Matlab

1) **Fundamentals**

a) **Getting Help**

for more detailed help on any topic, typing "help", then a space " ", and then the matlab command brings up a detailed page on the command or topic. For really difficult problems, use the "why" command.

b) **lower case**

always use lower case for **all** matlab commands. This is what matlab recognizes.

c) **silence**

to get Matlab to carry out a calculation, but not display the numbers on the screen, end any command or statement with a semicolon ";"

d) **Up Arrow**

The up arrow will scroll through the last commands that you entered. This can save you a lot of time typing, if you want to repeat a previous command, but with a slight modification. Using the up arrow key, you can bring up that previous command, and then using the left and right arrow keys you can make any needed changes. Hitting enter then executes the command.

2) **Arithmetic**

There are 2 different types of arithmetic commands in Matlab:
operators – these act on 2 numbers (or variables) and return 1 number (the answer)
example: 4*3
this is 4 times 3. The operator is the asterisk (*), and it is acting on the two numbers (4 and 3 in this case). If you type this in and hit return, Matlab supplies the answer.

All of the operators can be listed by typing "help ops". Some of the other major ones are:

+	addition
-	subtraction
/	division
^	raise to the power ("3^4" is 3 raised to the fourth power)

functions – these act on 1 number, and return 1 number
example: sin(1.57)
this is the sine of 1.57 (roughly π/2). Matlab returns the value of the sine of 1.57 if you enter this.

Other important basic functions are:

asin	inverse sine
cos	cosine
sqrt	square root
exp	exponential: exp(4) is the number **e** raised to the 4th power

log	natural logarithm
log10	base 10 logarithm
imag	imaginary part of complex number
real	real part of complex number
abs	absolute value
round	round to closest integer

the rest of the basic (or **el**ementary) functions can be listed by using "help **el**fun". For more detailed help on an individual function, type "help *functioname*"

more complicated functions include:

airy	airy function
bessel	Bessel functions
erf	error function
gamma	gamma function
legendre	associated legendre function

the rest of the **spec**ial functions can be listed by typing "help **spec**fun"

3) **Variables**

Variables are used to store individual numbers in matlab, and also to perform operations on the numbers. There are a couple key features about variables:

a) Naming

The name of any variable must start with a letter, and it may contain only letters, numbers, and the "_" character. The variable should not have the same name as an existing function. A good way to check this is to type "help *variablename*". If there is a function with name *variablename* then matlab will give you the help file, and you can choose something different.

For example, the following are valid variable names

a	b	hamil	a2	Kb	h_bar	g_bar

note: Matlab is case sensitive, so Kb, kb, KB, and kB are all different variables

b) "="

This is called the assignment operator (not to be confused with the logical equals operator, "=="). It assigns values to variables. The left side of this operator is always a variable, and the right side can either be a variable or a number. The statement "a = 3" would be read "a gets 3". Typing this statement would result in the variable "a" being created (if it didn't exist already), and the value 3 being assigned to it.

c) scientific notation

in order to assign variables really large or small numbers, without having to type in a lot of zeros, it is nice to use scientific notation. In Matlab, this is done by using the letter "e". For example, "2e5" means 2 x 10^5.
"Kb = 1.381e-23" would create the variable Kb, and assign the value of 1.381 x 10^{-23} to it.

d) types of variables

Matlab has some basic types of variables. The main ones are

integer	just like in math, these numbers have no decimals example: a = 1
double	short for double precision, which is in reference to the number of decimal places the computer will use. Example: a = 1.1451
complex	these are doubles which also have an imaginary component, or are entirely imaginary. Example: a = 1 + sqrt(-1)
string	these are variables that store text. Example: a = 'hello world'

e) Pre-assigned variables

there are some special variables in matlab that have already been assigned values. These variables can be over-written. The three most important ones are

pi	the number pi, out to the machine-limited number of decimal places
i and j	imaginary constant

f) "clear"

this command deletes all variables from memory. Use with caution, it is not reversible

4) **Vectors**

A vector in Matlab is almost identical to a vector in physics or mathematics. A position vector can be thought of as containing 3 variables: x,y,z. This is one way to think of vectors in Matlab – each entry in a vector is variable, that can be assigned any value. In this way, vectors in Matlab take on the same properties described above for variables. With some caveats, they also can be used with the same operators and functions.

a) **creating & assigning values**

The first way to create a vector in Matlab is to use brackets "[]". For example, to create a vector with:

2 entries	a = [3 1]
3 entries	a = [2 9 0]
4 entries	a = [7 3 10 0.1]

The space between numbers denotes/separates the individual entries. These commands create a row vectors. To create column vectors, you would use the complex conjugate transpose operator, or apostraphe '
a = [3 1]'

a = [2 9 0]'
a = [7 3 10 0.1]'

Sometimes it is necessary to access individual entries of a vector. This is done using parenthesis after the name. Inside the parenthesis you would put the integer value of the index you would like to access.
a(1) = 10

For doing calculations, it is useful to have a vector which corresponds to the x, or k, or whatever your independent variable is. To create a regularly spaced list of numbers in a vector, you would use the colon ":" operator. For example,

a = 1:10
is the same as
a = [1 2 3 4 5 6 7 8 9 10]

The two numbers indicate the first and last numbers of the series that Matlab will generate.

