Note that we can use this to find the best fit parameter. For example, change a as:

```
>>a(2)=2.0;
>>fitter(a)
ans =
  124.0156
```

It's a better fit since the sum of the differences with the data is less! Let's keep going!

```
>>a(2)=1.0;
>>fitter(a)
ans =
  2.0833e+03
```

Oops, that's worse. Well, what you have to do is keep changing the variables until you get the lowest result. It may take hours!
Just kidding. Matlab has a solution: type the following:

```
>> fminsearch('fitter',a)
ans =
   50.2501    2.0113
```

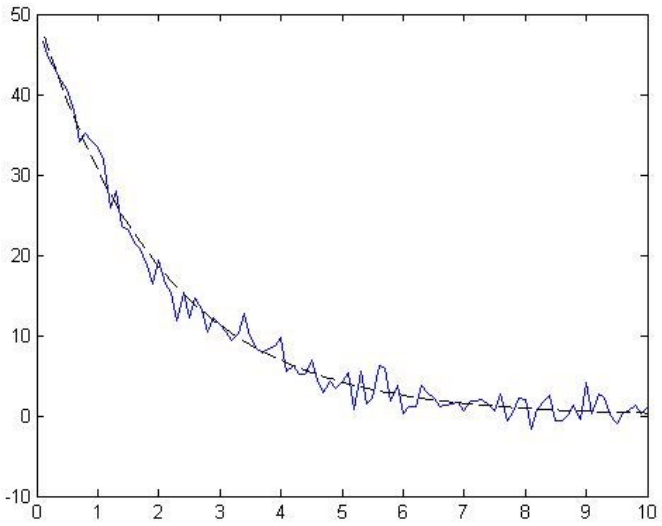
This is what is happening- fminsearch is querying the result of fitter.m for your value of "a". Next, it starts changing "a" in a directed way to lower the result. After a fair number of trials, it will give you its best result which is the value(s) of "a" that give the lowest possible sum of the differences in the data and the fit. Now type this:

```
>>a
a =
    50     1
>> fitter(a)
ans =
  2.0833e+03
```

```
>> fminsearch('fitter',a)
ans =
  50.2501    2.0113
>> a2=ans;
>> fitter(a2)
ans =
  123.2405
```

Clearly, the exponential decay described by the vector a2 is much better than our original guess. You should always double check the results however:

```
>> for i=1:100 fit(i)=a2(1)*exp(-mydata2(i,1)/a2(2)); end;
>> plot(mydata2(:,1),mydata2(:,2))
>> hold on
>> plot(mydata2(:,1),fit,'k--')
```



See? This looks like a very good fit. You can use this type of analysis for your gas effusion data!

Here is the most important point- when you make your own program, you don't have to necessarily use the equation of a line to fit non-linear data!

FYI: If you are totally stuck or make a mistake and can't figure it out, the file fitter.m is included in this tutorial.

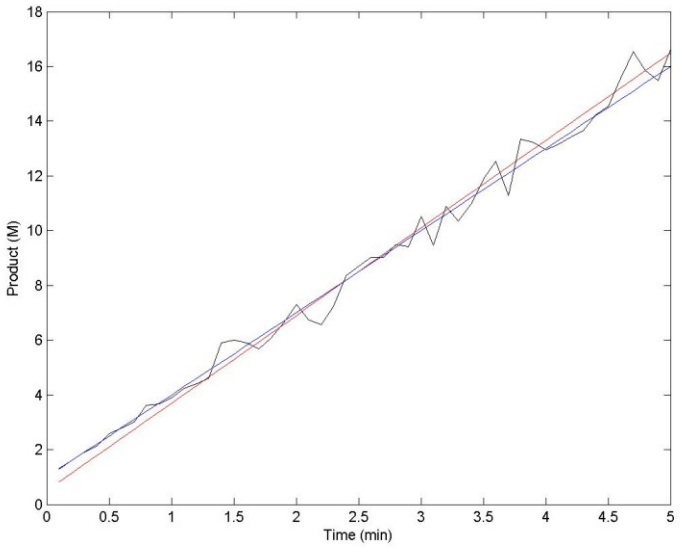
NOW YOU CAN FIT DATA TO ANY FUNCTIONAL FORM

Matlab Assignment

1. With this packet is a dataset (Problem2_1.txt) consisting of 2 by 50 points; the first column is time and the second is product formation. It is roughly linear. Your job is to calculate R^2 for three possible fits using Matlab:

a) A line described by Product(time)=3.2×time + 0.5
b) A line described by Product(time)=3.0×time + 1.0
c) A line described by Product(time)=2.8×time + 1.2

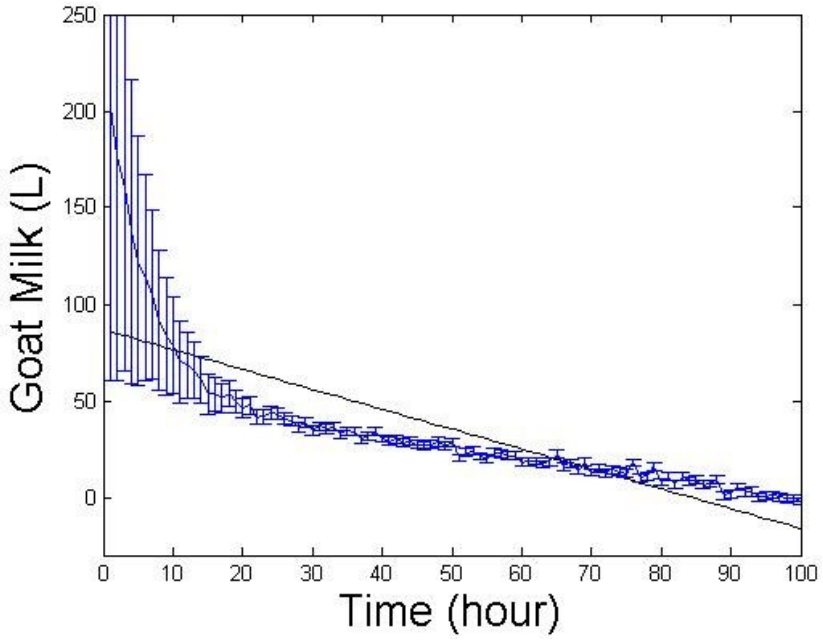
Last part: make a figure of the raw data and all three fits. Here is an example with two of the fits:



2. This exercise is used to show you how important error is to the result of a fit. As part of this packet is dataset "Problem2_2.txt", consisting of 3 by 100 points: the first column represents time, the second Liters of goatmilk, and the third the standard deviation [$\sigma_{(i)}$] of goatmilk that also has units of Liters. It is roughly linear although the first few points have a huge amount of error associated with them due to the early-time randomness of goatmilk production. Use the command:

```
>>errorbar(Problem2_2(:,1),Problem2_2(:,2),Problem2_2(:,3))
```

to see a graph with the standard deviations of the actual data points. First, calculate the linear least squares fit to the data using any available software you would like (note that I have a Matlab script, llsq.m, which calculates this for you and is included in this packet). With the parameters in hand, plot the data and the best fit line; it may look something like:



This is obviously a very bad fit. Your job is to determine the real best fit line by minimizing a MATLAB function that calculates:

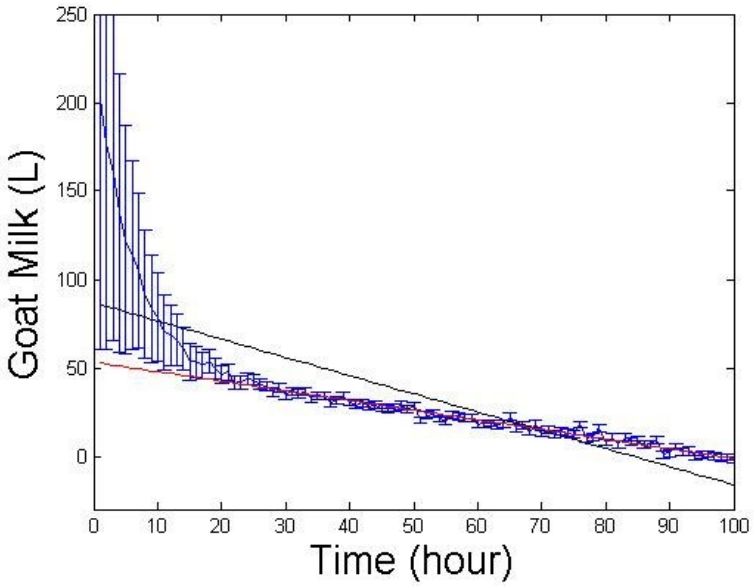
$$\chi^2 = \sum_{i=1}^{N} \frac{1}{\sigma_{(i)}^{\,2}} \cdot (\mathrm{data(i)} - \mathrm{fit(i)})^2$$

where the error of each data point σ(i) is considered. Here is an example to calculate χ^2 (albeit incomplete):

```
chisq=0;
for i=1:100
   chisq=chisq+(1/Problem2(i,3)^2)*(Problem2_2(i,2)-fit(i))^2;
end;
```

Complete the function to calculate χ^2 and then use fminsearch to determine the real best fit slope and intercept to the goatmilk data. When done properly, you should use the result of fminsearch to plot the new best linear fit. Last, send me for full credit a figure with the original data with errorbars, the linear least squares fit, and your own fit. The answer should look like the figure on the next page.

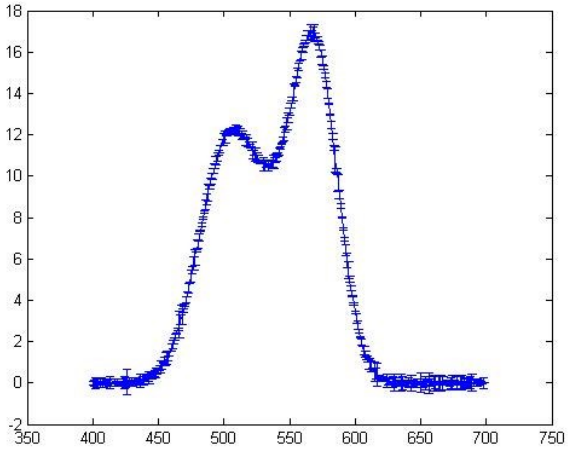
The lower line that takes into account the error of each datapoint is clearly the best fit and is mostly within the error bars.



3. Hidden data. We are using "Problem2_3.txt" as part of this packet. It contains 3 by 300 points: the first column represents wavelength in nm, the second is emission, and third are the error bars. It appears to be composed of two Gaussian peaks. A Gaussian has the functional form of:

$$f(t) = A \cdot e^{-(\lambda-\lambda_{center})/2\sigma^2}$$

where A is the amplitude, λ_{center} is the position of the peak, and σ is related to how wide the peak is.



