INTRODUCTION TO MATLAB

How do you plot a sine wave on a computer screen? There are a bunch of ways to numerically calculate sin(t) for a range of time points and then graph them on the computer display. There is the spreadsheet way. Fill a column with a range of numbers from `0` to `6.25` (almost 2*π), incrementing by `0.25` or so, and then in the second column calculate the sine of the value in the first column: entering `sin(a1)` in the cell `b1`, then copying the formula down through the end of the filled column length. This certainly works for simple math operations but not as much for complicated ones and it can be mouse intensive if many similar repetitive graphs are needed. Another way is using code, like C.

```
#include <stdio.h>
#include <math.h>
#define pi = 3.14159

main() {

     float dt = pi/8;
     float t[17], y[17];

     for (i=0; i < 17; i++) {
          t[i] = i*dt;
          y[i] = sin(t[i]);
     }
/*   then you need some graphics tool in C to plot this out to the
     screen,
     or you can output it to a file, import it to a spreadsheet and plot
     that out */

     return 0;
}
```

This is also perfectly acceptable, but the issue of graphic output is at play. There are a lot of graphics libraries to go with C and Fortran compilers and they work well. Still, this strategy is a little cumbersome. Look at all the overhead: `#include` statements, having to declare variables and the lengths of arrays, even the `for` loop is a little goofy. Mathematically, we can write a `for` loop vectorially with one line: **y** = sin(**t**), where **y** and **t** are vectors. This vector representation is similar to a spreadsheet layout. On the spreadsheet, in the first column one could place an array with the time points in it and in the second column the sine of each time point (each row value) would be placed. Thus, the first row of the first column is the first element of this 1 dimensional array or vector (t[1] in C, t(1) in Matlab). In the second column, the first row contains the sine of the first element (first row) of the time vector y(1) = sin(t(1)). Thus, the 7th element of the y vector is equal to the sine of the 7th element of the time vector and so on. If you don't know what this vector stuff means right at the moment, be patient. There will be much more on this later.

y = sin(**t**), where **y** and **t** are vectors. That is a simple and efficient way to mathematically describe the operation that the C code and the spreadsheet example were doing. Why couldn't we do that, that is, represent this vector operation with **y** and **t**, with code, too?

Here's where Matlab comes in. Click on the Matlab icon and up pops a window with a prompt which looks like this >>. This is the command line prompt, and you can directly enter a command into it. For example:

```
>> 2+2

ans =
     4
```

or

```
>> sin(pi)

ans =
     1.2246e-016
```

(very close to zero)

Note that `pi` (π) is already defined in Matlab.

Matlab has 3 advantages which can help us in numerical methods: 1) lots of built-in math functions, 2) easy elementary 2-D and 3-D graphing, and 3) the ability to do vector and matrix operations in a one-line command. The makers of Matlab call these vector and matrix capabilities "vectorization," and Matlab programmers try (some religiously) to "vectorize" their code. More on this later. First, plotting a sine wave:

```
>> t = 0:pi/8:2*pi;
>> y = sin(t);
>> plot(t,y)
```

That was it. It is easy to miss! The first line declares `t` to be an array (vector) with one row and columns filled from `0` to `2*pi` incrementing by `pi/8`. Its a 17 element array, but we don't have to care and declare it as such. The semicolon after the statement is optional. If one doesn't include it, the result of the operation is printed out in the window (just like with the `2 + 2` example).

```
>>t = 0:pi/8:2*pi

t =

  Columns 1 through 7
```

```
     0    0.3927    0.7854    1.1781    1.5708    1.9635    2.3562

  Columns 8 through 14

  2.7489  3.1416    3.5343    3.9270    4.3197    4.7124    5.1051

  Columns 15 through 17

  5.4978  5.8905    6.2832
```

To some, this is not an appealing way to view a vector (1 row, 17 columns). Matlab has a transpose function (′) to help here.

```
>>t = 0:pi/8:2*pi;
>>t = t';
>>t

t =

        0
   0.3927
   0.7854
   1.1781
   1.5708
   1.9635
   2.3562
   2.7489
   3.1416
   3.5343
   3.9270
   4.3197
   4.7124
   5.1051
   5.4978
   5.8905
   6.2832
```

or

```
>> t = (0:pi/8:2*pi)'
```

gives the same answer.

```
>> y = sin(t);
```

makes a vector `y` the same size as `t` and fills the first element of `y` with the sine of the first element of `t` and repeats it for the entire length of the vector `t`. This is vectorized code. Instead of using the `for` loop, one line does it! Even this line is superfluous to Matlab. The following also works:

```
>>t = 0:pi/8:2*pi;
>>plot(t,sin(t))
```