In the above statement, Matlab assumes you want a spacing of 1 between successive values. For other spacing values, you would use a statement like:
a = 1:0.1:10
equivalent to
a = [1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 …. 9.6 9.7 9.8 9.9 10.0]

Again, the first and last numbers in the "1:0.1:10" statement are the starting and end points respectively. The middle number, sandwiched between the colons, is the increment of the series. In order to count down, we have to explicitly tell matlab this by using:

a = 10:-0.1:0
The explicit definition of this command is "start at 10. add the number (-0.1) to 10 to create successive entries, until the number 0 is reached".

b) **Vector functions & operators**
   i) **functions**
   All of the **el**ementary functions work on vectors, to produce a vector of the same shape (if input is a row vector, output is a row vector). For example:

   x = 0:0.1:5
   Makes a vector which has incrementally increasing values from 0 to 5.

   y = sin(x)
   produces a vector which is the same shape as x (in this case a row vector) whose individual values correspond to the value of the sine of the individual values of the x vector. In other words,

x = [0 0.1 0.2 …. 4.7 4.8 4.9 5.0]
and
y = [sin(0) sin(0.1) sin(0.2) …. sin(4.7) sin(4.8) sin(4.9) sin(5.0)]
or
y = [0 0.0998 0.1987 -0.9999 -0.9962 -0.9825 -0.9589]

ii) **operators**
There are 2 types of operators which work on vectors. One set could be called vector operators, because they carry out the standard mathematical definition vector operations. The other set could be called array operators, because they treat the vector like an array, or list of numbers, and ignore linear algebra.

Important vector operators

+ and - addition and subtraction. These two operators require that the vectors are the same shape. Each individual entry is added together, producing a new vector whose individual entries are the sum of the original vectors individual entries
for example:
a = [1 3 10 7]
b = [-3 4 6 -2]
c = a + b
would yield
c = [-2 7 16 5]

* multiplication or **dot** product. The first vector must be a row vector, and the second vector must be column vector. The vectors must be the same length. This is just like the dot or inner product mathematically defined between two vectors. For example:
a = [1 0]
b = [0 1]**'** **note** the apostrophe which makes it column vector

a*b would yield a value of zero because:
a(1)*b(1) + a(2)*b(2) = 1*0 + 0*1 = 0

This is expected, because we know that the vectors a and b are perpendicular, and the dot product of two perpendicular vectors is 0.

Important array operators

.* **.**/ **.**^ array multiply, divide, and “raise to the power of”. Vectors must be the same shape. In this case each individual element acts on the corresponding individual element of the other vectors. For example:
a = [1 2 3 4]

```
b = [0  2  3   -1]

a.*b   would yield  [1*0   2*2    3*3   4*(-1)] = [0   4   9   -4]

a./b    would yield  [1/0   2/2   3/3   4/(-1)] = [inf   1   1   -4]

a.^b   would yield  [1^0   2^2   3^3    4^-1] = [1   4   27  0.25]
```

iii) **vector functions**

these functions apply to vectors, but not lone variables[1]. See "help elmat" and "help datafun"

size	returns the size of each dimension of a vector. Good for determining whether a vector is a row or column vector.
length	returns the number of entries in a vector. Extremely useful
max	returns the largest value of a vector
min	returns the smallest value of a vector
sort	sort in ascending order. Returns a vector of the same shape with the entries of the original sorted
sum	returns a number that is the sum of the entries of the vector

5) **Plotting**

The main command is plot. By supplying it with a list of values, it brings up a new window which contains a graph. For example:

```
x = 0:0.1:10
plot(x)
```

plots the values of the vector x on the y axis of the graph. The x axis is labeled with the indices of the vector x.

```
y = x.^2
plot(y)
```

plots the values of y on the graph, with the x axis the indices of the vector y.

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

plots the values of y on the graph, but this time the x axis values are taken from the x vector. x and y must be the same shape. The first point plotted has values x(1) and y(1), and so on.

To plot more than one curve on the same graph, use the "hold" command.

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

[1] They will work on lone variables, but the answer is somewhat trivial

```
hold on
z = sin(x).*(x.^2)
plot(x,z)
```

to clear a figure, use the “clf” command After executing a “clf” command, the figure is no longer “held”.

To bring up more than one figure, use the “figure” command. For example, “figure(2)” will create a second window in which the plot command can be used to make a graph. To go back to working on the first figure, use the command “figure(1)”. The “plot”, “hold on”, and “clf” commands all only work on the current active figure.

To add color to a plot, use a text string as the third argument of the plot command. For example:

```
figure(1)
clf
plot(x,y)
hold on
plot(x,z,'r')
```

produces the same plot as above, except this time the z curve is in **r**ed. For a complete list of plot features, type “help plot”.

