<u>MATLAB GUIDE UMD PHYS375 FALL 2010</u>

DIRECTORIES

Find the current directory you are in:

```
>> pwd

ans =

C:\Documents and Settings\ian\My Documents\MATLAB
```

[Note that Matlab assigned this string of characters to a variable name "`ans`". It does this every time it is asked to produce something without being told where to assign the value. For instance,

```
>> 1+1

ans =

     2 ]
```

You can also assign the value to a variable of your choice:

```
>> a=pwd

a =

C:\Documents and Settings\ian\My Documents\MATLAB
```

Matlab will echo the explicit command unless you end your statement with a semicolon:

```
>> a=pwd;
```

All the usual MSDOS directory manipulation applies:

```
>> cd ../../Desktop
>> dir

.                    CLASSES              PROPOSALS
..                   PRESENTATION         PROJECTS
```

Note that Matlab is case-sensitive, and all built-in commands we will be using are lowercase.

<u>Change directories to C:\Users\p375-0X\Documents\MATLAB ("X" varies at each lab station) and make a subdirectory with your last name. You will be using this directory all semester. It is also recommended that you back up this data on a USB thumb drive.</u>

MATRICES AND ARRAYS

From the Matlab "Getting Started" Guide
(http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/getstart.pdf):
"In the MATLAB environment, a matrix is a rectangular array of numbers. Special meaning is sometimes attached to 1-by-1 matrices, which are scalars, and to matrices with only one row or column, which are vectors. MATLAB has other ways of storing both numeric and nonnumeric data, but in the beginning, it is usually best to think of everything as a matrix. The operations in MATLAB are designed to be as natural as possible. Where other programming languages work with numbers one at a time, MATLAB allows you to work with entire matrices quickly and easily."

You can create an array of data with square brackets:

```
>> a=[3 5 9 6]

a =

     3     5     9     6
```

Here, it is a row vector. You can create a column vector as well:

```
>> a=[3; 5; 9; 6]

a =

     3
     5
     9
     6
```

Or just use the transpose operator on a row vector:

```
>> a=[3 5 9 6]'

a =

     3
     5
     9
     6
```

Note that it is sometimes useful to add comments to your commands. You can do this with "%":

```
>> a=[3 5 9 6] %here is my comment

a =

     3     5     9     6
```

At any time you can see the variables in the workspace and their properties including dimensions etc:

```
>> whos
  Name      Size                Bytes  Class     Attributes

  a         1x4                    32  double
```

You can save this array to a delimited text file named e.g. "data.dat"

```
>> save -ascii data.dat a
```

You can load data from a space/tab/etc delimited file of numbers:

```
>> load data.dat
```
This creates an array named the same as the file, without the extension:
```
>> whos
  Name      Size            Bytes  Class     Attributes

  a         1x4                32  double
  data      1x4                32  double
```

You can also get information about a specific variable:
```
>> length(a)

ans =

     4
```

You can concatenate these elements to other arrays:
```
>> data=[[1 5 3 8]' data']

data =

     1     3
     5     5
     3     9
     8     6
```

Note that these are all logical statements, not algebraic. They can therefore be self-referential.
Since this is a 2-dimensional array (4 rows and 2 columns), we can see the length is
```
>> length(data)

ans =

     4
```

but the number of elements is
```
>> numel(data)

ans =

     8
```

And the size reports the length of the rows and columns:
```
>> size(data)

ans =

     4     2
```

(BTW, there exists documentation on all the functions available in MATLAB; for instance
http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/refbook.pdf,
which is just for functions starting with A-E and takes up >1500 pages. The other two volumes for the rest of the alphabet are refbook2.pdf and refbook3.pdf. Rather than burden you with sorting through them, we will focus on the functions of greatest utility that will be used the most in the lab.)