If we wanted to plot multiple graphs on the same plot, the following works:

```
>>% t is already defined
>>plot(t,sin(t),t,sin(t+pi/8),t,sin(t+pi/4),t,sin(t+3*pi/8),t,cos(t))
```

(`%` begins a comment line where matlab ignores all characters after it on the line)

The color of the line changes with each plot (yellow, magenta, cyan, red, green). Notice that we can add a scalar to each element of the vector `t` (within the parentheses of the `sin` argument). Also, plot needs an x for every y plotted. This means we repeat `t` every time. This is clumsy in this example, but when different x vectors are needed for different y vectors (for example, if they need different time increments), this becomes an advantage. To get around this, the following can be done:

```
>> t = (0:pi/8:2*pi)';
>> z = [sin(t) cos(t)]

z =

         0    1.0000
    0.3827    0.9239
    0.7071    0.7071
    0.9239    0.3827
    1.0000    0.0000
    0.9239   -0.3827
    0.7071   -0.7071
    0.3827   -0.9239
    0.0000   -1.0000
   -0.3827   -0.9239
   -0.7071   -0.7071
   -0.9239   -0.3827
   -1.0000    0.0000
   -0.9239    0.3827
   -0.7071    0.7071
   -0.3827    0.9239
    0.0000    1.0000

>> plot(t,z)
```

`z` is a 17 x 2 matrix now. The first column is sine of `t` the second is cosine of `t`. The brackets (`[]`) enclose the matrix definitions. This matrix definition was the concatenation of two 17 x 1 vectors. To do this, only a space is needed between the two vectors. If the vectors are a 1 x n vector, a semicolon is needed to separate them. For example:

```
>>a = [1:3 ; 4:6]

a =

        1        2        3
        4        5        6
```

(the default for vector assignment, if no increment is given, is one. `1:1:3` is the same as `1:3`)

For clarity, see what this statement does.

```
>>a = [1:3 4:6]
```

To learn more about matrix manipulation in Matlab, consider two examples. First is a repeat of the 2-equations, 2-unknowns problem from Dr. Johnson's class in the first semester. Briefly restated, given two unknown variables--x and y--and two linear equations like:

x+2y = 5
5x-y = 3

as a specific example, or in general:

Ax+By = C
Dx+Ey = F

where A through F are constants. We learned how to solve this using determinants. Matlab can solve this problem, as well. In order to understand more about how Matlab deals with problems like these, we need to understand some linear algebra notation. We can represent the two equations as follows:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

This is the same as saying

$$a_{11} \cdot x_1 + a_{12} \cdot x_2 = b_1$$

$$a_{21} \cdot x_1 + a_{22} \cdot x_2 = b_2$$

The matrix and vector notation in the above equations is commonly used and some students may have seen it in their high school algebra class. Unfortunately, there is a jump in notation from A and B to a_{11} and a_{12}, but understanding how to get from the Matrix notation **a** · **x** = **b** (the same equation as three above where **a** is a 2x2 matrix and **x** and **b** are a 2x1 vectors) to the two equations below is important. Matlab allows this matrix notation:

```
>>a = [1 2; 5 -1]
a =

         1         2
         5        -1

>>b = [5; 3]
b =
         5
         3
>>x = a\b
x =
         1
         2
```

WOW! Three command lines to Matlab and we solved the 2-equation, 2-unknown problem. This quick way to do it is not foolproof (checking for parallel and coincident lines), but it certainly demonstrates the power of Matlab.

Along with this matrix math power comes some nuances. For example, what does \ (left division) mean? Does / (right division) mean the same thing? The answer is no and the reasons are complex and involve some linear algebra. It is probably more confusing to show an example than not, but for those students with an insatiable curiosity, here goes:

```
>>x = a\b
```
is the same thing as
```
>>x = (b'/a')'
```
also, the following works
```
>>z = x/5
z =
   0.2000
   0.4000
```
a vector / a scalar is each element of that vector divided by that scaler.