However, I'm not so sure that this is a doubly peaked emission spectrum; perhaps there are three peaks and the third one is hidden under the other two. First, try to fit these data using fminsearch to a 2-Gaussian fit. Here is how you program a (single) Gaussian in Matlab:

```
for i=1:300
gausfit(i)=x(1)*exp(-(Problem2_3(i,1)-x(2))^2/x(3));
end;
plot(problem5(:,1),gausfit);
```

The amplitude is x(1), x(2) is λ_{center}, and x(3) is $2\sigma^2$. Use fminsearch to calculate the best fit to the data using two Gaussian functions. If (when) the data don't fit well, add a third Gaussian (try an initial guess between the main two peaks). In the three Gaussian fit, you will be optimizing nine parameters. Sometimes this causes Matlab to get "stuck" and give up; if so, you will see the following:

```
>>fminsearch('fitter',x)
Exiting: Maximum number of function evaluations has been exceeded
    - increase MaxFunEvals option.
    Current function value: 0.017758
ans =
  10.1798  500.2755  905.2207   15.0907  569.9621  600.7211    5.8533  535.3556
956.3240
```

If so, try "recycling" the last set of optimized variables as so:

```
>> fminsearch('fitter',ans)
```

You may have to do this several times. Eventually Matlab settles on a set of parameters it thinks is optimal and will no longer give you that error.

And, if you get completely stuck, programs are provided in the packet that answer the question.

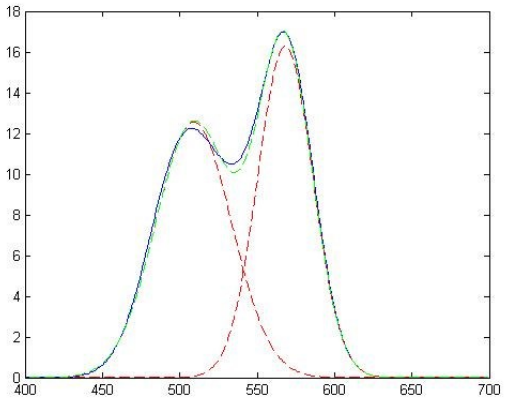
Answers:

1. a) R^2=0.9880 b) R^2=0.9891 c) R^2=0.9759

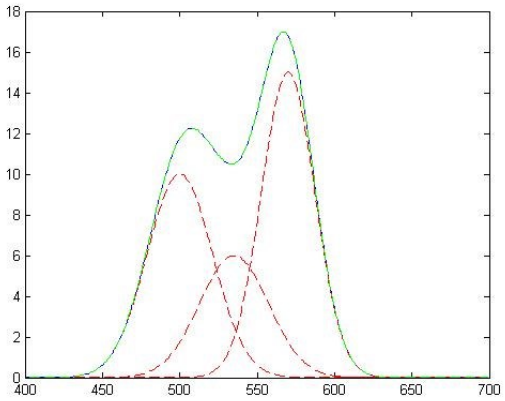
2. linear least squares fit slope= -1.0287 ± 0.0767326
intercept= 86.6580 ± 4.46337

Fminsearch slope=-0.5451 intercept= 53.23

3. If you use two Gaussians, you get this (and a χ^2 of 2.6817e+03).



If you use three Gaussians, you get this (and a χ^2 of 8.9445e-06).



Chapter 3. The Variance-Covariance Matrix

3.1 Non-linear Error Analysis

Our biggest feat so-far has been fitting a linear function to a set of data by minimizing the least squares differences from the fit to the data with fminsearch. When analyzing non-linear data, you have to use a program like Matlab as many types of data cannot be linearized such that Excel can analyze it. This also means that you do not have a direct route to calculating the error of your fit, as you have been using the error calculated from the linear least squares result (this doesn't work on non-linear data!).

Now you can make one of two assumptions- 1) error cannot be calculated from a non-linear fit or 2) error can be calculated from a non-linear fit, I just don't know how.
Unless you're a fre@¿*ħ↗≈*in' moron, you should pick 2.

Here is how! The variance-covariance matrix.

I will not cover the derivation, not that I don't understand it (I so totally do) but it is several pages of algebra long. I will show you the formula for the error analysis and prove it works by applying it to a linear set of data. Then we will use it for other problems…

Step 1: define a matrix of partial derivatives.

This is the only semi-hard part, you have to calculate the partial derivative of the function you are fitting your data to for each variable that you are minimizing in your least squares fit (remember fminsearch from the previous lesson?). Let's say you are fitting data to a function f(m,b) which has two variables that were fit my minimizing the error with fminsearch, call them m and b. The first step to calculate σ_m and σ_b is to derive the partial derivative matrix:
M=

$$\begin{vmatrix} \frac{\partial f(m,b)}{\partial m}\cdot\frac{\partial f(m,b)}{\partial m} & \frac{\partial f(m,b)}{\partial m}\cdot\frac{\partial f(m,b)}{\partial b} \\ \frac{\partial f(m,b)}{\partial b}\cdot\frac{\partial f(m,b)}{\partial m} & \frac{\partial f(m,b)}{\partial b}\cdot\frac{\partial f(m,b)}{\partial b} \end{vmatrix}$$

Where $\frac{\partial f(m,b)}{\partial m}$ is the partial derivative with respect to the variable your fitting (m) and likewise for $\frac{\partial f(m,b)}{\partial b}$ with variable b. Note that the diagonal (1,1) and (2,2) elements are **NOT** equal to the second derivative $\frac{\partial f(m,b)^2}{\partial^2 m}$ but are in fact just the first derivative squared, i.e. $\left(\frac{\partial f(m,b)}{\partial m}\right)^2$. The off-diagonal elements are the partial derivatives multiplied by each other. If this is confusing now, we will make it a lot clearer with an example on the next page. For now, let me simplify the matrix **M** as:

$$\begin{vmatrix} \left(\frac{\partial f(m,b)}{\partial m}\right)^2 & \frac{\partial f(m,b)}{\partial m}\cdot\frac{\partial f(m,b)}{\partial b} \\ \frac{\partial f(m,b)}{\partial b}\cdot\frac{\partial f(m,b)}{\partial m} & \left(\frac{\partial f(m,b)}{\partial b}\right)^2 \end{vmatrix}$$

Now let me be totally honest here: the matrix **M** is actually a sum over all data N points and is properly expressed as:

$$\begin{vmatrix} \sum_{i=1}^{N}\left(\frac{\partial f(m,b)}{\partial m}\right)^2 & \sum_{i=1}^{N}\left(\frac{\partial f(m,b)}{\partial m}\cdot\frac{\partial f(m,b)}{\partial b}\right) \\ \sum_{i=1}^{N}\left(\frac{\partial f(m,b)}{\partial b}\cdot\frac{\partial f(m,b)}{\partial m}\right) & \sum_{i=1}^{N}\left(\frac{\partial f(m,b)}{\partial b}\right)^2 \end{vmatrix}$$

Now before this gets confusing, let's cement everything with a simple example. Here is a set of data that we can fit (Dataset3_1.txt):

x_i	y_i
0	1.9
1	4.2
2	5.9
3	7.8
4	11
5	12.2
6	14.1

You should know how to load these values into Matlab, which we call mydata3. So let's start with a linear fit; first we define the partial derivatives of the function:

$$f(x(i)) = y(i) = m \cdot x(i) + b$$

Given this definition, we can determine that:

$$\frac{\partial f(m,b)}{\partial m} = x(i)$$

and:

$$\frac{\partial f(m,b)}{\partial b} = 1$$

Done! Let's put this into the matrix:

$$\begin{vmatrix} \sum_{i=1}^{N}\left(\frac{\partial f(m,b)}{\partial m}\right)^2 & \sum_{i=1}^{N}\left(\frac{\partial f(m,b)}{\partial m}\cdot\frac{\partial f(m,b)}{\partial b}\right) \\ \sum_{i=1}^{N}\left(\frac{\partial f(m,b)}{\partial b}\cdot\frac{\partial f(m,b)}{\partial m}\right) & \sum_{i=1}^{N}\left(\frac{\partial f(m,b)}{\partial b}\right)^2 \end{vmatrix}$$

which is now:

$$\begin{array}{|c|c|}\hline \sum_{i=1}^{N}(x(i)\cdot x(i)) & \sum_{i=1}^{N}(x(i)\cdot 1) \\ \hline \sum_{i=1}^{N}(1\cdot x(i)) & \sum_{i=1}^{N}(1\cdot 1) \\ \hline \end{array}$$

Simplifying gives:

$$\begin{array}{|c|c|}\hline \sum_{i=1}^{N}x(i)^2 & \sum_{i=1}^{N}x(i) \\ \hline \sum_{i=1}^{N}x(i) & N \\ \hline \end{array}$$

Let's calculate each element with Matlab to show you how simple these formulas are:
To define the 2-by-2 matrix above (called m):

```
>> mydata3=load('Dataset3_1.txt');
>> m(1,1)=0; for i=1:7 m(1,1)=m(1,1)+mydata3(i,1)^2; end;
>> m(1,2)=0; for i=1:7 m(1,2)=m(1,2)+mydata3(i,1); end;
>> m(2,1)=0; for i=1:7 m(2,1)=m(2,1)+mydata3(i,1); end;
>> m(2,2)=0; for i=1:7 m(2,2)=m(2,2)+1; end;
>> m
m =
   91   21
   21    7
```

YOU'RE NOW DONE WITH THE HARD PART!!

Step 2- This matrix must be inverted.

Now what does it mean to invert a matrix? The inverse of matrix **M** (called $\mathbf{M}^{-1}$) has the property such that $\mathbf{M}\cdot\mathbf{M}^{-1} = \mathbf{1}$. Not quite the number 1, but a matrix with the same number of elements as M, each diagonal element being the number 1 (others are 0). So in this example:

a	b
c	d

The inverse is (see http://mathworld.wolfram.com/MatrixInverse.html for more info):

$$\begin{vmatrix} \left(\frac{1}{ad-bc}\right)\cdot d & -\left(\frac{1}{ad-bc}\right)\cdot b \\ -\left(\frac{1}{ad-bc}\right)\cdot c & \left(\frac{1}{ad-bc}\right)\cdot a \end{vmatrix}$$

Multiply the matrix with its inverse using the rules of matrix algebra and you get:

$$\begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix}$$

Here is the great thing- **don't worry about any of this**- Matlab does everything for you!

```
>> m_inv=inv(m)
m_inv =
   0.0357   -0.1071
  -0.1071    0.4643
```

Let's double check it:

```
>> m_inv*m
ans =
    1    0
    0    1
```

See, once you set up the **M** matrix and its inverse, you don't have to do anything else!

YAY! YAY MATLAB!!

Step 3- Lets first talk about the data we are fitting: we will define N as the number of data points to fit, and lets call the actual data we wish to fit y(i). In our case, there are 2 variables to the function we are fitting the data to (m and b), so let's define p as this quantity (p=2). For step three, we have to know something about the deviation of the data from the fit; that obviously must play a part in the errors of m and b. To do this part, define s_y^2 as:

$$S_y^2 = \frac{\sum_{i=1}^{N}(y(i) - fit(i))^2}{N - p}$$

Almost done! Define the best fit from fminsearch for the line (which are the same parameters off my Excel spreadsheet oddly enough):

```
>> for i=1:7 fit(i)=2.060714*mydata3(i,1)+1.975; end;
```

Next step:

```
>> sy2=0;
>> p=7-2;
>> for i=1:7 sy2=sy2+((mydata3(i,2)-fit(i))^2)/p; end;
>>sy2
sy2 =
   0.1748

>> varcovar=m_inv*sy2
varcovar =
   0.0062   -0.0187
  -0.0187    0.0812
```

This is it! The Variance-Covariance Matrix!!

