Week 1: Tutorials for running Octave.

Ultimately you will need to turn in a script file (which you can break up into parts if you want) that shows that you have worked through this tutorial. The file(s) should contain each of these exercises, so that anyone could simply run the script file and see the output plots, etc. that are generated. Be sure to document your code; use comments to provide short answers to the questions below. Remember to create separate figures for each exercise. (See "help figure" to see how to do this.) I will not specify each figure number below; it is up to you to be sure that you create different figures for each exercise. You may want to generate a script file for each part of the assignment, because they will be reused in later assignments. (A cut-and-paste approach might work to generate a single file to upload for grading, but be sure you don't overwrite needed plots.)

Adding your initials to the title would help. For example, I would use BHR_tut1_A.m

For additional help explore the Octave_Getting_Started and the Control_Plots documents that are posted with this tutorial.

Part A: Enter Data, Plot Data and Fit Data to a Straight Line.

Here we will summarize what you need to do that will be applicable to the experiments you will do in this course. You need to be able to enter data, and get a plot of the data. You'll then model the data and plot the model over the data. We will go through these steps here. (We will come back to this later to discuss in more detail what goes into data and error analysis; but that is for next week.).

Start Octave, go to your work directory of choice, add the Utility Subroutine to your path (the *addpath* command). Verify you can access utilities (Asking for help on a particular subroutine will test the path).

A.1: How to enter data and make a plot of the data.

Here (to get a head start on problem 4 of your statistics Work Sheet) is some example data. Concentration of Copper and the resultant Battery Voltage.

[Cu(I)], 10^{-4} M	E vs. SCE, volts
2.22	-0.32372
4.06	-0.30828
6.15	-0.28717
9.84	-0.28271
23.2	-0.26148
38.4	-0.23570
61.2	-0.23237
84.9	-0.22904
108.0	-0.22115
158.0	-0.20882

We will start by creating a script (*.m file) for our processing. We will be entering the data into variables, so think about how to best structure the data. There are many ways to do this; nothing is right or wrong, but some structures will be easier to work with, as you'll learn. We need to choose proper names for the Cu concentration and the Battery Voltages. Generally you want something descriptive but not too long (typing gets tiring). And you want to put the numbers in with some efficiency (that is, think about how to cut/paste from here to the Octave editor.) It is nice to line up the X and Y data so you know your data pairs stay together. The general form of what you type is CuConc=[x1 x2 x3 ..] and Volt = [y1 y2 y3 ….]. If you entered these as row vectors, but like columns of data, then X=X' etc. If you like to bundle your data together then Data_Set = [X Y] or Data_Set = [X; Y] depending on whether your data is column or row data. Try these different ways to group your data.

A.2 Plot your data:

plot(X, Y, 'k*') is a typical way to plot {X,Y} data. Just use markers (*) for each data pair. 'k' means 'black'. Use *help plot* for details on what markers or colors are available. Also, look at the document on controlling plots (called Control_Plots).

Plot your raw data, as figure(1) and then in figure(2) plot the data so that X= log(Cu_Conc)

Always label axes (include units) and put a title on each figure: (below are examples)

```
xlabel('log[Concentration]')
ylabel('Voltage [V]')
title('Data from Problem 4, Statistics_Work Sheet')
```

A.3 Plot the Model Fit to the data

Now fit your data: [Params, S_Fit] = polyfit(X, Volt, 1)

Take a moment look at "help polyfit" just to understand input and output variables.

and plot the model fit to the data

Y_fit = S_Fit.yf;

plot(X, Volt, 'g*', X, Y_fit ,'k-')

If you have time also try *EZLstSq* in the Utility Subroutines. It is almost a direct replacement for *polyfit* but has a few added features.

Part B: Statistical Evaluation of Random Numbers

B.1: Look at random numbers and do statistical analysis:

Create N (e.g. N=100, or N=1000, etc) random numbers and plot them as a function of their counter:

N = 1000;

Rn = rand(N,1); (later: for Normal Distributed random numbers Rn = randn(N,1);)

To visualize you can use:

```
figure(3);   %r Remember to start new figures for new plots
plot(Rn,’r.’)   % This plots each data point as a red dot.
```

Note that usually we plot x vs y, but here we have only the y values. This statement is equivalent to having x=1:N and then using plot(x, Rn, 'r.')

Do they look random?

Determine the mean, variance and standard error. These are the basic items one needs from any data analysis.

The help on rand says it gives uniform random number from 0 to 1. Uniform means uniformly distributed randomly-generated numbers. Based on this information, what mean and standard error would you expect and why? How close do you come to what you expect?