The \ / difference may not be as common as the calculations coming next. Consider calculating the tangent of `t`. (Matlab is good for more than trigonometry. There are just a lot of useful examples to be found with trig functions). Matlab has the tangent function built-in, of course, but let's suppose we had to calculate it from the sine and cosine, tangent = sine/cosine.

```
>>clear
>>t = (0:pi/8:2*pi)';
>>sn = sin(t);
>>cs = cos(t);
>>tng = sn/cs
```

The screen will fill with output of several rows and columns. What happened? To find out, we need a few housekeeping commands, one of which to be described later is `clear`. The first though is `who`:

```
>>who

Your variables are:
t      sn       cs      tng
```

it tells us what variables are in the Matlab workspace. If `who` had been called before the `clear` command, it would have listed a bunch of variables, `t, y, z, a`, all from the above work. `clear` clears the workspace. If you want to delete only one variable,

```
>>clear a
```

clears just a. `who` tells us which variables exist in the workspace, `whos` tells which variable are in the workspace ***and*** how big each is. `whos` is a superset of `who`.

```
>>whos

     Name      Size Elements     Bytes     Density    Complex
       cs      17x1       17     136       Full       No
       sn      17x1       17     136       Full       No
        t      17x1       17     136       Full       No
      tng     17x17      289     2312      Full       No
```

tng is a 17x17 matrix. how did that happen? Well, Matlab got its name because it was made to handle matrices well and it automatically performs matrix math operations as we learned in the above example. That's what happened; an n by 1 vector was divided by an n by 1 vector to produce an n by n matrix...matrix division. We did not want a matrix division, though; we wanted sn(1)/cs(1) in tng(1) and sn(2)/cs(2) in tng(2) and so forth. This is called array division as opposed to matrix division. Another term which is useful to describe this is element-by-element division. So / in Matlab is matrix division, and `./` is element-by-element division. Likewise, `.*` is element-by-element multiplication and `.^` is element-by-element power.

The right way to calculate tng, therefore, is:

```
>>clear
>>t = (0:pi/8:2*pi)';
>>sn = sin(t);
>>cs = cos(t);
>>tng = sn./cs       % the only difference is the .


tng =
1.0e+016*

        0
   0.0000
   0.0000
   0.0000
   1.6332
   0.0000
   0.0000
   0.0000
   0.0000
   0.0000
   0.0000
   0.0000
   0.5444
   0.0000
   0.0000
   0.0000
   0.0000
```

The results look strange. Everything is 0.0000 x 10^{16} except two points which are large positive values.

```
>>plot(t,tng)
```

or

```
>>plot(t,tan(t))    %the matlab function
```

produce plots that look nothing like the tangent. Also, the first value is 0, but subsequent values are 0.0000. What gives? Well, Matlab after all is a numerical methods package and numerical round-off errors occur. Matlab knows what exactly zero is:

```
>>x = 0.0000000000000000

x =
        0
```

If it says `0` with no decimal point, then it is exactly zero. Matlab also knows what "exactly `1`" divided by "exactly `0`" and "exactly `-1`" divided by "exactly `0`" are:

```
>>1/x

Warning: Divide by zero

ans =

      Inf


>>-1/x

Warning: Divide by zero

ans =

     -Inf
```

But if it is slightly different from `0` like the `sin(pi)` that we saw in the beginning, Matlab can't estimate it as being infinity. Also, only 5 significant figures are shown, but Matlab keeps much more (the output format can be changed if you like with the `format` command).

Is tng right except for these cases very near + and - infinity? To check we can examine individual values of tng:

```
>>% the third element of the t array is pi/4,
>>% the third element of the tng array is tan(pi/4)
>>tng(3)

ans =

   1.0000
```

(tangent of pi/4 is 1.0)

We can also check tng values by modifying the plot of it. All we need to do is set the minimum and maximum of the y axis on the plot. How can we do that?

```
>>plot(t,tng)
>>axis([0 2*pi -2 +2])
```

After the plot command is entered, the figure appears in a new window on the screen. If the Matlab user just starts typing the next line, control goes back to the Matlab workspace and the figure is usually hidden behind it. After the `axis(..)` command is entered, the figure changes. We may not be able to see this change though because the figure is hidden behind the workspace window. The user can just alt-tab between applications to get back to viewing the figure window. `Axis(..)` is pretty self-explanatory; the minimum and maximum values for the x and y axes are in the brackets as follows `[xmin xmax ymin ymax]`. The resulting plot with a new y axis looks more like the tangent of t.

Whew! A lot has been covered so far. We saw how to make an array which increments from one value to another by a prescribed (or default) amount. There was a brief introduction to math functions and how to manipulate vectors, like the transpose, combining arrays to form a matrix, and vectorization, like `y = sin(t)`. Also, we learned about plotting and round-off features of Matlab. Matlab already seems like a pretty neat tool for working with numerical problems, but this introduction is only the tip of the iceberg. The scope of Matlab's utility becomes more evident as its user becomes more familiar with it.

Example 1: Linear Regression

Oftentimes, one is given a data set (a series of x,y points) from an experiment or test and is asked to fit a straight line through the data points. Consider, as an example, the following plot where force is plotted as a function of velocity (it could just as easily have been voltage as a function of current or something like that).

To fit a straight line through these data points, one may take a ruler and "eye-ball" the line that he or she thinks may be the best fit. That is a problematic way. First, its not reproducible. If that person tries to fit the same data set another day he or she may come up with a different answer. Second, it is subjective. The person fitting the line may have some agenda in mind when fitting the data. He or she may want the slope of the line to be a certain value and will be prejudiced in favor of that slope even if the best line does not appear to be through it.

Here's where linear regression comes in. It is an objective way to fit x,y paired data to the following equation y = m*x + b, where m is the slope and b is the intercept. The equations for m and b follow along with some other definitions. Let **x** and **y** be vectors of length n. the notation below x_1 and y_1 correspond in Matlab language to `x(1)` and `y(1)`. Then let

$$\bar{x} = \frac{\sum x_i}{n} = \frac{x_1 + x_2 + x_3 + \ldots + x_n}{n}$$

and

$$\bar{y} = \frac{\sum y_i}{n} = \frac{y_1 + y_2 + y_3 + \ldots + y_n}{n}$$

$\bar{x}$ and $\bar{y}$ are the average (or mean) values of the x and y data sets respectively. Calculation of these averages in C involves a `for` loop, but it is easier in Matlab as we'll see later. Then

$$m = \frac{n \cdot \sum (x_i \cdot y_i) - \left(\sum x_i\right)\left(\sum y_i\right)}{n \cdot \sum \left(x_i^2\right) - \left(\sum x_i\right)^2}$$

or in terms of $\bar{x}$ and $\bar{y}$

$$m = \frac{\sum (x_i \cdot y_i) - n \cdot \bar{x} \cdot \bar{y}}{\sum \left(x_i^2\right) - n \cdot (\bar{x})^2}$$

and

$$b = \bar{y} - m \cdot \bar{x}$$

Important to the linear regression is how good a fit is the line to the actual data. A correlation coefficient r^2 is used here. r^2 of 1 is a perfect line, r^2 of 0 is lousy. Some engineers and scientists like their r^2 very high (> 0.95), but it really depends on the circumstance. In one of Dr. Olson's electronic calibration experiments where some standard electronic parts are used, an r^2 of less than 0.998 usually means trouble, but in some of his biological data, an r^2 of 0.90 may be acceptable. The equation for r^2 is as follows.

$$r^2 = \frac{\left[\sum (x_i - \bar{x})(y_i - \bar{y})\right]^2}{\left[\sum (x_i - \bar{x})^2\right]\left[\sum (y_i - \bar{y})^2\right]}$$

As alluded to above, all these sums will require `for` loops in C, but built-in Matlab functions makes it easier. So, lets perform a linear regression in Matlab.

Step 1: loading a data set into Matlab and introduction to script files

Discussed below will be three ways to load an x,y data set on to Matlab and view the results: 1) the direct way on the workspace, 2) the "script file" way, and 3) the "data" file way.

1) The direct way. Here is one way (of many) to manually load data onto the workspace. We want to enter each x,y data pair as a set to avoid confusion. Again using the example, the x variable is force and the y variable is velocity. There will be 15 x,y data pairs in the data set and we will enter them as x_1, y_1, x_2, y_2, etc.

```
>>data = [
-0.0006741     5.4128;
0.0545  7.5800;
0.0898  11.2001;
0.0820  14.1576;
0.1589  14.7301;
0.1736  20.9915;
0.2088  25.7235;
0.2336  26.1302;
0.3017  30.4154;
0.3117  35.5325;
0.3350  37.5553;
0.3609  39.8222;
0.4384  43.4745;
0.4294  47.3630;
0.4854  49.8659]

data =

-0.0007   5.4128
 0.0545   7.5800
 0.0898  11.2001
 0.0820  14.1576
 0.1589  14.7301
 0.1736  20.9915
 0.2088  25.7235
 0.2336  26.1302
 0.3017  30.4154
 0.3117  35.5325
 0.3350  37.5553
 0.3609  39.8222
 0.4384  43.4745
 0.4294  47.3630
 0.4854  49.8659
```

For entering the data, the x-y pair must be separated by a space and the pairs must be separated by a semicolon. The number of significant figures is arbitrary, and the spaces and new lines are all purely for readability (Matlab only needs one space between x and y and doesn't need new lines). The brackets are also mandatory. Now we want to put force in one vector (15x1 array) and velocity in another and plot them like on page 10.

```
>>velocity = data(:,1);

>>% this means put all the rows (:) of the first column into a variable
>>%called velocity.

>>force = data(:,2);

>>plot(velocity,force,'co')

>>%the 'co' means cyan circles.
>>% If nothing is mentioned it is a line of a default color.
>>% details to make the graph look nice
>>axis([-0.01 0.59 0 55])
>>xlabel('Velocity (m/s)')
>>ylabel('Force (N)')
>>title('X-Y Plot of Force versus Velocity')
>>grid on
```

2) The script file way. Above was one manual way to load experimental data onto Matlab's workspace. If something were to happen (computer crashing, you wanting to reload on a different machine, or something like that), you would have to retype every command. To avoid this, Matlab has programming capabilities. Just like in C, you had the filename.c file, in Matlab, you have the filename.m file. In Matlab, though, you don't need to compile, link and execute, it automatically runs in Matlab. There are two kinds of *.m or m-files: Script files, and Function files. First is the script m-file. The script m-file gets its name because it is a sequential list of Matlab commands given so that Matlab can read them as an actor reads a script. Here is a script m-file called `loaddata.m` which does the same thing as the Matlab command lines above.

```
data = [
-0.0006741 5.4128;
0.0545  7.5800;
0.0898  11.2001;
0.0820  14.1576;
0.1589  14.7301;
0.1736  20.9915;
0.2088  25.7235;
0.2336  26.1302;
0.3017  30.4154;
0.3117  35.5325;
0.3350  37.5553;
0.3609  39.8222;
0.4384  43.4745;
0.4294  47.3630;
0.4854  49.8659];

velocity = data(:,1);

force = data(:,2);

plot(velocity,force,'co')

axis([-0.01 0.59 0 55])
xlabel('Velocity (m/s)')
ylabel('Force (N)')
title('X-Y Plot of Force versus Velocity')
grid on
```

In Matlab, here is how a script m-file is run

```
>>loaddata
```

(make sure to press return after the filename and no .m extension) Up pops the plot from before. You must make sure that the m-file is in the directory in which Matlab is running. A good way to ensure that is to use Matlab's file editor. By learning this, Matlab has suddenly turned from a neat command-based tool into a programming language, too.

3) the data file way. In some situations, it makes more sense to store data (like the `data` array above) in a separate file. How then can you load this data file onto the Matlab workspace? The `load` command will help us. Consider the following data file called `data.in`

```
-0.0006741 5.4128
0.0545   7.5800
0.0898   11.2001
0.0820   14.1576
0.1589   14.7301
0.1736   20.9915
0.2088   25.7235
0.2336   26.1302
0.3017   30.4154
0.3117   35.5325
0.3350   37.5553
0.3609   39.8222
0.4384   43.4745
0.4294   47.3630
0.4854   49.8659
```

To import this data file onto Matlab, do the following:

```
>>clear
>>% housekeeping task to clear all variables off workspace
>>load data.in
```

Now, in the workspace is a variable called `data` (the data filename without the extension). It is a 15x2 array with the data in `data.in` in it (say three times real fast... data in data.in in it, data in data.in in it, ...). So now, the rest of the loading of the data and plotting is the same as the script m-file above. Consider a new script m-file called `loadin.m` which does the same thing as the one on p.13.

```
load data.in

velocity = data(:,1);

force = data(:,2);

plot(velocity,force,'co')

axis([-0.01 0.59 0 55])
xlabel('Velocity (m/s)')
ylabel('Force (N)')
title('X-Y Plot of Force versus Velocity')
grid on
```

Now to run this m-file, just

```
>>loadin
```

This is an organized and efficient way to load data onto Matlab.

Step 2: making a linear regression function and introduction to function m-files:

Matlab has a vast number of functions,

```
>>y = sin(x);
>>[ymax,index] = max(y);
```

but some, like the common linear regression, are curiously absent from Matlab. That's not such a big deal, though, because Matlab gives us the ability to *write our own functions*, and, if we do it right, we only have to write a linear regression function once. This gives us even more power than the script m-files do and we can customize the Matlab language to our own preferences. First, a few details about Matlab functions in general. on the left-hand side of the equal sign is the output(s) of the function. There can be more than one, but as in the max example above, they must be surrounded by brackets and separated by commas. On the right-hand side of the function is the name of the function and then, in parenthesis, the input(s) to the function. Again, there can be more than one input, but they must be separated by a comma. The goal of writing functions in Matlab(and the equivalent subroutines in other languages like C, if you are familiar with that) is creating *Modular* programming, that is, separating easily defined tasks into different regions of code. Some computer progammers would assert that it is easier to understand the following script m-file called `loadreg.m`

```
% first load the data
loadin

% then calculate the linear regression
[m,b,r2] = linreg(velocity,force)

%where linreg is as yet an undefined function that we will create later
```

This "program" or script m-file has only two lines, and it is very intuitive what it does. The first line loads the data in and the second calculates the slope, m, the y intercept, b, and the correlation coefficient, r2, given the x and y data (velocity and force). The details of `loadin` and `linreg` can be discovered separately by looking at `loadin.m` and `linreg.m`, respectively.

In a sense, we're almost done with Example 1. All we need to do is create this wonderful `linreg` function. Here is how to make a function m-file. It is absolutely essential that the m-file name agree with the function name, that is, this m-file must be called `linreg.m`. Here's what's in this file. We'll go over what each line means below it.

```
function [m,b,r2] = linreg(x,y)
%  FUNCTION [M,B,R2] = LINREG(X,Y)
%    calculates the slope, m, intercept, b, and correlation coefficient,
%    r^2 given, an independent variable, x, and dependent variable, y.
%    x and y must have the same number of points and be the same
%    dimension.
%

n = length(x);
xbar = mean(x);
ybar = mean(y);

m = (sum(x.*y)-n*xbar*ybar)/(sum(x.^2)-n*xbar^2);
b = ybar - m*xbar;
r2 = ((sum((x-xbar).*(y-ybar)))^2)/(sum((x-xbar).^2)*sum((y-ybar).^2));
```

It's pretty short considering it's power. The main difference, upon inspection, between a script m-file and a function m-file is the first line. A function m-file starts with the line `function` and then what follows is the *form* of how the function should be called, specifically the number of inputs and the number of outputs. The fact that the left-hand side of the function call in this function m-file is the same as that in the script m-file `loadreg.m` is just a coincidence. They don't have to be the same name, there just has to be three output variables in brackets and two input variables of the same size, nx1 or 1xn. For example, the following would work from the Matlab command prompt:

```
>>[tom,dick,harry] = linreg(tweedledee,tweedledum);
```

assuming `tweedledee` and `tweedledum` are the right size.

Another important point about functions is that the only variables that are shared between the function and the command workspace are those which are passed on the left and right-hand sides of the function. If, in the workspace, there is a variable called `needbad` which you needed for the linear regression calculation (there isn't, but just imagine). Including the following line in the `linreg.m` file would produce an error

```
b = b*needbad;
```

When you called `linreg` from the command prompt, Matlab would beep and tell you that `needbad` is undefined. Moreover, the variables calculated in `linreg`, that is, `n`, `xbar`, and `ybar`, are inaccessible from the main workspace. They remain local to the function only, and when the function is through with its calculation, they are erased from memory.

There is a way, other than passing through the function call, of getting variables to be shared between functions and the Matlab command workspace. It uses the command `global` and if the reader is more interested, he or she can look it up on Matlab.

The comments below the function call are of a specific form, first the function call in all caps, then a description of each of the variables to be passed. Every function you write should have comment lines of this form *immediately* below the `function` statement (although the function still works without it). Why? Look what happens when you do the following from the Matlab command prompt:

```
>>help linreg
  FUNCTION [M,B,R2] = LINREG(X,Y)
    calculates the slope, m, intercept, b, and correlation coefficient,
    r^2 given, an independent variable, x, and dependent variable, y.
    x and y must have the same number of points and be the same
    dimension.
```

You've created your own on line help for this function you've wrote! This is extremely helpful if, for example, you can't quite remember how to call this function. Of course, there is on-line help for all built-in Matlab commands and they are of this exact type. One goofy thing about Matlab help is that the function call is given in all caps, but, because Matlab is case-sensitive, if the user follows the *exact* form given in the help output, Matlab would produce an error. It is a goofy convention, but it is consistent, so one can get used to it.

One last point about this function `linreg` is that it is not foolproof, that is, we did not check for invalid inputs, etc. That is a good thought exercise for the reader to end this example.

Example 2: Fourier Domain Transfer function (for only the brave at heart. Save until Junior year)

Fundamental to engineering practice is the use of linear theory in the Fourier (or Laplace) domain. To wit:

$$Y(j\omega) = X(j\omega) \cdot H(j\omega)$$

where Y(jω) is the output signal, X(jω) is the input signal and H(jω) is the system transfer function. Why do engineers mess around with the fourier domain? In the time domain, y(t) is x(t) convolved with h(t) which can sometimes be sticky, and h(t) may not be easily obtained, whereas H(jω) can have an analytic form (Proof of this statement can be easily found in a circuits book).

Consider the following: The frequency domain transfer function for a given biological system (mass exchange in an organ, but it could just as well been from circuit analysis) was derived from physical principles to be:

$$H(j\omega) = e^{-\left(a \cdot \frac{j\omega}{j\omega + b}\right)}$$

where a and b are given constants. Thus, H(jω) is easily calculable given a range of jω.

A typical job for an engineer is to take a measured input function x(t), have a known H(jω), and calculate y(t) from them. Matlab is well-suited to perform such a task. Assume x(t) is of the form shown in the following figure. Also suppose it exists in an ascii file (two columns, first one is time, second one is x(t)) named `input.dat` on computer. So the first step in getting y(t) from the x(t) in this file and the H(jω) in the above equation is getting x(t) into Matlab.

File Input

Two Matlab commands are needed for file input/output: `load`, and `save` (more on save later). Matlab has its own file format for saving and loading variables, but it also works with ascii files. How to use `load` to get `t` and `x(t)` onto the Matlab workspace is as follows. First make sure that `input.dat` is in the matlab directory, then:

```
>>load input.dat
>>t = input(:,1);
>>x = input(:,2);
plot(t,x)
```

The figure in this handout is what will be displayed. Executing `whos` will show that `t` and `x` are each 256x 1 arrays of data. `input` is a 256x2 matrix. This is interesting. `load` took a data file which itself was in a matrix format and converted it into a matrix in Matlab with the variable name the same as its filename (extension excluded). That's easy!

The next line assigns the first column of `input` to `t`. The notation after the variable name `input`, `(:,1)` means the following: `:` means all the rows, and `1` means the first column. If `input` was 400 points long and you only wanted the first 256, the following statement would work:

```
>>t = input(1:256,1);
```

There are some caveats to load. Matlab, unlike spreadsheet importing, is pretty sensitive to stray characters at the beginning of a file, so some editing of a file may be necessary before it can be successfully input into the Matlab workspace. C commands like `fscanf` can be made to work in Matlab, too, so formatted input is also possible.

Script Files and Language

Matlab is more than a powerful command line interpreter. Extensive computer programs can be written and executed with Matlab. The way to use Matlab as a program executor is using the m-file. m refers to the extension of an ascii text filename with Matlab commands in it. The script m-file is a "script" or list of instructions which the Matlab interpreter will execute sequentially. Consider the following m-file called `loadin.m` which was made using the MS-DOS editor (any old ascii file generator will do). The contents of this file are as follows:

```
load input.dat
t = input(:,1);
x = input(:,2);
plot(t,x)
```

Except for the >> in front of each line, it is exactly like the command line example above. In Matlab, this script m file can be executed.

```
>>clear
>>loadin
```

This script file will produce the exact same result as if we had typed these lines in by hand. For a simple four line script file, it hardly seems worth it, but this ability transforms Matlab into a programming language.

The following example shows some of Matlab's programming power. It would be nice to write a subroutine or function which computed an output `y(t)`, given `t`, input `x(t)`, and `a` and `b`. Maybe we'd call this dream function `in_2_out` and using it would look something like this:

```
>>clear
>>loadin
>>y = in_2_out(t,x,a,b);
>>plot(t,x,t,y)
```

The answer to our linear systems problem would be plotted on our screen. The code is short and easy to understand: y is the output of this function which needs the inputs t,x,a, and b. The only trick we need to make this dream a reality is the ability to write functions in Matlab. Fortunately, Matlab allows two kinds of m-files: script files, and functions.

Making Your Own Matlab Function

Matlab function m-files are very similar to script m-files. There are only two substantial differences: 1) the first line must declare the m-file as a function, and 2) all the variables created in the function do not exist in the workspace after the function is through calculating the output. The second difference means that the workspace and the function must pass variables to each other through the function call, just like a C subroutine. As importantly as it is with script m-files, the function name (in_2_out, in the above) must be the m-file name also, i.e. in_2_out.m. Here is the contents of a working in_2_out.m:

```
function y = in_2_out(t,x,a,b)
%   FUNCTION Y = IN_2_OUT(T,X,A,B)
%   outputs an array y of same length as t and x with and
%   satisfying the equation Y(jω) = X(jω)·H(jω), where
%   H(jω) is exp{-a·[jω/(jω+b)]}
%   t and x must have an even number of points.
%   a and b are input constants
%

X = fft(x);  %fft is the fast fourier transform

wo = 2*pi/(max(t));      % omega_o is the fundamental frequency
                         % wo = 2pi/the period T.

w = (0:wo:wo*(length(t)-1))';   %calculating the omega array

H = exp(-a*(j*w)./(j*w+b));     %calculating the transfer function


% housekeeping chores to make sure the transfer function folds over
H(length(H)/2+1:length(H)) = zeros(length(H)/2,1);
HF = flipud(H);
HFS(2:length(HF)) = HF(1:length(HF)-1);
HFS(1) = 0;
H = H + HFS';

%calculating output
Y = x.*H;

y = abs(ifft(Y));   %take the absolute value of the inverse fast
                    %fourier transform to get y(t)
```

This function is densely packed with new information. It also is powerful given it performs the in_2_out function needed for this example. As remarkable as its power is that it can be done in only 12 lines of Matlab code not including comment lines!

The first line declares the m-file to be interpreted as a function, not a script file. The way that the programmer calls the function from matlab or a script m-file must agree exactly with how it is recorded on this first line, although the names of the variables can be different. If multiple outputs to the function are desired, for example, if the user wanted the omega array and the transfer function H(jω) in the ouput also, the function could be declared as such:

```
function [y,w,H] = in_2_out(t,x,a,b)
```

There is virtually no practical limit to the number of outputs and inputs to a function (I think it's 50 or some outlandish number).

Immediately below the function declaration is a contiguous string of comment lines in a pretty uniform format. These comments aid the Matlab user at the command prompt:

```
>>help in_2_out

   FUNCTION Y = IN_2_OUT(T,X,A,B)
   outputs an array y of same length as t and x with and
   satisfying the equation Y(jω) = X(jω)·H(jω), where
   H(jω) is exp{-a·[jω/(jω+b)]}
   t and x must have an even number of points.
   a and b are input constants
```

Neat, huh? After the first noncomment line, the help output stops. The next line, `X = fft(x);`, teaches two things: 1) Matlab is case sensitive, `X` is not the same as `x`, and 2) functions like the fast fourier transform are built-in to Matlab. It is a convention that Matlab users use capital letters for fourier domain arrays and lower case letters for time domain arrays, but there is nothing in Matlab's syntax that prescribes it. Much of the rest of this function is self-explanatory. `max(t)` will give the last value of the `t` array (which is also its maximum value and the period, of course). `length(t)` gives the length (i.e. number of elements) of the `t` array. In the definition of `H`, `j` is predefined by Matlab as the imaginary variable (the sqare root of -1). The only hard part is the housekeeping stuff needed to ensure that the transfer function in this digitized fourier domain folds over around the Nyquist frequency. It is probably a good idea to ignore the details of this until much later in the Matlab learning curve.

Ascii File Output

The `save` command is useful for outputing data from Matlab. Suppose, an output file with `t` as the first column, `x` as the second column and `y` as the third was desired. The following works:

```
>>out = [t x y];
>>save output.dat out -ascii;
```

That's apparently as easy as load. If the `-ascii` part is not included in the command, then Matlab will save it in its own special file format.