Actually, this is kinda like learning that the ultimate answer to the ultimate question in the universe is the number 42. You have to fully understand the question to truly get the answer.

The variance-covariance is actually equal to:

σ_m^2	$\sigma_m \times \sigma_b$
$\sigma_b \times \sigma_m$	σ_b^2

So if you want to know the error of the slope you type:

```
>> sqrt(varcovar(1,1))
ans =
```

0.0790

Likewise for the intercept:

```
>>sqrt(varcovar(2,2))
ans =
```

0.2849

Rememer the result from the Excel spreadsheet? The linear least squares result was:

slope = 2.0607 ± **0.0790**

intercept: 1.975 ± **0.285**.

Looks like the variance-covariance matrix works!

Last bit, let's look back at the variance-covariance matrix:

```
>> varcovar
varcovar =
   0.0062   -0.0187
  -0.0187    0.0812
```

How did we know that the square root of the (1,1) element (upper left-handed) is the error in the slope? Easy, that was defined when you took the derivative of the equation for a line with respect to the slope as: $\frac{\partial f(m,b)}{\partial m}$, likewise for the intercept.

Now what do the off-diagonal elements $\sigma_m \times \sigma_b$ and $\sigma_b \times \sigma_m$ mean? First, the off-diagonal elements are the "covariances." The covariance elements tell you that if your calculated slope is too low, then your calculated intercept it too high (as in this case the covariances are negative.). If they are positive, then an underestimate in the slope means that your intercept is also underestimated (i.e. they are "going-together"). A large off-diagonal element means that there is a lot of "cross-talk" between the m and b variables. Ideally, the covariances are 0, meaning that if you made a bad fit to the intercept, it did nothing to your estimate of the slope. Thus, the variables are independent. This is unfortunately rarely the case with real data.

Here is the most important part- starting from the beginning, who the heck sez you have to work with linear fits?

YOU NOW CAN CALCULATE THE ERROR OF ANY FUNCTION YOU CHOOSE AS THE BEST TO FIT YOUR DATA

3.2 More advanced fitting, and exponential decays

An exponential decay is described by the equation:

$$f(t) = A \cdot e^{-k \cdot t}$$

where A is the amplitude, t is time, and k is the decay constant. Let's figure out how to make a variance-covariance matrix from this equation. Noting that we are only fitting two variables, and starting from the beginning. Here we are going to add to the fact that the matrix elements should be weighted by the values of the errors of the individual data points, if they are known:

M =

$\sum_{i=1}^{N} \left(\frac{1}{\sigma(i)}\right)^2 \left(\frac{\partial f(A,k)}{\partial A}\right)^2$	$\sum_{i=1}^{N} \left(\frac{1}{\sigma(i)}\right)^2 \left(\frac{\partial f(A,k)}{\partial A} \cdot \frac{\partial f(A,k)}{\partial k}\right)$
$\sum_{i=1}^{N} \left(\frac{1}{\sigma(i)}\right)^2 \left(\frac{\partial f(A,k)}{\partial k} \cdot \frac{\partial f(A,k)}{\partial A}\right)$	$\sum_{i=1}^{N} \left(\frac{1}{\sigma(i)}\right)^2 \left(\frac{\partial f(A,k)}{\partial k}\right)^2$

Now let's fill in the individual pieces:

$$\frac{\partial f(A,k)}{\partial A} = e^{-k\cdot t}$$

And:

$$\frac{\partial f(A,k)}{\partial k} = -A\cdot e^{-k\cdot t}\cdot t$$

Now let's plug this into the matrix, and we get

M=

$\sum_{i=1}^{N}\left(\frac{1}{\sigma(i)}\right)^2\left(e^{-k\cdot t}\right)^2$	$\sum_{i=1}^{N}\left(\frac{1}{\sigma(i)}\right)^2\left(e^{-k\cdot t}\cdot -A\cdot e^{-k\cdot t}\cdot t\right)$
$\sum_{i=1}^{N}\left(\frac{1}{\sigma(i)}\right)^2\left(-A\cdot e^{-k\cdot t}\cdot t\cdot e^{-k\cdot t}\right)$	$\sum_{i=1}^{N}\left(\frac{1}{\sigma(i)}\right)^2\left(-A\cdot e^{-k\cdot t}\cdot t\right)^2$

This can be simplified as:

$\sum_{i=1}^{N}\left(\frac{1}{\sigma(i)}\right)^2 e^{-2\cdot k\cdot t}$	$\sum_{i=1}^{N}\left(\frac{1}{\sigma(i)}\right)^2\left(-A\cdot t\cdot e^{-2\cdot k\cdot t}\right)$
$\sum_{i=1}^{N}\left(\frac{1}{\sigma(i)}\right)^2\left(-A\cdot t\cdot e^{-2\cdot k\cdot t}\right)$	$\sum_{i=1}^{N}\left(\frac{1}{\sigma(i)}\right)^2 A^2\cdot t^2\cdot e^{-2\cdot k\cdot t}$

Now try to apply the above to Dataset3_2.txt, if you recall that from the last module where you fit an exponential fit to some data as shown here:

When you did the fitting to a function:

$$a(1)\cdot e^{-t/a(2)}$$

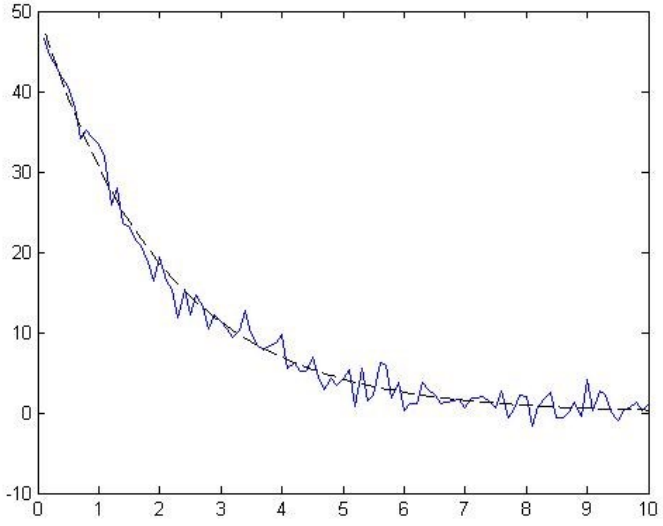
you found an amplitude of a(1)= 50.2501 and a time constant a(2)= 2.0113. If you do the above error analysis, you should find:

Amplitude: 50 ± 3

Time constant: 2.01 ± 0.18

Note that we are using proper error reporting values in the last step.

Included in this packet is "chapter3_cheatcode.m", a script which will do this analysis.



Matlab Assignment

1. Exponential Decay. Use the file "Problem3.txt" enclosed in this packet. It contains 3 by 500 points: the first column represents time in minutes, the second Liters of Cat, and third are the error bars. It is roughly exponential, which is a function of the form:

$$f(t) = A \cdot e^{-k \cdot t}$$

where A is the amplitude and *k* is the rate or decay constant. Here is a good way to make an exponential decay in Matlab-ese:

```
for i=1:500
fit(i)=x(1)*exp(-x(2)*Problem3(i,1));
end;
```

The amplitude is x(1) and the decay constant is x(2). We often call $\tau = 1/x(2)$, where τ (tau) is called the time constant.

Use fminsearch to calculate the best fit amplitude and decay constant. Don't forget you have to make a guess at the amplitude (it's about 10 Liters of Cat) and decay constant (it's about 1 minute)! Here is a shorter way to calculate χ^2 in your function:

```
chisq=0;
for i=1:500
   chisq=chisq+(1/Problem3(i,3)^2)*(Problem3(i,2)-x(1)*exp(-x(2)*Problem3(i,1)))^2;
end;
```

2. Next, calculate the error of these variables using the variance-covariance matrix. First, calculate the following matrix, which we will call m(1:2,1:2) :

$\sum_{i=1}^{N} \left(\frac{1}{\sigma(i)}\right)^2 e^{-2 \cdot k \cdot t}$	$\sum_{i=1}^{N} \left(\frac{1}{\sigma(i)}\right)^2 \left(-A \cdot t \cdot e^{-2 \cdot k \cdot t}\right)$
$\sum_{i=1}^{N} \left(\frac{1}{\sigma(i)}\right)^2 \left(-A \cdot t \cdot e^{-2 \cdot k \cdot t}\right)$	$\sum_{i=1}^{N} \left(\frac{1}{\sigma(i)}\right)^2 A^2 \cdot t^2 \cdot e^{-2 \cdot k \cdot t}$

As an example, the m(2,2) (i.e. the lower right hand one) is shown here:

```
m(2,2)=0;
for i=1:500
   m(2,2)=m(2,2)+(1/Problem3(i,3)^2)*(x(1)*Problem3(i,1)*exp(-x(2)*Problem3(i,1)))^2;
end;
```

Remember x(1) and x(2) are your optimized results from #1, not the guess!

Once you have this and the other three elements, invert the matrix:

```
>>m_inv=inv(m);
```

Calculate s_y^2 and then multiply the inverted matrix by it and multiply by it:

```
>>varcovarm=m_inv*sy2;
```

Remember that varcovar(1,1) isn't the variance in the amplitude, but the square of the variance in the amplitide!

Some hints: Write the matrix calculator in a script so that you will be easily able to make edits to it. Also, remember that variables in scripts are "seen" by the consul, which is not true of functions.

ps if you forgot how to calculate s_y^2:

```
sy2=0;
for i=1:500
sy2=sy2+((problem3(i,2)-fit(i))^2)/p;
end;
```

Remember that p is the number of points minus the number of fitted parameters, and if you get stuck, then you can look at the scripts that were included in this packet.

Answer:

The file “Ch3_problem_answer” provides the following:

$A = 10.2026 \pm 0.0988$ $k = 0.9741 \pm 0.2444$

Chapter 4. The BOOTSTRAP

4.1. Normal Errors

The definition of error of a fitted variable from the variance-covariance method relies on one assumption- that the source of the error is such that the "noise" measured has a normal distribution. In case you don't remember what a normal distribution is, it's a bell-shaped curve (or Gaussian). In case you are not sure what it means for noise to have a normal distribution, take the following set of data:

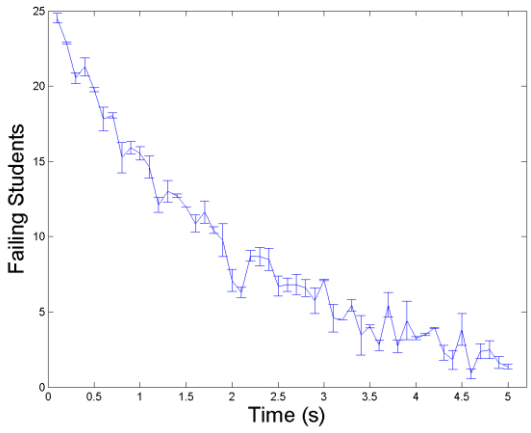


Figure 1

Now, I make a histogram of the magnitude of the errorbars:

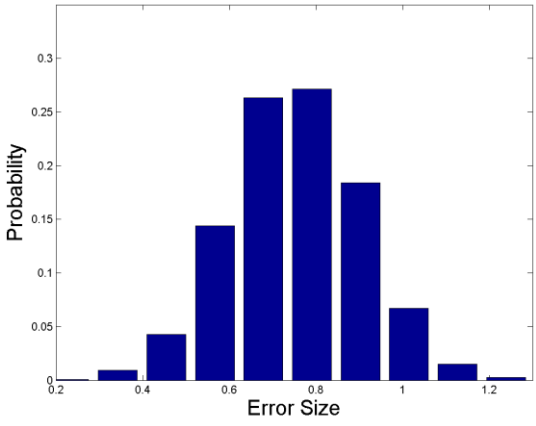


Figure 2

See- you can tell the error distribution is bell-shaped! That means that there is a decreasing probability to get a point with an error that is very far away from the average error, and there is no error bar as small as 0 (i.e. there is always a little bit of error). Generally, this makes a lot of sense as our word for "outliers" implies something that is away from the norm.