You can address individual elements of this array (row index first, column index second):
```
>> data(1,1)
```

```
ans =

     1

>> data(4,1)

ans =

     8
```

You can also use this address syntax to assign values:

```
>> data(2,1)=18

data =

     1     3
    18     5
     3     9
     8     6
```

The colon ":" is extremely useful in Matlab. You can use it to define subsets of elements of arrays:

```
>> data(1:2,1:2)

ans =

     1     3
     5     5
```

Or, use it to create arrays of evenly spaced numbers. For unit spacing:

```
>> 1:4

ans =

     1     2     3     4
```

Or, for other spacing values e.g.

```
>> 0:5:15

ans =

     0     5    10    15
```

<u>Q: How would you create an array of all odd numbers up to 99?</u>

To define simple arrays, you can use these two self-explanatory commands:

```
>> ff=zeros(2,1)

ff =

     0
     0

>> ff=ones(2,1)

ff =

     1
     1
```

To create an array of random values equally distributed between 0 and 1:
```
>> ff=rand(2,1)

ff =

    0.8147
    0.9058
```

[Note that `rand`, by itself, is equivalent to `rand(1,1)`]

If you want to erase all variables from the workspace use "`clear`",
```
>> clear
```

Then there are no available variables:
```
>> whos
>>
```

OPERATIONS ON ARRAYS

Matlab can do statistical analysis:
```
>> a=rand(10,1);
>> mean(a) % find the mean value of this array of random numbers

ans =

    0.5777

>> std(a)  % find the standard deviation of this array of random numbers


ans =

    0.2985
```

<u>Q: The example above is for only 10 numbers, and if you run the commands again you will get a slightly different answer. What values do the mean and standard deviation approach in the limit of a large number of random variables evenly distributed between 0 and 1? Prove that this is correct analytically.</u>

Be aware that one of the most useful MATLAB commands is "`help`", e.g.
```
>> help mean
 MEAN   Average or mean value.
    For vectors, MEAN(X) is the mean value of the elements in X. For
    matrices, MEAN(X) is a row vector containing the mean value of
    each column.  For N-Dimensional arrays, MEAN(X) is the mean value of
the
    elements along the first non-singleton dimension of X.

    MEAN(X,DIM) takes the mean along the dimension DIM of X.

    Example: If X = [0 1 2
                     3 4 5]

    then mean(X,1) is [1.5 2.5 3.5] and mean(X,2) is [1
                                                      4]

    Class support for input X:
       float: double, single

    See also median, std, min, max, var, cov, mode.

    Overloaded functions or methods (ones with the same name in other
directories)
       help timeseries/mean.m

    Reference page in Help browser
       doc mean
```

You can not only find out how to use a function, but also discover other related functions that might be helpful as well in the "`See also`" section above.

Very often we need to calculate these parameters not of arrays of random variables, but of probability distributions y(x).
For instance,

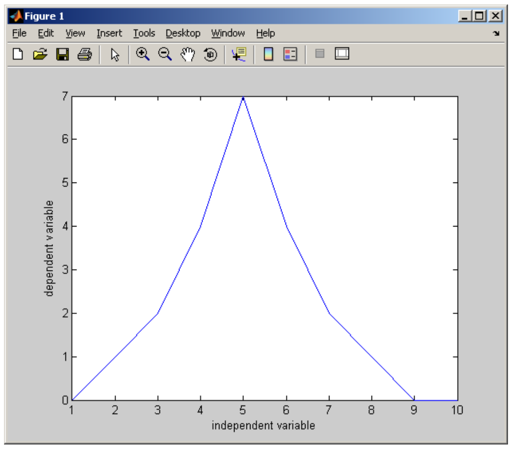
```
>> x=1:10; y=[0 1 2 4 7 4 2 1 0 0];
```

We can plot a figure to visualize this distribution:

```
>> plot(x,y)
```

And label the axes:

```
>> xlabel('independent variable')
>> ylabel('dependent variable')
```



We can even print this to an image file with one of:

```
>> print('-dbmp','fig1.bmp')
>> print('-dpdf','fig1.pdf')
>> print('-djpeg','fig1.jpg')
```

etc., where the first argument specifies the file type and the second defines the filename of the resulting image.