The needed routines are *mean*, *var* and *std* (which you probably guessed, along with *min*, *max*, *range* etc. See the first 10 pages of Chapter 7 of Hansen’s book). To get the standard variance of a set of numbers, X, you need VarX = var(X,1). Use help var in Octave to understand why the 1 is attached. The same rule applies to std as well. What is the difference between std(x,0) and std(x,1)? Which one do you need?

Make a histogram (use: “help hist” to see how the *hist* routine works) and see how that looks. If you used *rand* or *randn* you will get different results. Compare with a neighbor who used the other routine. Try different numbers of bins and see how the results look different.

Is the mean within the expected standard error of the mean?

To make uniform numbers symmetric about zero try:

Rn = 2*rand(N,1)-1 (N is predefined by you earlier).

Rn will then be your set of N random numbers uniformly distributed from -1 to 1. (Subtracting 1 will make the values have a mean near 0.)

Show a histogram of this and estimate the mean, and the standard error and variance with calculus and compare with the numerical results you get.

What would you expect the mean to be? What would you qualitatively expect the standard error to be?

For a randomly distributed but uniform distribution, $P(x)$ from -1 to 1: This means that P is a constant over the interval -1 to 1 but it is zero otherwise. Use the general idea of statistics and distributions to determine the exact value to your intuition about what these quantities ought to be. From statistics:

$$\langle f \rangle = \frac{\int f(x)P(x)dx}{\int P(x)dx}$$

Using calculus, solve for the quantities $\langle f \rangle = \langle x \rangle$ and $\langle f \rangle = \sigma^2 = (x - \langle x \rangle)^2$, that is the expected mean and variance.

B.2: Create a 100 by 100 matrix (or array) of random numbers. Yes, 10,000 random numbers in all.

```
N=100;
Rn =  2*rand(N,N)-1  ;
```

To visualize these random numbers do

hist(Rn(:,1)   => you are seeing all values just in column 1

hist(Rn)   => you are seeing each column separately, but within each of the 10 bins

hist(Rn(:)) => you are seeing all values treated as a single vector in 10 bins

Note that you are now making a histogram of a matrix, not a vector.

Look at the differences and explain what you see. Explain what you are seeing, notice the numbers on the axes (they help to verify what you think).

B.3: Now we can do a lot with these numbers.

Suppose we average all 100 numbers in a column. What should the distribution look like and what should the standard error be? By having 100 rows when we average all the columns we will still have 100 random numbers to do statistics on.

So try this: (see "*help mean*" to understand the arguments to the function mean.)

```
Rn1 = mean(Rn,1);
Rn2 = mean(Rn,2);
```

What did we do? Look in the workspace to understand the row/column nature of the results.

What does "mmr=mean(mean(Rn))" give and why? Again is the mean (i.e. mmr) within the error you expect from the standard error of the mean? We are testing equation 3 of our summary (See the ErrorAnalysis_Summary document).

If you have not heard of the central limit theorem, you are in for a surprise when you look at the histograms of Rn1 and Rn2: ***Compute the mean and standard deviation of these two 100 element vectors and comment on their values. In particular, why is the standard error so small?***

To get practice with some helper routines (see Hansen's book, pages 73ff): ***Of the 100 points in Rn1, how many are between 0.0 and 0.025? Use the find command to find the data points greater than 0.0 and less than 0.025.***

```
idx = find(Rn1 < 0.025 & Rn1 > 0.0)     %returns a vector of element positions or indices ("idx")
```

```
Rn1(idx)    %shows you the actual values found in this range
```

Using a vector (idx) of indices now allows you to see each Rn1 value at each position. Remember that to see one element you use Rn1(3) for the third element. This shorthand allows you to see many elements at once.

Part C: Getting Data into analyzable forms

Much of our work will be to bring in data from text files. So here are some routines that will help you load your data into Octave.

The built-in routines are *load* and *importdata* (as one word). (See the section in chapter 4 about loading ASCII data in Hansen's book.) I have written several routines as well that are particularly adapted to the data styles we get from the experiments in the lab. The Utility Subroutines directory contains them. In particular: Some data comes from an Ocean Optics Spectrometer so there is an OO get file, and some comes from a standard terminal emulator called Putty (used in Exps 3 and 4) and there is a get Putty type file:

get_Text_Data.m, get_OO_Data.m and get_Putty_Data.m

You will need to let Octave know about these utility routines. The best way to do this is to add the path to the Utility directory to Octave's search path. You can do this with this command:

```
addpath("Utility Subroutines", 1)
```

This tells Octave to check this directory (in addition to any other work directory) when a function is called. Depending on where the Utility Subroutines directory is the *addpath* arguments may be slightly altered.