So when you're fitting data with normally distributed errors, you can analyze the errors to the fits with the variance-covariance matrix method which I am still waiting for most of you to do.

4.2 Abnormal Errors

Now let's look at this exponential decay curve:

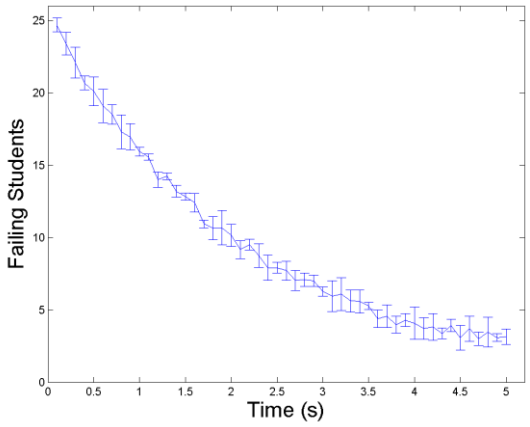


Figure 3

Here is a histogram of the errorbars:

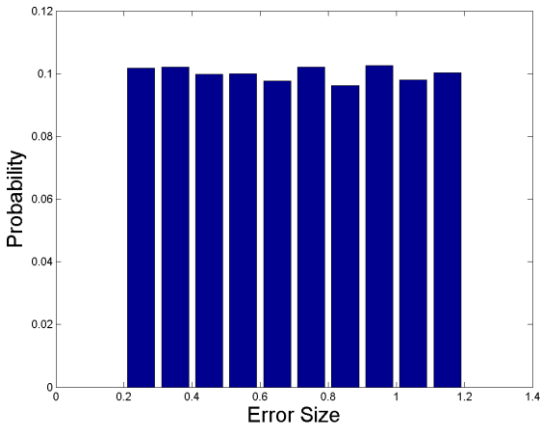


Figure 4

While the magnitude of the error bars span a similar range as before (0.2→1.2 failing students), this does ***not*** look like a bell-shaped curve but a flat line. In this case, the variance-covariance method will **FAIL**.

The solution is the Bootstrap Method. In this procedure, you will first fit your data as best you possibly can by minimizing χ^2. This gives you the best fit but you don't know how wide the variances to the parameters of that fit are. In the next step, you take that best fit, add noise that conforms to your distribution of noise, and you re-fit this new set of data. This gives you an all-new set of parameters to your best fit. Now you repeat this process several times giving you a very large dataset of parameters to work with. The variance in the parameters can then be estimated by the standard deviation of all those fitted parameters.

Step 1. Fit the data by minimizing χ^2. In this example case, this is an exponential decay: I calculate an amplitude of 25.3581 failing students and a rate constant of 0.4579 s^{-1} (or τ = 2.1839 s) using a script I call fitter4; the data is stored in three columns in the file 'Dataset4_1.txt':

```
function [return_val]=fitter4(x)
err=0;
mydata4=load('Dataset4_1.txt');
for i=1:50
 err=err+(1/mydata4(i,3)^2)*(mydata4(i,2)-x(1)*exp(-mydata4(i,1)*x(2)))^2;
end;
return_val=err;
```

In the consul, I typed:

```
>> mydata4=load('Dataset4_1.txt');
>> x(1)=25; x(2)=1;
>> fminsearch('fitter4',x)
ans =
  25.3581    0.4579
>> x2=ans;
```

In case you forgot, the point of the above is to find the parameters for the fit that minimize χ^2, which is:

$$\chi^2 = \sum_{i=1}^{N} \left(\frac{1}{\sigma(i)^2}\right) [\text{data}(i) - \text{fit}(i)]^2$$

Its fairly frequent to see the effect of the individual errors associated with each point to not be included, in which case $\chi^2 = \sum_{i=1}^{N} [\text{data}(i) - \text{fit}(i)]^2$. Most analytical instruments report data without the associated σ, which is why this is done.

Step 2. Make a fit from these best parameters:

```
>> for i=1:50 fit(i)=x2(1)*exp(-mydata4(i,1)*x2(2)); end;
```

Here is a plot; the data are in blue, the fit in red; it looks like a very good fit.

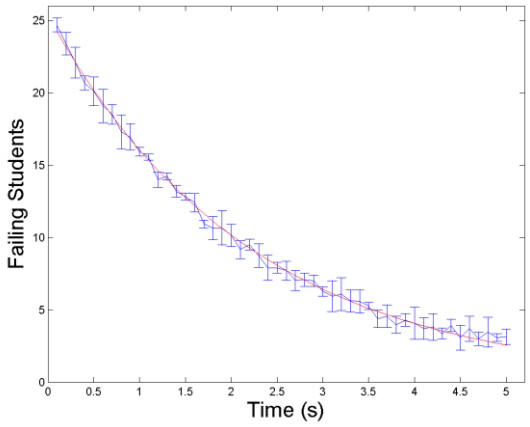


Figure 5

Step 3. Add noise to this fit from the known distribution of error and treat this as a new set of data. This is called the **Monte-Carlo** method. Re-calculate the best fit parameters to this "fake" data.

Now this is where some of you might get flustered, but it is actually quite simple. In the example above, the error is centered at ~0.7 and varies fairly evenly from 0.2 → 1.2. This means that the data point can either be in error no less than 0.2 failing students or no more than 1.2 failing students, and that it is equally probably that the error is somewhere within this range. Now I know that Matlab can make evenly (i.e. flat) distributed random numbers from 0→1 using the **rand** command.
To explore this, do the following:

>> for i=1:10000 catinhat(i)=rand; end;	This is just a series of 10000 random numbers.
>> hist(catinhat);	This is how you make histograms.
>> h_legend=xlabel('Magnitude')	These commands allow you to make the
>> set(h_legend,'FontSize',20);	x-axis and y-axis labels bigger. I'm just
>> h_legend=ylabel('Number of occurrences');	showing you how for the heck of it.
>> set(h_legend,'FontSize',20);	
>> axis([-.5 1.5 0 1200]);	

To aid your understanding, just cut and paste the line below into matlab (it's the same commands):

```
for i=1:10000
catinhat(i)=rand;
end; hist(catinhat);
h_legend=xlabel('Error Size');
set(h_legend,'FontSize',20);
h_legend=ylabel('Number of occurrences');
set(h_legend,'FontSize',20);
axis([-.5 1.5 0 1200]);
```

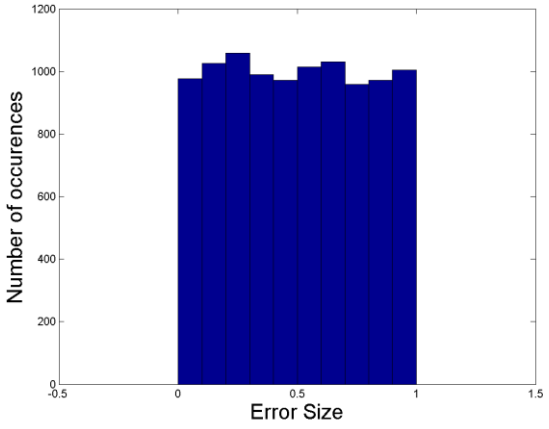


Figure 6

Remember that histograms are like probability distributions; in this case I see that it generates random numbers evenly from 0 → 1. To make random numbers from 0.2 → 1.2, I just add 0.2 from rand:

```
>>for i=1:10000 catinhat(i)=rand+0.2; end;
>>hist(catinhat);
```

To save time, cut and paste the following into matlab:

```
for i=1:10000
catinhat(i)=rand+.2;
end;
hist(catinhat);
h_legend=xlabel('Error Size');
set(h_legend,'FontSize',20);
h_legend=ylabel('Number of occurrences');
set(h_legend,'FontSize',20);
axis([0 1.4 0 1200])
```

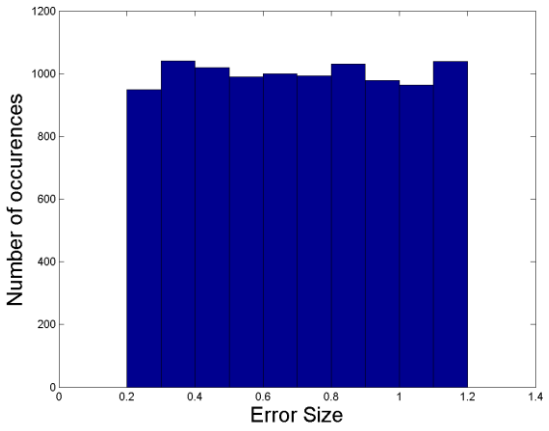


Figure 7

Note that the random numbers spread from 0.2→ 1.2, which is just like the error distribution in the data that I am analyzing; it's very similar to Fig. 4. We are almost done.

To make our first Monte-Carlo set of data, we now take the fit and add the "flat" random numbers from 0.2 to 1.2. **NOTE! In reality, errors can be positive or negative!** To account for this, we multiply the magnitude of the error by sign(randn), a function that randomly makes the error positive or negative:

```
>> for i=1:50 montefit(i)=fit(i)+(rand+0.2)*sign(randn); end;
```
This adds noise to the fit

```
>> plot(mydata4(:,1),montefit);
```
Plots the "fake data"

```
>> hold on;
>> plot(mydata4(:,1),mydata4(:,2),'r');
```
Overlays "fake" and real data

Here is the same for you to cut and paste into Matlab:

```
for i=1:50
montefit(i)=fit(i)+(rand+0.2)*sign(randn);
end;
plot(mydata4(:,1),montefit);
hold on;
plot(mydata4(:,1),mydata4(:,2),'r');
h_legend=xlabel('Time (s)');
set(h_legend,'FontSize',20);
h_legend=ylabel('Failing Students');
```

set(h_legend,'FontSize',20);

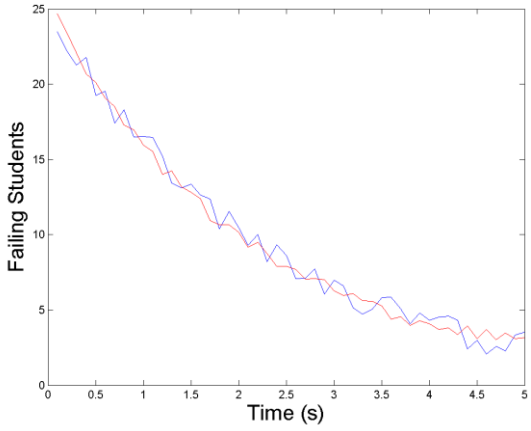


Figure 8

Notice that you can't really tell which is the "real" set of data, and which is your Monte-Carlo data!

