Setting functions:

In programming, often you would see a list of lines or phrases that somehow execute a certain task. How do they do it? Well all of these lines and phrases execute a certain purpose such as displaying a message, calculate the average of a data set, or any other tasks that can be done by a computer. In Matlab, if you type `display("Hello World")` on the window, the message "Hello World" will be printed below the command window. The combinations of all those phrases will ultimately execute the task that the "program" was asked to do.

In order to execute a certain overall task in a program, you have to break the task into parts such that each different part performs different parts of the program's main task. Sometimes certain tasks are used in a program multiple times but it takes up a lot of space in a program. In this case, it would be simple to write the sub-task separately from the program and then type a command in the main program that executes the sub-task from its separate location. For example, suppose you wanted to write a program that wanted to calculate the average over five data sets. One part of this program would be to find the total sum of the data for each data set. One could simplify the program by writing a separate function that calculates the total sum, in which one would only write the "shortcut" command five times in the main program. In structured programming, we refer to this as creating a *function*, and the command that executes the function from within the main program is called a *function call*.

MATLAB has a call for most elementary functions (if not all). Some examples include the exponential function, the trig functions (sine and cosine), and some special ones such as the sync function. Polynomial functions can be easily defined in MATLAB, almost the same way you would when writing them down on your algebra homework. A list of MATLAB functions can be found if you enter the command: `doc functions` and then go under **Functions-- Aphabetical list**. If you're not sure, whether a function is defined as a MATLAB function, you can always find out online either www.mathworks.com or just Googling it. Commonly used ones are: sin(x), cos(x), tan(x), acos(x), asin(x), atan(x), exp(x), sind(x) (returns the sine value for a value expressed in degrees), sqrt(x), etc.

The syntax for calling any function in MATLAB is: *function_name*(*parameter*)
The parameter is the input required by the function in order to return a result, which in most cases is just a number or a vector of numbers. Usually if you omit the semi-colon after the function call, the result will be displayed on the MATLAB screen.

1) The function call for the sine and cosine function is **sin(x)** and **cos(x)** respectively. Find the cosine values and sine values of 0, pi, pi/2, pi/3. -pi/4 and pi/5. Bonus: Find the inverse sine and cosine values of 1, ½, 1/sqrt(2), -1/sqrt(2), .4
2) The function call for the exponential function is **exp(x)** respectively. Find the cosine values and sine values of 0, pi, pi/2, pi/3. -pi/4 and pi/5.

When handling data sets, and making the function values from that data set, the most common way to get a range of values from a function would be to set your data set as a vector, and then input the vector into the function of your choice. The result of a function with a vector input will return a vector with each element giving the function value of the corresponding element of the input vector. Now try it!

3. Type in command, **x = linspace(0,2*pi, 50);** to create the vector of 50 equally spaced values between 0 and 2pi and calculate the corresponding vector y such that y returns the sine value of each element in x, the cosine value, the sine value and the exponential values of the vector.
4. Plot the results with x as the domain, and the function values attained from x as the range for each case. Hint: To plot any two vectors of the same size and type as domain and range, just use the plot

command

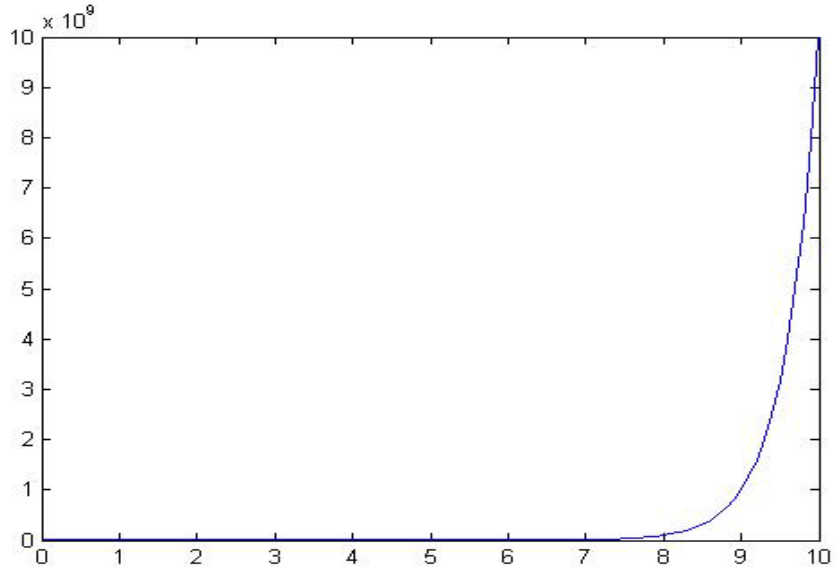
If you wish to use it, in Matlab, one could simplify complicated function expressions by giving them a new “nickname” and then using the nickname in place of the long expression.
ex: I wish to use the function y = sin(x) .+ cos(x) .+ exp(x) multiple times in a code.
Why not create a shorthand for it?. It can be written in the following MATLAB syntax:
functionhandle = @(t) (sin(t) .+ cos(t) .+ t.^2) ;
If I wanted to know the value of this function when t = 3, I use the following syntax:
functionhandle(3)*;*