6) **Examples of Potentially Useful Calculations**

The above list of commands may appear extremely basic. But they actually span quite a diverse set of mathematics, and used in conjunction with one another, they can be used to carry out a surprising number of complex calculations. Here is an example of a calculation that might be useful in 5.61, that can be done using the above commands.

To start: particle in a box. The box is length L = 10, from x = 0 to x = 10. The stationary state wavefunctions then are:

$$y = \left(\frac{2}{L}\right)^{\frac{1}{2}} \sin\left(\frac{n\pi}{L}x\right) \qquad \text{where n is a positive integer}$$

First, lets calculate & plot the first stationary wavefunction

```
clear
dx = 0.1
x = 0:dx:10;
y = sqrt(2/10)*sin(pi*x/10);
plot(x,y)
```

A quick check: is the wavefunction zero at the boundary?

A not so quick check: is the wavefunction normalized? Normalization constants can be tricky sometimes, and you can't just look at a plot of a wavefunction and tell if it is normalized. But, there is a quick way –

```
dx*y*(y')
```

This is a quick way to integrate the wavefunction. Let's take this apart.

1) The first operation will be the one inside the parenthesis. This produces the complex conjugate transpose of the vector y. y was originally a row vector, so now it is a column vector.
2) The next operation will be either of the multiplications. Let's look at y*(y'). y and y' have the complementary shapes, and the asterisk tells Matlab to take the dot product between 2 vectors. Since each entry of y corresponds to a value of the function, summing these up corresponds to taking the Riemann sum of the function (aka rectangular integration). Since what we're actually summing is y and its complex conjugate, what we're taking is the Riemann sum of $\psi^*\psi$, which is exactly what we want for the normalization
3) The last operation is multiply by the constant dx. This is required by the definition of the integral – this is the width of the individual rectangles that we're adding together

The number we get out should be 1.000. If it isn't we know we have to go back and check the wavefunction[2].

Now let's look at the second stationary state:

```
z = sqrt(2/10)*sin(2*pi*x/10);
figure(2)
plot(x,z)
```

Are the boundary conditions correct? Does it have the correct number of nodes? Is it normalized? Is it orthogonal to the first function? To answer this last question, we carry out our quick and dirty integration, but this time instead of $\psi_1^*\psi_1$ we want $\psi_2^*\psi_1$. In Matlab, this would be:

```
dx*z*(y')
```

If we've done everything correctly Matlab should return zero or a number pretty close (because of "machine roundoff error" it may be a number like $1\mathrm{x}10^{-10}$, which is essentially zero). If it isn't, then we know we've made an error and we can go back and check our equations.

At this point, in order to automate some of our tasks, it behooves us to look at….

[2] Or we can numerically normalize the wavefunction. To do this we would enter y = y/sqrt(dx*y*(y'));

9) Fundamental Programming Concepts

a) Script file

A script file is a file that you write, which is just a list of matlab commands. The commands are listed in the file, usually one on each line. The commands in the script file are identical to the ones you enter directly into matlab. To create a matlab script, type "edit". This brings up a new window which contains the matlab editor. Now you can list some commands, for example:

```
clear
dx = 0.1;
x = 0:0.1:10;
y = sqrt(2/10)*sin(pi*x/10);
figure(1)
clf
plot(x,y)
dx*y*(y')
```

you can save your script file by typing "control-s" or by going to the file menu (upper left corner) and clicking on save. You need to end your filename with a ".m" in order for matlab to recognize it. For example rwf.m is a valid filename.

Now, if we close all of our open figure windows, and type "rwf" in our matlab command window, we see that the program should execute and subsequently plot our first stationary wavefunction. The last line of the script causes matlab to display the normalization integral.

b) For Loop

The for loop is used to automate matlab commands, in order to avoid repetitive typing, and to get Matlab to do repeated calculations. Let's say we want to look at the first 10 or 20 stationary states. That would be a pretty arduous task if we had to carry out the calculation by typing in each of the scripts above, or entering the commands directly into the matlab window. Here's how we do this with a for loop in a script, by modifying our above script to look like:

```
clear
dx = 0.1;
x = 0:0.1:10;

for n = 1:10
        y = sqrt(2/10)*sin(n*pi*x/10);
        figure(1)
        plot(x,y)
        dx*y*(y')
        pause
end
```

we've also introduced the pause command. This just tells matlab to pause in its operations until we hit a key. When we save & execute this script, matlab will calculate each of the first ten stationary states, pausing between each one.

Taking apart the for loop: beginning with the "for" keyword, the next item is the variable which is used as our counting index, in this case the letter "n". At the end of the line we see the familiar statement "1:10", which we recognize as generating a vector containing the numbers 1 through 10, incremented by units of 1. The confusing part is the "=" sign in the middle. The **for** keyword tells matlab that n is not actually getting assigned the vector we create on the right ("1:10"). Instead, it tells matlab to set the value of n to each of the values in the vector created on the right, starting with the first index, and ending on the last. This means that we can go through any list of values we want – decimal increments, count down instead of up, or any variable we care to create.