Now when I fit this set of Monte-Carlo data, I first saved it as follows:

>> save -ascii montefit.txt montefit;

And I rewrote my fitting function and saved it as fitter4_2.m:

```
function [return_val]=fitter4_2(x)
err=0;
mydata4=load('Dataset4_1.txt');
montefit=load('montefit.txt');
for i=1:50
   err=err+(1/mydata4(i,3)^2)*(montefit(i)-x(1)*exp(-mydata4(i,1)*x(2)))^2;
end;
return_val=err;
```

Now run it:

>> fminsearch('fitter4_2',x2)
ans =
  25.7957 0.4662

Remember that x2 is your previous best fit parameters to the real set of data you fit earlier

I get A= 25.7957 failing students and rate k=0.4662 s^{-1}.
Now if I do this one more time:

>> for i=1:50 montefit(i)=fit(i)+(rand+0.2)*sign(randn); end;
>> save -ascii montefit.txt montefit;
>> fminsearch('fitter2',x2)
ans =
  25.6746 0.4620

I get A=25.6746 failing students and rate of k=0.4620 s^{-1}. Now if I generate another set of "fake" data and refit, I get A=25.6226 failing students and rate of k=0.4629 s^{-1}. Now you see I have a statistically significant set of data; I can calculate the error of the fitted amplitude by the following:

```
>> amps=[25.7957, 25.6746, 25.6226];
>> std(amps)
ans =
   0.0888
```

The result is that the best fit to the amplitude of the data is A=25.3581 ± 0.0888 failing students, or if you **would actually like to properly report the result**, its A=25.36 ± 0.09 failing students. The same analysis of the three Montecarlo rates yields k=0.4579 ± 0.0022 s^{-1} (or k=0.458 ± 0.002 s^{-1}). **Note that I am reporting the value from the fit to the "real" data, whereas I get the errors from the standard deviation from the "fake" data.** Also note that you do not report the standard deviation of the mean, just the standard deviation.

Here is an interpretation of what just happened- given that your error is random, you were just as likely to have taken the data that you "simulated" with the Monte-Carlo method as the data that you actually took (see Fig. 8). Thus, the Monte-Carlo fits you just made are as valid as the fit to the real data. In that case, the standard deviation of those fits is a meaningful statistical description of the errors of your fits.

Now while the above example works just fine, you should realize that you probably should make more than just three sets of Monte-Carlo data. The point of DAS BOOTSTRAP is that you can make hundreds or thousands (I typically make ~10,000 to 1 Billion) of such simulations. Thus, while you're fitting fake data which is normally kinda bad, you can fit so much fake data that the results are actually very meaningful. In our example above, let's make 100 Monte-Carlo fits- here is a set of examples so that it isn't so hard. Start with making the Monte-Carlo data:

```
>>for j=1:100 for i=1:50 montefit2(i,j)= fit(i)+(rand+0.2)*sign(randn); end; end;
```

You can actually plot several of them (let's do the first 3) at once to see what is happening:

```
>> plot(mydata4(:,1),montefit2(:,1:3))
```

Now you already have a function that calculates χ^2, we are just going to write a series of commands to get it to do that over and over again:

```
for j=1:100
montefit=montefit2(:,j);
save -ascii montefit.txt montefit;
params(:,j)=fminsearch('fitter4_2',x);
end;
```

(hint, write the whole thing to a script so that you can edit it more easily for your own homework: It's no different than writing the exact same thing in the consul, save it as filename.m and type >>filename <enter> in the consul).

Now you might be waiting for a few minutes…

DONE!

See, this might be the simplest exercise to date. Now the errors in my amplitude are calculated by taking the standard deviation of all the amplitude fits that are in the first row of the variable params:

```
>> std(params(1,:))
ans =
   0.5391
```

which gives an amplitude of A=25.4 ± 0.5 failing students; likewise, the rate is k=0.458 ± 0.012 s^{-1}. Note that the result is somewhat different when you ran just three Monte-Carlo simulations; this shows you that you need to perform a lot of these simulations. To check, I performed 1000 simulations, and I get the same results as when I ran just 100 simulations.

Matlab Assignment

1. The Bootstrap. The file "Problem4.txt" is part of this packet. It contains 3 by 500 data points: the first column represents time in seconds, the second Turkey Giblets, and the third is the σ of Turkey Giblets. It is roughly exponential, which is a function of the form:

$$f(t) = A \cdot e^{-k \cdot t}$$

where A is the amplitude and k is the rate or decay constant.

First, use fminsearch to calculate the best fit amplitude and decay constant. In this case, be sure to weigh the contribution to χ^2 by σ^2 of each data point. Don't forget you have to make a guess at the amplitude (it's about 100 Turkey Giblets) and decay constant (it's about 0.01 s^{-1})! Next, make a fit using the best amplitude and decay constant you calculate from fminsearch.

Now you have to use the fit to create fake data. First, histogram the σ of the data and determine what the distribution of the σ is, i.e.

```
>>hist (Problem4(:,3)); <enter>
```

I will only give it to you that it is a "flat" distribution like the example I provided in class and in the handout. Note that it isn't exactly like the example in class…

Next, create 1000 random sets of data and analyze each one with fminsearch. **Then report to me the amplitude and decay constants from the best fit to the real data, as well as the errors of the same as determined by the bootstrap method** (that's the standard deviation of the 1000 fits to the fake data, see my example handout).

-Note you **don't report the average amplitude or decay rate from the "fake data"** although those should be very close to the amplitude and decay rate you calculate from minimizing χ^2 of the real data using fminsearch. This is a good way to check that your programs are working properly.

-Note that as you are using a random number generator in this process, **absolutely none of your errors will be exactly the same** as anyone else's.

Answer:

A: 99.0392 ± 2.4079 k: 0.0101 ± 5.8837e-04

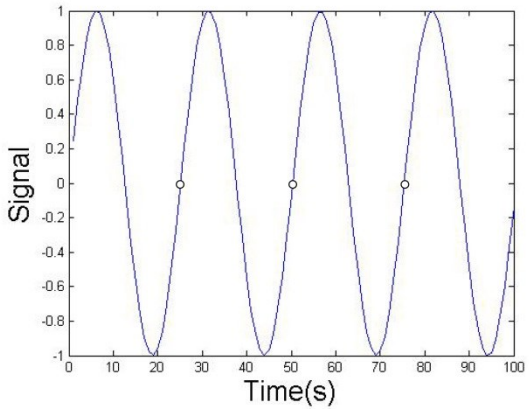
Note that your result will vary due to the Monte Carlo nature of the analysis.

Chapter 5. Fourier Transforms and Signal Analysis

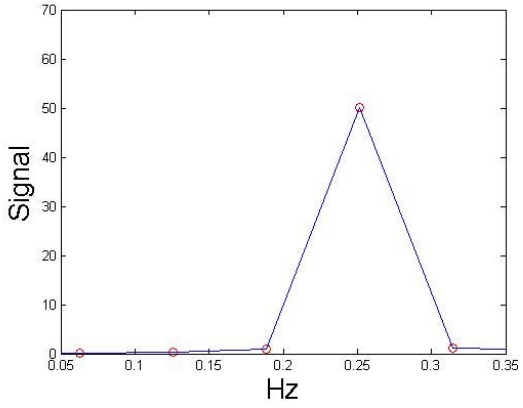
5.1 Frequencies in signals

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:

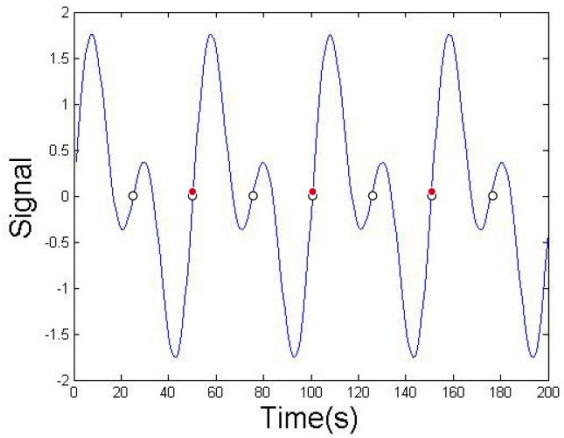


Now understanding the nature of the x-axis of a Fourier transform can be tricky. First, note that the FT spectrum has an x-axis that is the inverse of the original signal, which is seconds in this example. Thus, the FT spectrum has an x-axis of Hertz, or s^{-1}. Next, a feature at 0.25 Hz corresponds 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:

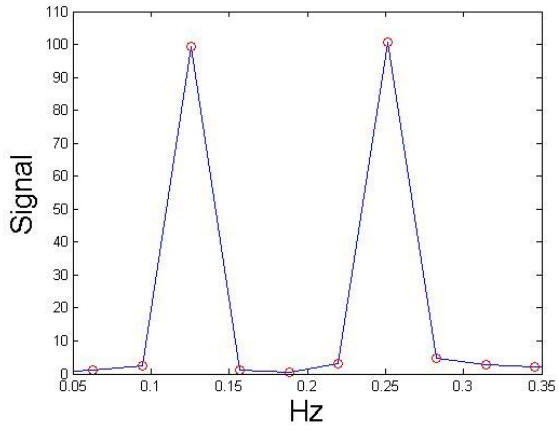
```
for i=1:100
signal(i)=sin(i*1/4);
end;
```

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:

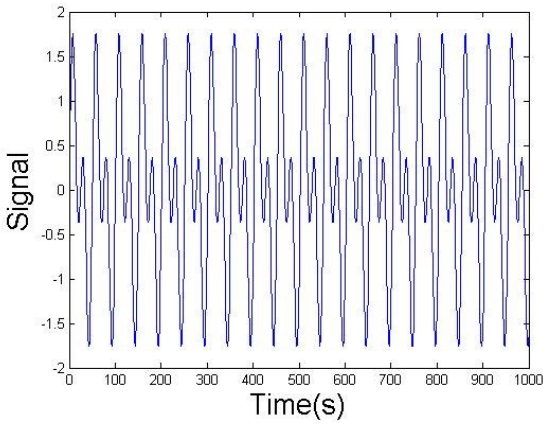
```
for i=1:100
signal(i)=sin(i/4)+sin(i/8);
end;
```

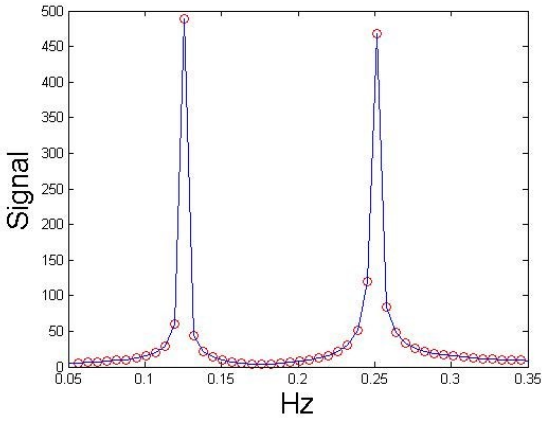


A beating pattern of two waves overlapping is clear; repeated patterns are observed every ~25 seconds (circles) and ~50 seconds (red dots). Fourier transformation of these data yield:

