MATLAB GUIDE UMD PHYS401 SPRING 2012

We will be using Matlab (or, equivalently, the free clone GNU/Octave) this semester to perform calculations involving matrices and vectors. This guide gives a brief introduction of some useful commands to get you started.

Firstly, note that documentation on all the functions available in MATLAB are at 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 just a few functions of greatest utility. Furthermore, at any time you can find out more on what a function does and suggestions for other related useful functions by typing *help <function>* at the command prompt.

For instance:

```
>> help help
   HELP Display help text in Command Window.
      HELP, by itself, …
```

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 at the bottom of the text now printed in the command window.

Arrays

Arrays (matrices and vectors) can be defined by hand. For instance, this command using square brackets assigns a row vector of the numbers from 1 to 3 to the variable *x*:

```
>> x=[1 2 3]

x =

     1     2     3
```

But we can simplify this command by specifying just the limits of the array:

```
>> x=1:3

x =

     1     2     3
```

Or, for other spacing values e.g.

```
>> 0:5:15

ans =

     0     5     10     15
```

Note that you can also use `linspace` to generate equally-spaced numbers. For example, you could produce the same output as above by finding the 4 equally spaced numbers between 0 and 15 (inclusive):

```
>> linspace(0,15,4)
```

```
ans =

     0     5    10    15
```

A column vector is defined either by ending each row with a semicolon

```
>> x=[1; 2; 3]

x =

     1
     2
     3
```

Or transposing a row vector:

```
>> x=[1 2 3]'

x =

     1
     2
     3
```

Note however that this single-quote command not only transposes but also conjugates, so when using complex-valued matrices all imaginary components change sign:

```
>> x=[1 i]'

x =

   1.0000
        0 - 1.0000i
```

We can select an element of the array by indicating the index:

```
>> y=2:7; a=2; y(a)

ans =

     3
```

Note that I have concatenated commands (and suppressed echo) with a semicolon.
We can likewise select a subset of elements by indicating the array of indices

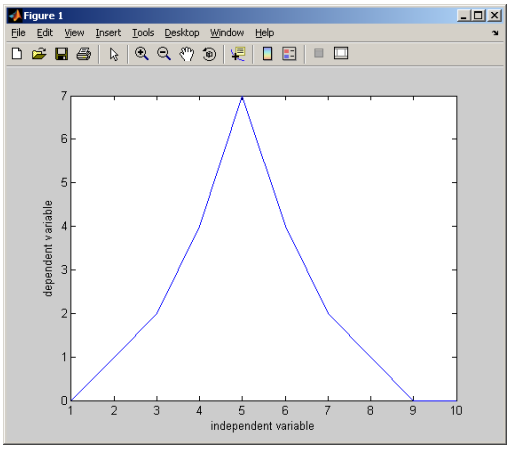
```
>> b=5; y(a:b)

ans =

     4     5     6
```

You can plot vectors of equal length:

```
>> x=1:10; y=[0 1 2 4 7 4 2 1 0 0];
>> plot(x,y)
```

This plots *x* vs *y*. We can label the axes:

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



We 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.

Matrices

2-dimensional arrays (matrices) are constructed similarly to 1-dimensional vectors. For instance, we can define the 2x2 identity

```
>> z=[1 0; 1 0]

z =

     1     0
     0     1
```

But Matlab has a shortcut for this:

```
>> eye(2)
```

This shortcut is, of course, much easier to use than explicitly entering elements when the matrix becomes very large!

Matlab also has a random number generator, so we can make random arrays by specifying the size in *row, column* format (a row vector will always have 1 column and a column vector will always have 1 row):

```
>> rand(1,2)

ans =

    0.8147    0.9058
```

For square matrices which have the same number of rows and columns, this can be simplified by using only one value:

```
>> rand(2)

ans =

    0.1270    0.6324
    0.9134    0.0975
```

Because these are random numbers, you will get different results everytime you execute the command. [Note that `rand`, by itself, is equivalent to `rand(1,1),` and produces a single random number]

Other useful matrix-generating functions are *zeros()* and *ones()*, which use the same row, column format as above. Try them! (Or use *help* to learn more about it!)

Note that the element-addressing scheme used for vectors also works here. For example,

```
>> id=eye(4); id(2:3,2:3)

ans =

     1     0
     0     1
```

You can also use this address syntax to assign values:

```
>> A=zeros(2); A(1,1)=3

A =

     3     0
```

```
     0     0
```

You can choose an entire column of a matrix by specifying all rows within a column. For instance, the first column is:

```
>> A(:,1)

ans =

     3
     0
```

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

```
>> whos
  Name      Size             Bytes  Class     Attributes

  A         2x2                 32  double
  a         1x1                  8  double
  ans       1x1                  8  double
  b         1x1                  8  double
  x         2x1                 32  double    complex
  y         1x6                 48  double
```

or you can also get information about a specific variable:

```
>> length(y)

ans =

     6

>> size(id)

ans =

     4     4
```

You can save arrays to a delimited text file. For instance, saving *y* to a file named "y.dat"

```
>> save -ascii y.dat y
```

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

```
>> clear
```

Constructing diagonal matrices

Rather than tell MATLAB what each element of a matrix is individually, it is often useful when constructing matrices to use the `diag` command:

```
>> diag(ones(2,1),1)

ans =

     0     1     0
     0     0     1
     0     0     0
```

Note the second argument tells MATLAB which diagonal to use, relative to the main diagonal (which would be 0). Positive values go above the main diagonal, and negative integers go below, e.g.

```
>> diag(ones(2,1),-1)

ans =

     0     0     0
     1     0     0
     0     1     0
```

If you don't specify a second argument, the array will go on the main diagonal itself.

The eig function returns eigenvalues and eigenvectors:

```
>> [v,d]=eig(eye(2))

v =

     1     0
     0     1


d =

     1     0
     0     1
```

Note that the eigenvectors are entered into the columns of `v` and eigenvalues are the diagonal elements of matrix `d`. The `diag` command can be used to extract these values:

```
>> diag(d)

ans =

     1
     1
```

ARITHMETIC

Matrix operations are overloaded onto the scalar operations, for example: a row vector times a column vector is a scalar

```
>> size(rand(1,2)*rand(2,1))

ans =

     1     1
```

But a column vector times a row vector is a matrix

```
>> size(rand(2,1)*rand(1,2))

ans =

     2     2
```

However, it is often useful to operate on an array element-by-element. For instance, if you want to square every element, put a "." before the operator:

```
>> (1:3).^2

ans =

     1     4     9
```

Note that two row vectors cannot be multiplied together so this gives an error without the "."! Addition and subtraction do not have this problem since they are operations requiring arrays of identical size:

```
>> size(rand(2,1)+rand(2,1))

ans =

     2     1
```

When multiplying arrays element-by-element, use the ".*" operation. For instance, the probability density in QM is the squared norm of the wavefunction at every point Ψ*Ψ:

```
>> conj(Psi).*Psi
```

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. For example, consider a text file "script1.m" which 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. Also, notice that I used the exponential notation "1e2" to represent "100" in the above and that "`disp()`" displays the value of a variable.]

Now, just type the name of the script (without the ".m") at the command prompt to run it. :

```
>> script1
    0.4990
```

Make sure the .m file is in your current directory, or another part of your path!

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
```