As long as you stay in your directory, Octave will know about these routines. Chapter 1 of Hansen's book talks about the *path* and *addpath* commands a bit, and the help is useful too.

One of the test data files that we can use here as an example file comes from Experiment 4 (and applies to both experiments 3 and 4). The data has four columns. The time is in column 2 and the temperature in column 4. You will be plotting the temperature as a function of the time.

There is something peculiar going on with the time values (for the abscissa): Putty outputs time in standard military time (HHMMSS). The data points were collected every 10 seconds, so the data ought to be uniformly spaced; and they are if you read the numbers correctly. ***However, let's take a look at the time values by plotting the time data as a function of the index of the time values. What do you see and what could we do about this?***

C.1:

File "Exp 4 parr.txt" is available and you can take a look at it by right clicking on it (and selecting Open) in Octave's File Browser window to open it in the editor. Octave will show the text in the editor so you can see the structure. There are 4 columns of data. Column 2 is the time (X) axis, and

Column 4 is the temperature (Y) axis. The file editor in Octave is like any other text editor, like notepad etc. You can also use XL to look at the data as well and that interpreter helps too.

Try the *load* command:

```
  load(“Exp 4 parr.txt”)
```

This will fail if the first line of the dataset is not correctly formatted data, but is instead text (a comment.)

Now, using the editor, put a % or a # in the first line and try the *load* command again. You can also load data by double clicking on the file (in Octave), or by right clicking on the file (in Octave) and choosing "Load".

This time the load should work, and the data will have the same name as the text file without the extension.

OPTIONAL: Also try *importdata*:

```
        A=importdata ("Exp 4 parr.txt", ",", 1);
```

Note that this brings the data into a scalar structure (1x1) called A. The data is then accessed by A.data. You can use the data with that structure, or you can set a new variable, DS (dataset), equal to it: DS = A.data; The command *importdata* can handle a wider variety of datatypes, although this should not be necessary here.

Alternately, you can run get_Text_Data routine (in Utility Subroutines). This also brings the data in correctly. One advantage of this routine is that it lets you browse for the data file of interest (you don’t need the name), just do not enter any arguments on the input side of the routine.

Enter the command   data = get_Text_Data

Then at the prompting browse for the data file “Exp 4 parr.txt”. Note that this utility routine does basic data checking and so will ignore line 1 whether or not you have put in the "%" or "#".

A plot of time (column 2) as a function of its indices is performed by:

```
time = data(:,2);    % this grabs all values (:) in column 2
plot(time)
```

Notice the irregular jumps in the data. ***Look at the numbers and explain what happens? How do we understand this or work around it?***

There are several strategies, we can discuss them. A most simple approach is to redefine the time assuming it really is in 10 second increments:

time = 10*(1:length(time));   ***% Explain what this does and why we can do it.***

In general, such an assumption may be unwarranted and one really should interrogate the time vector to be sure each time point is indeed separated by 10 seconds. To actually extract the true time from this time, one needs the *mod* function. For example,

    t2_sec = mod(time,100) will give you the seconds.

t2_min = (mod(time,10000)-t2_sec)/100; gives you the minutes.

t2_hr = (time –(100*t2_min + t2_sec))/10000; gives you the hours

Then, putting it all back together the true time in seconds is

trut = t2_hr*3600 + t2_min*60 + t2_sec;

Now plot(trut) to verify that indeed they are all separated by 10 seconds.

Another way to do it is to look at diff(trut). *diff* gives the difference between each value and its neighbor and so all values of diff(trut) should be 10. A related test is that the standard error of diff(trut) should be zero. If it is not zero, then they are not all the same.

Another way to look at the time number is to treat them as text strings so we could manipulate each character. The routines *num2str* and *str2num* let you do this. (Remember that Octave is a strongly typed language; numbers and characters are treated quite differently.) For example

strtime = num2str(time(1))

shows an example of what happens to the first time value. strtime(1:2) will be 15, but it is a string. Therefore   t_hr1 = str2num(strtime(1:2)); will give 15 as a number (an integer) now. Using this approach the true time can be reconstructed.

Just as a caution, note that if you are running an experiment elsewhere that uses a similar time base, and it runs past midnight you will find the hours will flip from 23 to 00 at midnight. If this happens you will need to process your data further to adjust for this. (Hint: use "diff" to see if this happens, and add 24 to your hours values.)

Using a corrected or regularly spaced time, plot the data and see what trends are there. Plot data with icons (dots) unless the data is spaced too closely; then you are better off just connecting the dots.

Y= data(:,4);

plot(trut,Y,'k.')

IMPORTANT: In general, one does not connect individual data points; this would assume that you actually know what the data does between the data points. But one does plots the model (or theory) with a solid line, since this is continuous over all values.