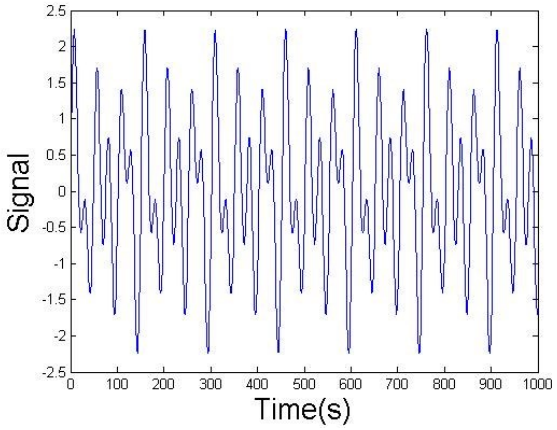


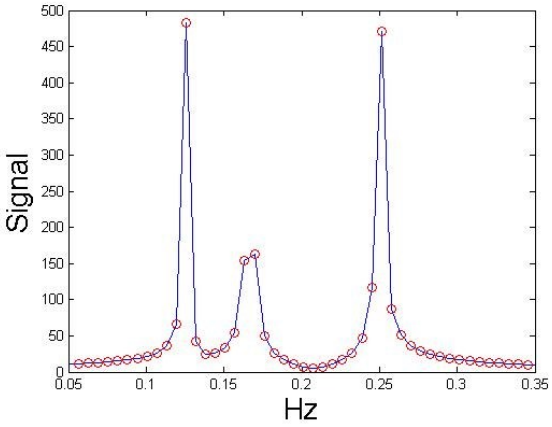
A new feature appears at a smaller frequency 0.125 s^{-1} (=1/8 s^{-1}), and note that (1/0.125 s^{-1}) × 2π = 50.2 s. This is in fact what I added to the signal. Note that I had increased the number of data points, measuring the signal out to 200 s. Let's see what happens when we go to 1000 seconds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

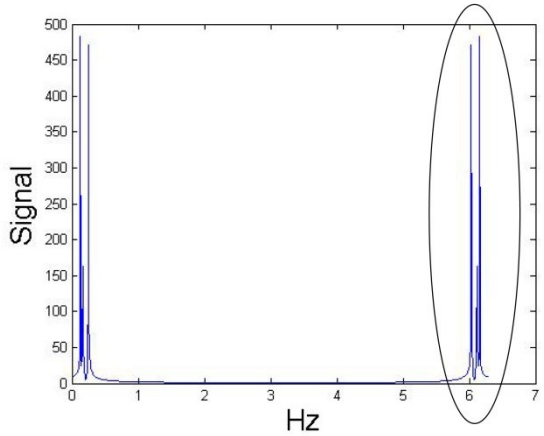
Note that we have real and imaginary components; the proper way to deal with this is to calculate them separately as above. Sometimes, depending on what kind of data you are taking Fourier transforms of, the real and imaginary components may mean something different. However, this is uncommon and as 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)

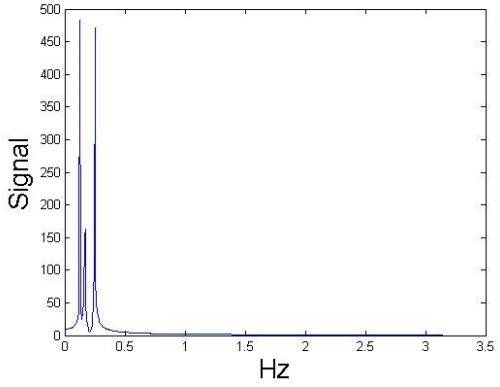
Here is what you see:

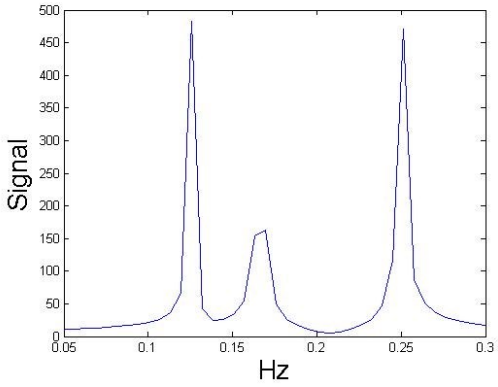


While we expect the peaks on the left side (0.1→0.3 Hz), as seen above, the right handed high frequency peaks (circled) may be confusing. These are actually mirror images of the real peaks from 0.1→0.3 Hz. They form because the real data on the left side are calculated with "right" travelling waves while 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

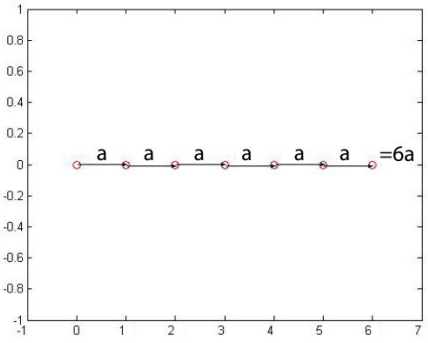
One of the biggest mistakes a person makes with discrete Fourier transforms is that they do not know what are the allowed frequencies are. As stated above, the allowed frequencies are:

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

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:

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

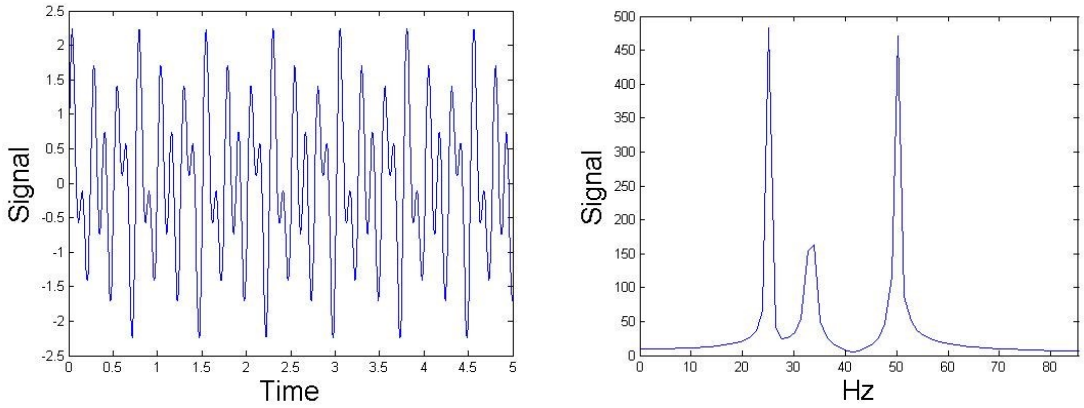
First, note that (N-1)·a is actually the total length of the data set. Imagine a data set of **7** points each set "a" amount apart as shown below:



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):

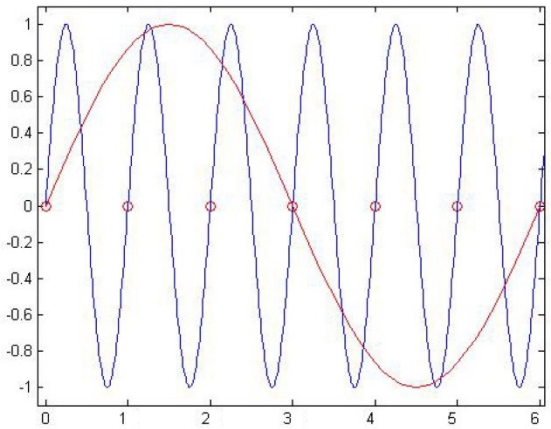


The frequencies have increased by a factor of 200, or 1000/5, which is correct.

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.



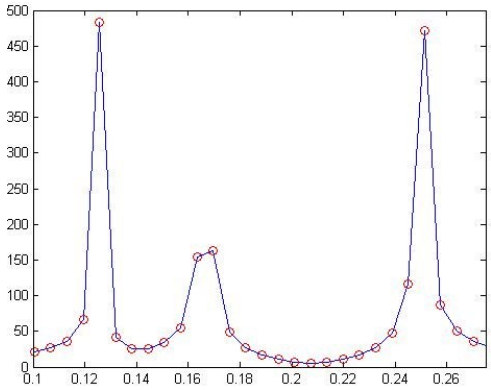
Why do the frequencies work this way? It's because there is a fundamental limit on the frequencies that can be explored. The highest frequency has a wavelength equal to the spacing between points as shown here in blue. While 0 is the lowest allowed frequency, the next lowest is the length of the data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

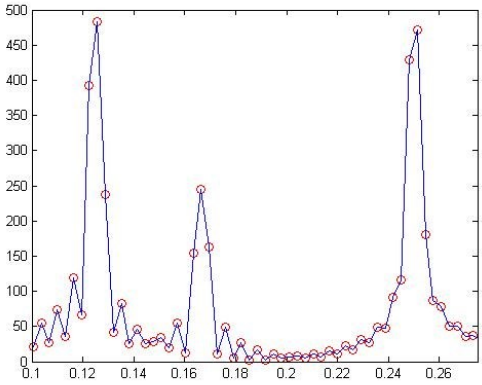
$0, \frac{\pi}{\text{length}}, \frac{2\pi}{\text{length}}, \frac{3\pi}{\text{length}}, \frac{4\pi}{\text{length}}, \ldots$

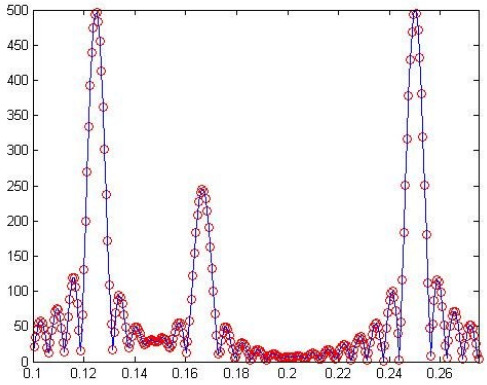
where $\frac{\pi}{\text{length}}, \frac{3\pi}{\text{length}}$, etc. are not normally allowed? Wouldn't that add data that would make the peaks narrower?