The mean value of x (the first moment of the distribution y(x)) is the weighted sum of x, normalized by the distribution:

$$\bar{x} = \frac{\sum_{i=1}^{N} y_i x_i}{\sum_{i=1}^{N} y_i}$$

We can implement this in MATLAB via

```
>> xbar=sum(y.*x)/sum(y)

xbar =

     5
```

where here I have used the element-by-element multiplication operator "`.*`". **(Notice the dot!)**
<u>What would happen if I only used "`*`"?</u>
Or, more simply, (making use of the transpose "`'`")

```
>> xbar=y*x'/sum(y)
```

```
xbar =

     5
```

Or, in terms of the dot product,

```
>> xbar=dot(y,x)/sum(y)

xbar =

     5
```

We can then calculate the variance of the distribution:

$$\sigma_x^2 = \frac{\sum_{i=1}^{N} y_i (x_i - \bar{x})^2}{\sum_{i=1}^{N} y_i}$$

```
>> v=dot(y,(x-xbar).^2)/sum(y)

v =

     2
```

Where I used the element-by-element power operator "`.^`".
What would happen if I used only "`^`" without the dot?

From this, one can calculate the standard deviation of the distribution, σ_x,

```
>> s=sqrt(v)

s =

    1.4142
```

Q: Test this procedure out on the function $e^{-\frac{(x-\bar{x})^2}{2\sigma^2}}$. This is called the "gaussian" function: Use $\bar{x} = 5$ and $\sigma = 1$. Use `plot()` to create a figure of the distribution from x=0 to x=10. Hint: use `exp()`. Do you get the expected values of variance and mean? How can you get closer to the *analytic* limit?

SCRIPT AND FUNCTION FILES

You can eliminate a lot of typing at the command prompt by saving commands to a script – essentially a text file that contains the commands, but is always named with a ".m" extension. Just type the name of the script (without the ".m") at the command prompt to run it. For example, if your file "script1.m" contains the lines

```
clear;
N=1e2;
A=rand(1,N);
M=mean(A);
```

```
disp(M)
```

(notice that it is very often a good idea to start every script with "clear" to avoid variable name ambiguities and errors)

You would see something like

```
>> script1
    0.4990
```

(Make sure the .m file is in your current directory!) Notice that I used the exponential notation "1e2" to represent "100" in the above. Also, note that "`disp()`" displays the value of a variable.

If we want to easily interactively see the effect of e.g. larger and larger numbers of random variables without directly editing the value of `N` in our script file, we can make another .m file which is a ***function*** that takes `N` as an argument on the command line. The following is the contents of a file func1.m:

```
function mymean=func1(N)

A=rand(1,N); %generate an array of random variables between 0 and 1
mymean=mean(A); %calculate the mean of the array
```

(Note that the function definition on the first line is the same as the filename without the ".m", and the value it returns is similarly defined on the same line before the "=").

Then we can set the value of `N` on the command prompt,

```
>> func1(1e6)

ans =

    0.5003
```

Create the function file above and test it in your command window.

CONTROL STRUCTURES

There are several elements of simple programming that we will use:

Conditionals are also called "if statements". For instance, the following function prints something conditional on the input:

```
function out=if1(in)

if in==1
    disp(['input is ' num2str(in) '!'])
    out=4;
elseif in==2
    disp(['INPUT IS ' num2str(in) '!'])
    out=9;
```

```
else
    disp(['input is not 1. It is ' num2str(in)])
    out=0;
end
```

Note that I made use of "`num2str()`" which does just what its name implies: convert a number to a string. Then, it is concatenated with string constants to create the displayed text. Another useful function like this is "`int2str()`" which only works for integers but will suppress printing out a number to 4 decimal places or more.