C.2:

As further practice, look at the data in Exp1 (the Ocean Optics kinetic experiment).   The data file is “KBP240nm26degrees.txt“. Look at this data in the editor. We will be using just columns 2 and 3. As you will discover, loading data from files can sometimes be tricky. It is good to understand this since it will help you figure out how to save data in the future in ways that can be properly interpreted.

Try the *load* command. After seeing the data format in the editor can you understand why it failed?

Try the *importdata* command. But, look at how the column 1 data was interpreted; and, columns 2 and 3 aren't even parsed.

Now try *dlmread* command. Note that you'll need to specify the delimiter as the tab character.

Data = dlmread("KBP240nm26degrees.txt", "\t"). Look at the size of Data (it will have 2 or 3 colmuns of data, which contains the columns you will see in the text editor).

Look at the data; you'll see that column one is not interpreted properly, but that doesn't matter here.

Try the *get_OO_Data* command (in the Utility Subroutine directory). It works and automatically ignores the first column of data. (The [X Y] data are bundled into a single variable name.)

```
Eg:   ds = get_OO_Data;
t = ds(:,1);  % extracts the time
Abs = ds(:,2);  % Extracts the absorbance
t = (t-t(1))/1000;  % puts the time in seconds relative to time zero at the experiment start.
```

Verify that you can get data in by these two means by plotting the time and absorbance values (columns 2 and 3 of the text file) independently. Then plot them as plot(X,Y).

C.3: The data are basically an exponential decay with other features, principally the early time data are doing nothing and then the exponential form of the absorbance starts in after some initial reference time (we will call t_0, such as about 30 to 50 seconds in, in this example). For the part that is exponential, the data (for times greater than t_0 should follow the equation:

$$y = Abs = A_\infty - \Delta A e^{-r(t-\ t_0)}$$

The constants that will optimize the fit of this model function to the data are called the **parameters** of the fit. In this case the parameters are A_∞ , ΔA and r. You can pretty much read the parameters off the data: t_0 is the smallest time you set to use the fit, perhaps around 20 seconds. A_∞ is the absorbance at infinite time (so it is about 1.5). ΔA (as a positive number) is the absorbance difference from infinite time and the time you pick as t_0; somewhere about 1. The decay rate, r, is a bit trickier, assume r=1/100 (sec^{-1}) for now and then adjust it by eye later. We will develop better ways to optimize all of these parameters in the third tutorial.

Our initial task then is to define to and from there extract a reduced set of times, so that $t > t_o$, and then choose the three parameters and see how well we fit the data over the reduced set of times.

Suppose we want to start the fitting where the absorbance is ~0.7 say. Looking at the data, this would occur at $t_0 = 30\ sec$ (or so). So we need the data and times greater than this place. So the best way is to identify the index of the time axis where t=30. This may appear cumbersome but if you take a moment to examine it you will find a useful strategy here:

```
[ to , indto ]= min( abs(t- 30) );
```

t is the entire t axis, and indto is the index into the t axis, and to = t(indto). The reduced times are then

```
t_red = t(indto:end);
Abs_red = Abs(indto:end);
```

or:
```
ds_red = ds(indto:end,:);
```

To verify that you have obtained the correct set of data points: plot your reduced set of data (in black, say), over the full data set (in red).

```
plot(t,Abs,'r-',t_red,Abs_red,'k-');
```

You are now ready to generate your fit, given the model and the approximate parameters.

Define the parameters:

```
Ainf = 1.5; DA = 1.0; r = 1/100;
Abs_fit = Ainf – DA*exp(- r*( t_red – to) );
hold on
plot(t_red,Abs_fit,'g-')
hold off
```

The hold on and hold off lets you keep the plot as you had it above and overlays this new data.

Alternatively, to get the same figure, you can plot:

```
plot( t,Abs,'r-',t_red,Abs_red,'k-', t_red,Abs_fit,'g-' );
```

Now you may optimize your parameters by hand and do a better fit. How do you know when you have the best fit? We use the least squares criterion:

See Octave help on the routine var (this gives the variance)

```
Var = var(Abs_red-Abs_fit,1);   % The standard error of the fit is sqrt(Var)
```

Why is the 1 in the input argument list?

When you can get Var as small as possible by choosing the parameters then you have done the best you can do. Of course, there is no substitute for looking at the fit, but this criterion is the one our fitting/optimizing programs will use in the next two tutorials.

As with all good graphs, you should label your plot. For example here:

```
xlabel(“time [sec]”)
ylabel("Absorbance")
title( "Exp1 Data Plotted in a Linearized Form" )
```

Show your final plot to your TA.