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:

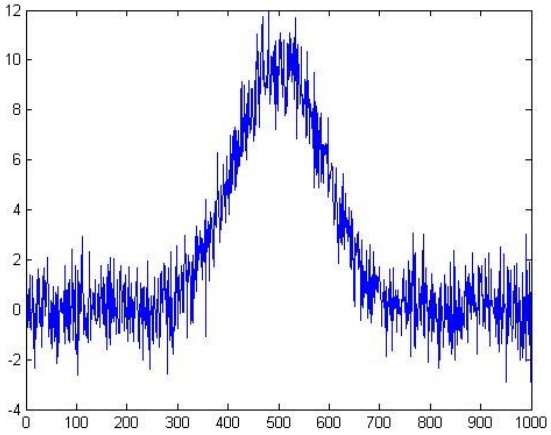
```
length=1000;
i=sqrt(-1);
for j=1:length
    back(j)=0;
    for a=1:length
        a2=2*pi*(a-1)/length;
        back(j)=back(j)+1/length*forward(a)*exp(i*a2*j);
    end;
end;
```

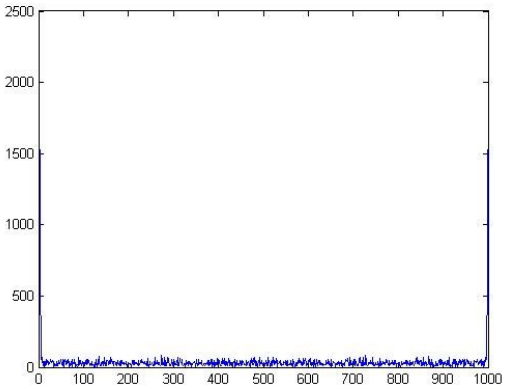
So, you can go forward and backward all day. So what? There is a method for removing noise from a spectrum called optimal windowing that uses forward and backward transforms; here is how it works. Let's make a noisy signal:

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

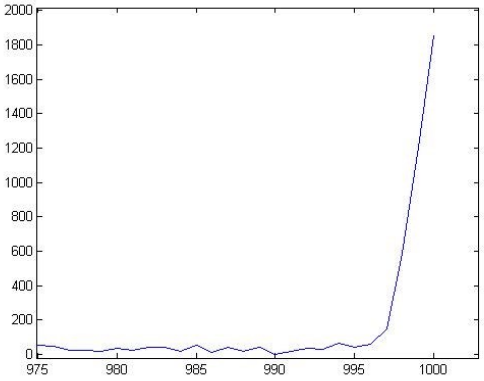
And now make the Fourier transform and plot it:

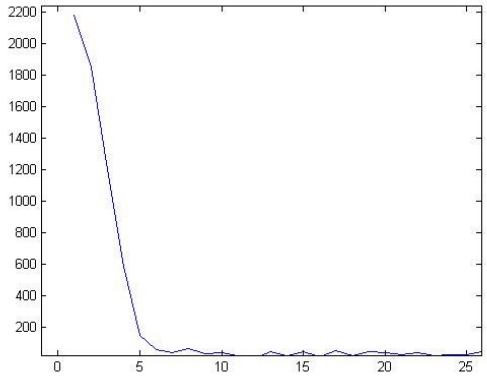
```
plot(abs(forward));
```





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

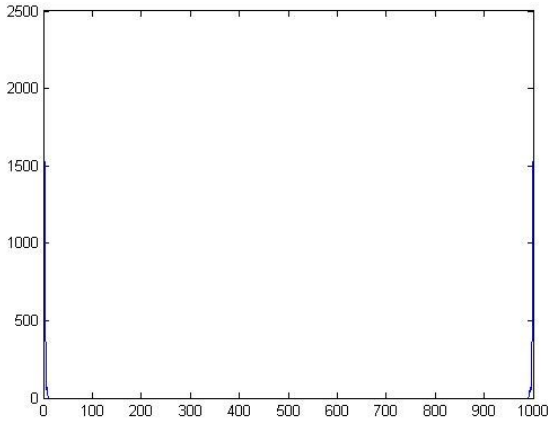




What you are seeing is that there is a significant signal at the lower frequencies (0→5 and 995→1000), while the higher frequencies (6→995) appear like noise. This is ironic- look closely at the noisy spectrum; imagine what kind of frequency signals would make that noise. Are they high frequencies? You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

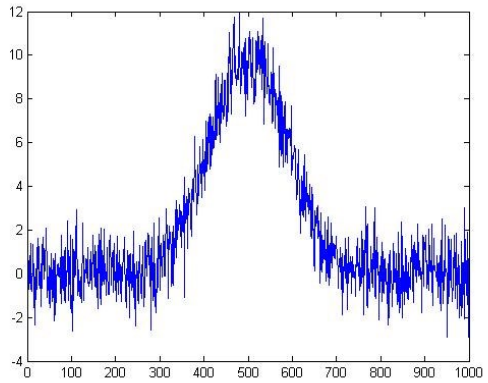
```
for i=5:995
forward(i)=0;
end;
```

Here is the new, cleaned Fourier spectrum:

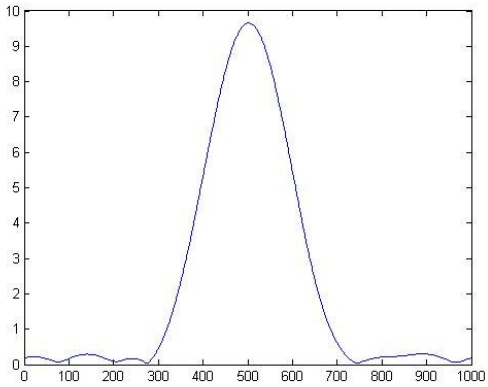


Now when I do the backward transform I get this (the old on is on the left for comparison):

Old



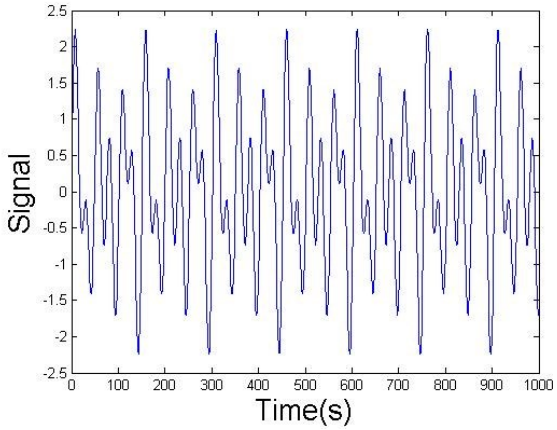
New

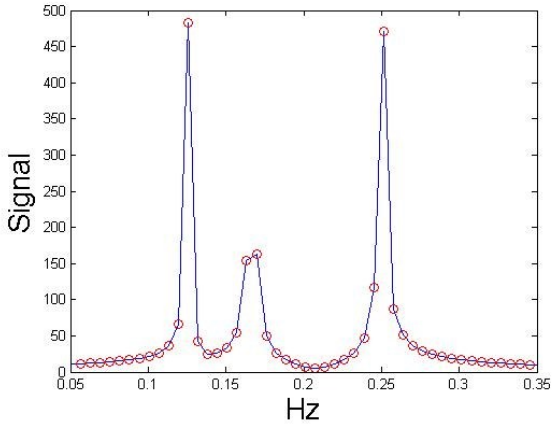


Quite a trick, isn't it?

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.

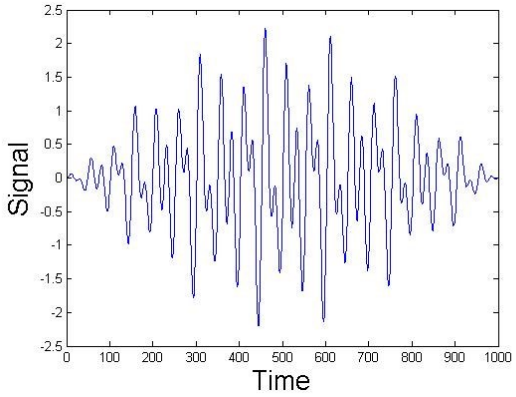


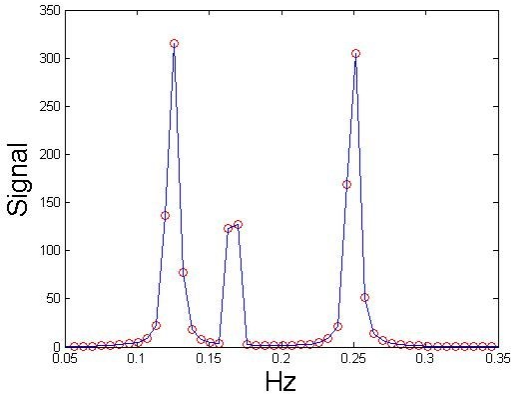


Note that the Fourier transform signal on the left has a somewhat uneven baseline. This is a result of the way the transform calculates data on the left and right sides of the total signal; there is some "badness" as the data simply come to an abrupt end. As such, you can smooth the baseline by multiplying 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:

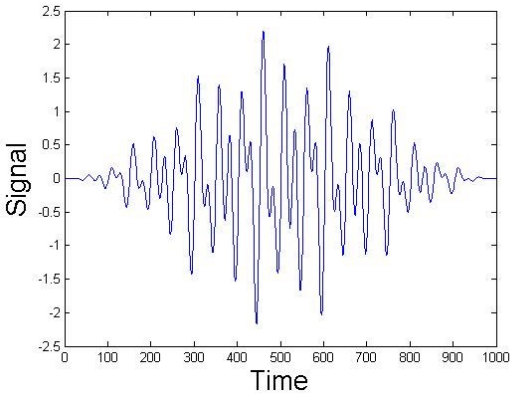




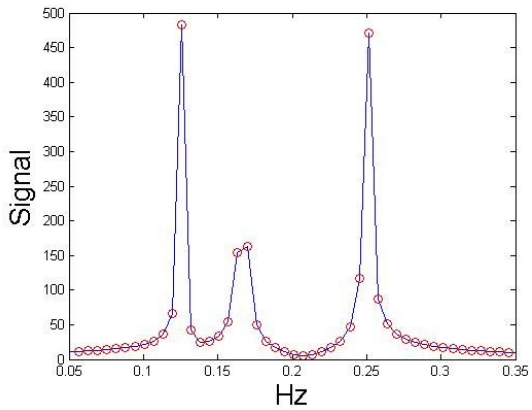
See how more smooth and flat the baseline is now? Especially for the middle peak. This is called a cosine power window. Over the years, mathematicians have developed a large number of window functions, see:
http://en.wikipedia.org/wiki/Window_function
I will provide one more example here, the Hanning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

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

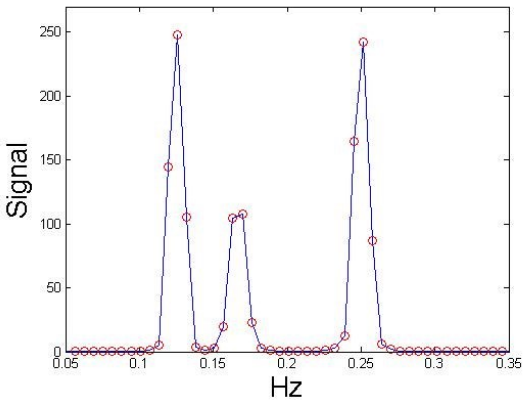
Which make the data look like this (unfiltered transform is on the left for comparison):