We can run it with arbitrary input:

```
>> b=if1(1)
input is 1!

b =

     4

>> b=if1(10)
input is not 1. It is 10

b =

     0
```

<u>Create the function above and test it in your command window.</u>

<u>Loops</u> are extremely important for programming MATLAB to repeat a measurement in our labs. For instance, if we want MATLAB to do something 10 times,

```
for ii=1:10
 rr(ii)=ii^2;
end

>> rr

rr =

     1     4     9    16    25    36    49    64    81   100
```

The loop sets "ii" equal to successive values of the array 1:10 every time it executes until it has looped through all the values.

Note that this is a particularly BAD way to produce an array of squares because MATLAB is optimized to operate on whole arrays, not by looping through an array's elements

<u>Q: How would you construct an array of squares with a **one-line command**?</u>

There are other types of loops. For instance,

```
ii=0;
while rand<0.9
 ii=ii+1;
end
```

```
disp(ii)
```

The loop will execute the code before the "end" statement and increment "`ii`" by 1 until the random number generated by "rand" is equal to or more than 0.9. Once it satisfies this constraint, it will display the number of loop iterations.

Q: what is the mean value of `ii` after this code executes a large number of times? Write a function and script which calls it to show this.

Note that it is possible to write (faulty) code for a loop which will never end if the "while" statement is always satisfied. If your code gets into an infinite loop, you can stop it by pressing control-c.

PROBLEMS:

1. What does this script do?

```
clear;
vals=1:10;
for ii=vals;
     v(ii)=rand;
     x(ii)=ii;
     pause(1);
     plot(x,v);
end
```

How would you achieve the same output with a one-line command?

2. MATLAB can be used for calculations other than data acquisition and statistical analysis. For instance, it can be used for "Monte-Carlo" calculations, which are especially useful for evaluating multidimensional integrals:
Imagine that despite your ability to easily evaluate a function describing a closed surface, integrating it can not be done analytically. The Monte-Carlo technique would be to simply draw rectangular bounds along all dimensions, and uniformly sample the (known) interior volume, counting only the points which fall inside the boundary of integration. Then, the integral is simply the sum of the samples, normalized by the total number of sample points.
Write a MATLAB script to evaluate the volume of a sphere of known radius using this Monte-Carlo method. Compare this to the analytic solution, and extract a calculated value for π. What happens as the number of sample points increases?

3. MATLAB can be used for physics simulation. For instance, in electrostatics, the problem is often simply just to find the potential everywhere inside a region with known boundary conditions. If the dielectric constant is the same everywhere, Gauss' Law implies the potential solves "Laplace's equation" $\nabla^2\phi = 0$, which is just a statement that the solution has zero total curvature. In 2 dimensions, we can use the definition of the second derivative to arrive at a condition for the values of the solution on a regular grid:

$$\phi_{i,j} = \frac{\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1}}{4}$$

where *i* and *j* are the indices along *x*- and *y*- directions. In other words, the value of the potential at every point is just the average of all the nearest neighboring points. With this in mind, run the code below in a script and use comments to add descriptions of the role of each of the blocks of commands:

```
clear

NUM=21;
THRESH=1e-6;
M=zeros(NUM);

AVG=(diag(ones(NUM+1,1),1)+diag(ones(NUM+1,1),-1))/4;

newM=rand(NUM);
while (sum(sum(abs(newM-M))) > THRESH)

  newM=M;

  M=[zeros(1,NUM); M; zeros(1,NUM)];
  M=[zeros(NUM+2,1) M zeros(NUM+2,1)];
  M(12,12)=1;

  M=AVG*M+(AVG*M')';

  M=M(2:(NUM+1),2:(NUM+1));

end

M=[zeros(1,NUM); M; zeros(1,NUM)];
M=[zeros(NUM+2,1) M zeros(NUM+2,1)];
M(12,12)=1;

surf(M)
```

What physical system is this code modeling?