In Matlab, this is known as the function handle for the above expression. In general for an anonymous function y(x), the function handle can be written as:
fhandle = @(t) (*expression of y in terms of t*);

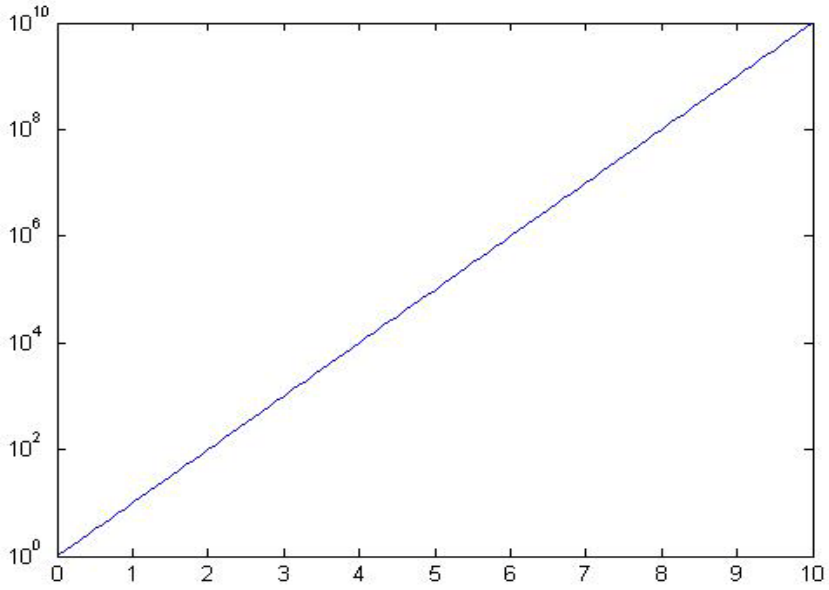
Now,if I wanted to know the value of y at t=3, I express it as: *fhandle*(3);
Similarly this can be done for function of several variables.
Ex 1) Set the function handle for the function y(X) = x^2 + x^3 + 3cos(x);
Ex 2) Evaluate the functions value at x =1,2,45,60 using the function handle
3) Set the function handle for the function $z(x) = \cos^2(x) + \sin^2(x)$.
4) Evaluate the function handle using several inputs and verify that it returns the familiar result for this identity.

Introduction to Logarithmic plots

Some data sets usually use a form of scaling such that on either or both the x and y axis, an tick x represents a^x, (a =10 is a standard use). This is known as **logarithmic scaling**. Logarithmic scaling is generally used a lot in scientific data, because it helps in reducing power laws and exponential functions into linear forms.
For example, if one were fitting data that seemed to follow an exponential pattern, the form of such a curve fit would usually be y = a * b^x, it would be messy to find the parameters of such a function, plus chances are it would look a lot like this:



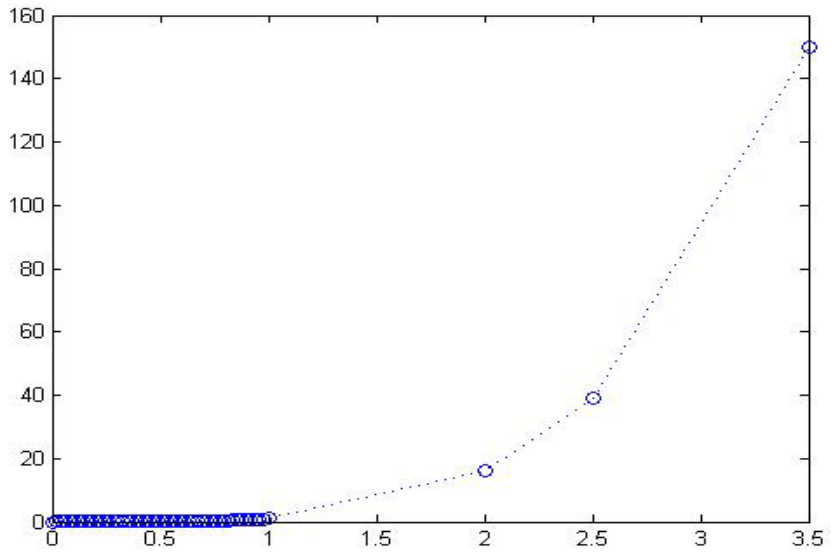
The above picture is a plot of the curve y = 10^x. As you can see, it doesn't look very nice since we can not see most of the data. It would be more useful if we changed the scaling in such a way that more of the data can be seen. Observe, if we took the fit of the form y = a*b^x and took the logarithm of both sides then: log(y) = log(a*b^x) => log(y) = log(a) + log(b^x) => log(y) = log(a) + x*log(b).
So in this case of the curve y = 10^x, if we changed the scale on the y-axis to a logarithmic scale (that is 10^x), the curve would then resemble a a linear curve with slope 1 and intercept 0:



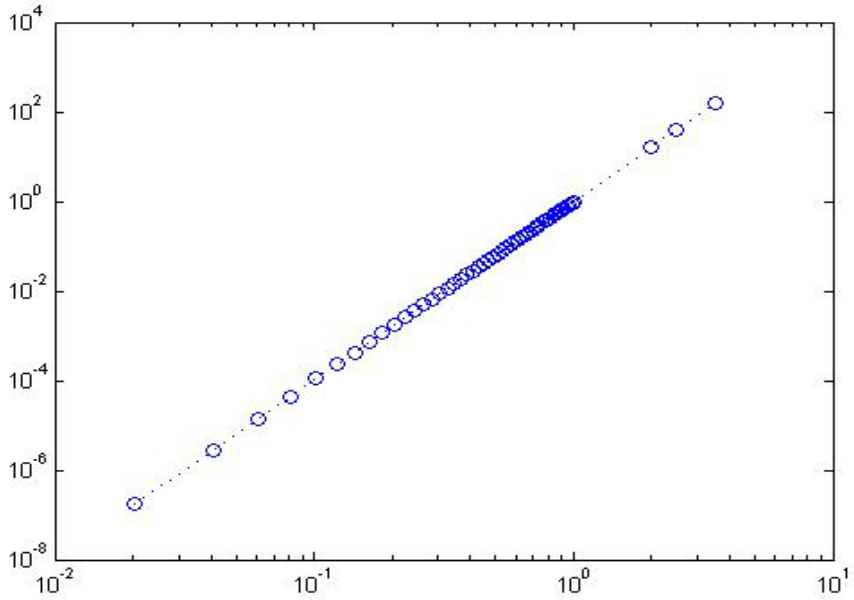
In this a **semi-logarithmic plot** is made along the y-axis. The logarithmic scaling is along the y-axis, where it is most useful in transforming a curve of the form y = a*b^x to a curve of the form, y' = cx + d where y' = log(y).
It is also possible to apply the logarithmic scaling along the x-axis, and is more useful when plotting the logarithmic function y = a*log(x) + b.

In Matlab, it possible to create a semi-logarithmic plot by using either of the functions `semilogx()`, or `semilogy()`. The input of the functions is in the same format as the plot function, where an input data vector x and its range vector y is required, and specifications can be made to change the format of the line used to connect the data points.

In a **log-log plot**, both axes are scaled to a logarithmic scale (usually an increment represents 10^x), and is often applied to power laws. It is the same deal as with the semi-log plots. Suppose you had a set of data that fit to the curve y = x^4. The plot would look a lot like this:



Now we see that a bigger problem exists in that for the data points located below x = 1, we can't see how much it deviates from the curve y =x^4. One obvious way we could see the data more clearly is if we ignored the outliers located at x = 2, 2.5, and 3.5. That way, we can zoom in on the cluster of the data. However, a more efficient way would be to change the x and y scale to a logarithmic scale. Notice, the curve fit of a power law would be of the form y = c*x^n where n is some rational number. With log-log scaling, you would apply the logarithm to both sides and get: log(y) = log(c*x^n) => log(y) = log(c) + log(x^n) => log(y) = log(c) + n*log(x). So the log-log scaling of the curve would take a linear form, y" = b" + n*x" where y" = log(y), x" = log(x). Now take a look at how it looks now:



Notice, now we can have a clearer view of the majority of the data without omitting the data outliers. This will help us a lot once we perform analysis on the data, but for now you can see that applying a logarithmic scale is an efficient way of making your plots look clearer, and one of the best ways to notice certain trends in the data (in this case, the trend is that the non-logarithmic data follows a power relation y = x^4).

In Matlab, it is possible to set a log-log scaling by using the function **loglog(x,y)**. The function works in the same manner as the **plot** function, where **x** is the list of domain values, and **y** is the list of range values.

It is important to mention that in using any of the logarithmic scaling functions, you have to domain/ range values that do not have negative values. Remember, the logarithm of a number cannot be expressed if the number is negative or positive. The logarithm of a number returns the power of the base (which in this case is 10) which gives that number, but no power of any base could ever give a negative number or zero. For example, 10^0 = 1 and 10^(-4) = .0001, but I will never find an x such that 10^x = -32. In Matlab, if your x or y vector has negative values, Matlab will print `Warning: Negative values ignored` and then plot only the positive values.