Old



Windowed



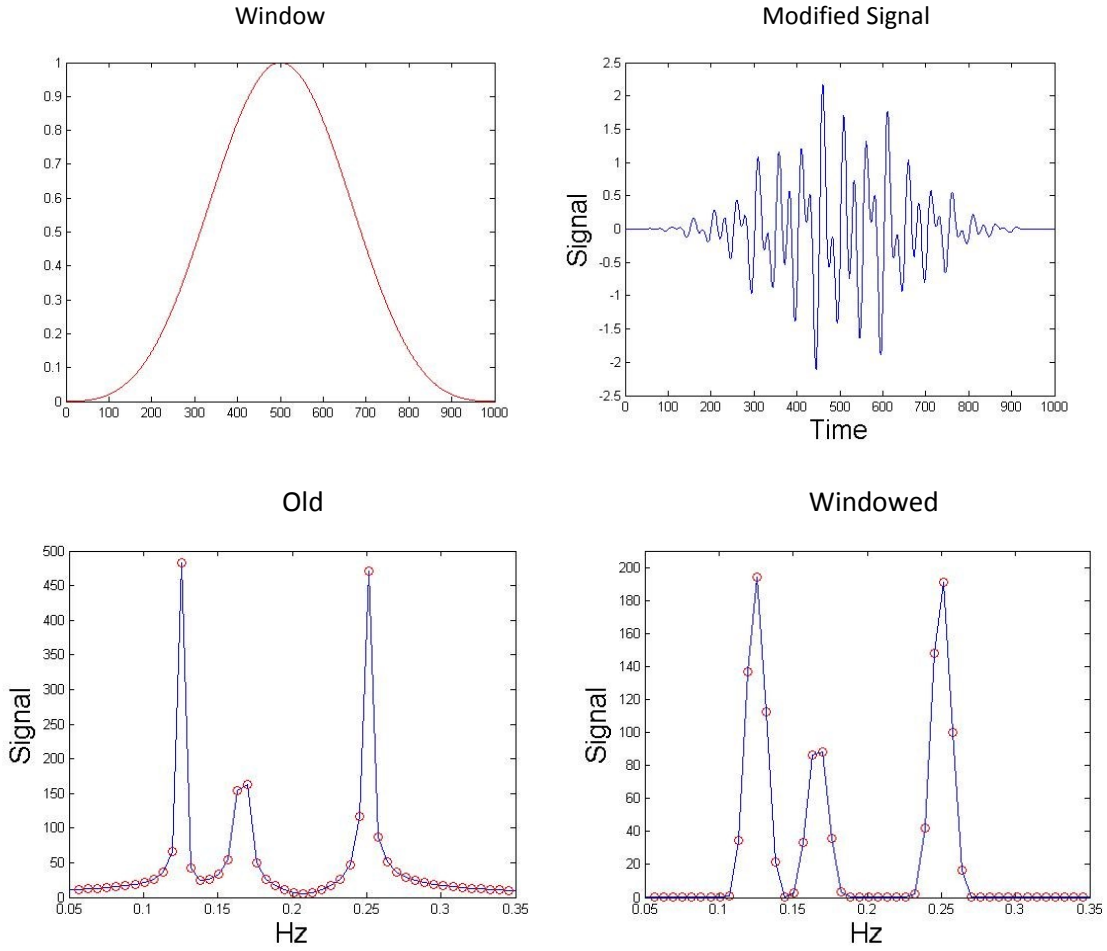
Almost all Fourier transform instruments window their data. One last example of a very common window is the Kaiser-Bessel function:

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

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;
```

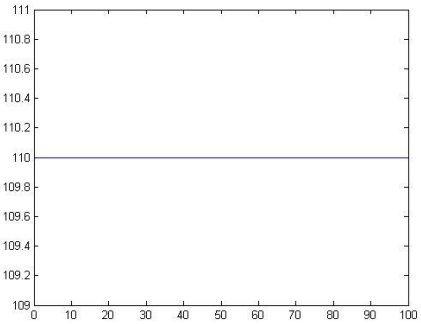
The results are:



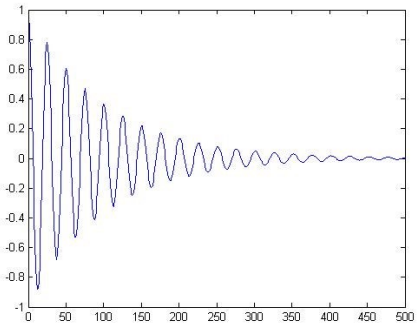
You can see that this window is very popular due to its ability to completely remove a baseline. Note that the overall signal does decrease, a drawback to this method.

Matlab Assignment

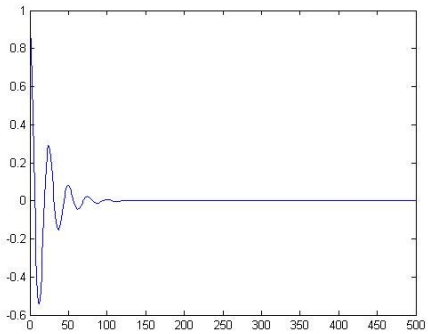
1. a. Fourier Transforms. Calculate the absolute value of the Fourier transform for the Problem5_1.txt data set shown below.



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

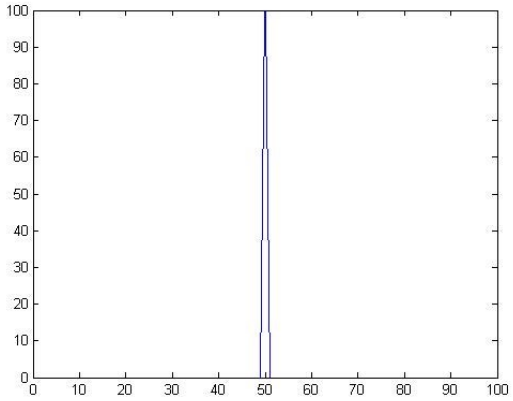


c. Calculate the absolute value of the transform for the Problem5_3.txt data set as well as the characteristic frequency. Compare and contrast the results from pt. b.

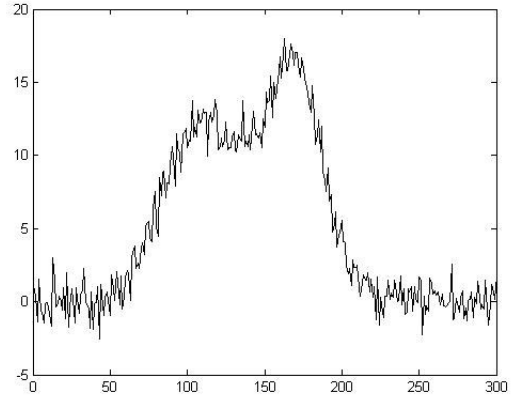


d. Apply a sine window (or any window) to the data in pt. c (Problem5_3.txt).

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

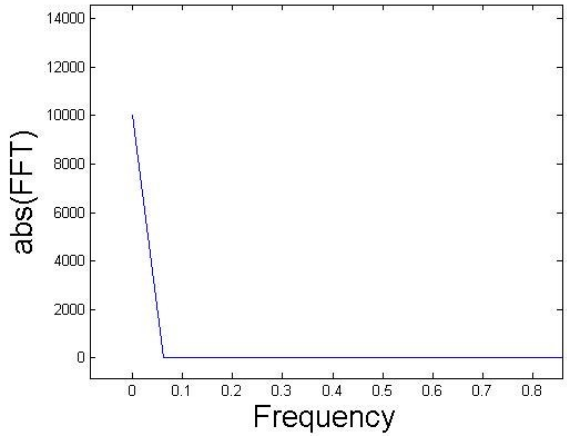


f. Use the optimal filter technique to "clean" up the following data in Problem5_5.txt.

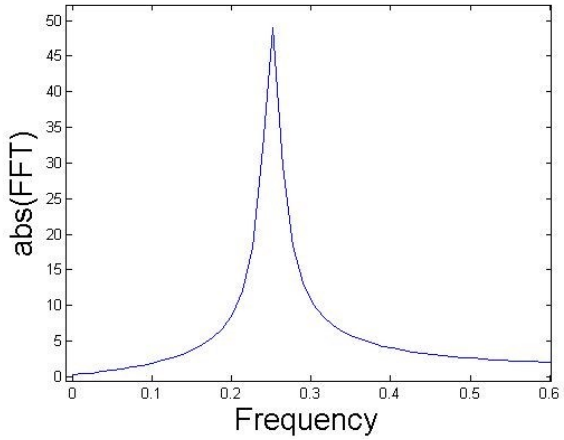


Answers:

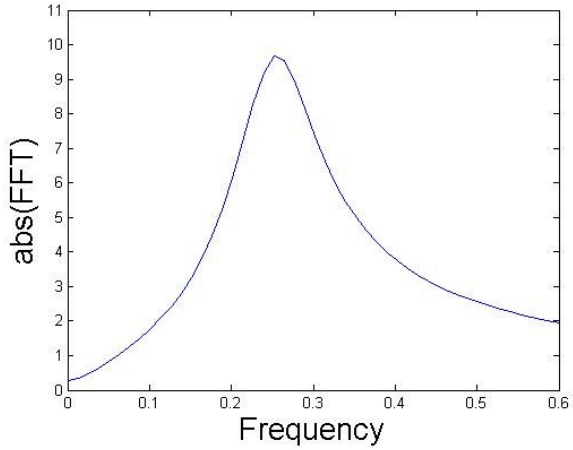
a. As there is no real signal, the FFT is singularly peaked at 0 frequency:



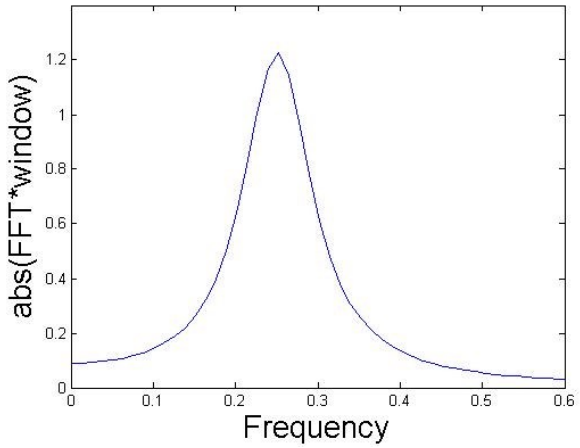
b. The frequency is 0.252 Hz.



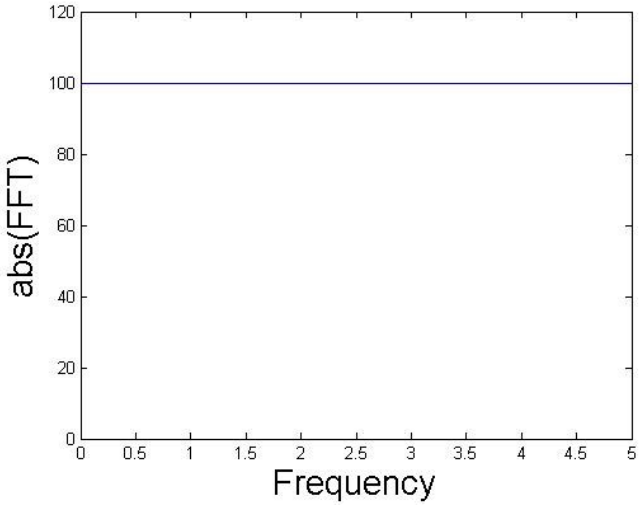
c. Basically the same, but the fact that the signal is shorter broadens the signal.



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: