A Matlab introduction/refresher

This set of notes should help you to either refresh your memory about how MATLAB works or give you an introduction to MATLAB. We will make extensive use of MATLAB this term, adding more advanced concepts and operations as we go along, so if you are struggling to remember the basics of MATLAB, you will be at a significant disadvantage.

Aside from this course, there are plenty of great (better?) sources of information about MATLAB on the web. You are encouraged to explore different online tutorials. One good place to start is here: http://uk.mathworks.com/support/learn-with-matlab-tutorials.html

Let’s get down to work

Launch MATLAB. You should see the interface appear on the screen. Look at each panel: what does it do? If you don’t need a panel, you can close it. This will declutter the MATLAB window.

The most important panel is the command window, in which you should see the MATLAB prompt:

```
>>
```

This is where you enter commands, using the keyboard. The double arrow indicates that MATLAB is waiting for input from the user.

Remember, however, that when you are working on a problem for class, it usually makes sense to use a MATLAB script. This is a set of commands that are listed in a file. The file should have a .m extension. You can then run the entire file on the MATLAB command line. Your script could, for example, contain the code for all the problem in a particular problem set.

Common sources or errors here are that the scripts may be saved in folders that are not in what is called the MATLAB search path. This may seem trivial, but to run the script, MATLAB needs to know where it is! You can add new folders to MATLAB’s path using the “set path” button. Alternatively, you can navigate to the folder containing the scripts in question.

While you can change working directory using your mouse, this can be done directly from within the command window, using “unix-like” commands, three of which are particularly useful: pwd, which prints the path of the current working directory, ls, which lists the files and directories available in the current working directory, and cd, which can be used to navigate between directories.

Getting help on Matlab Most people cannot remember all the details of all the commands in MATLAB. As a consequence, they rely heavily on the help command. To use help to learn about the clear command, type

```
>> help clear
```

Read the output! Another way to get help with MATLAB (or, more generally, in life) is to use Google (more info here http://lmgtfy.com/?q=matlab+help).

Creating a Matlab script Figure 1 shows an annotated MATLAB window pointing out the button to create a new script. The new script will be saved, by default, in the current working directory. You can change directories using the graphical interface. You might like to have a directory called SecondYearMaths in which you store your scripts for this class.

When you run a script, all the variables that were defined in your MATLAB workspace before you ran the script are available within the script itself. This can cause problems! It is highly recommended that you begin your script with
clear; close all;

This will give you a fresh workspace and ensure that all the necessary commands are contained within the script.

Before (or after) these commands, you should include a block of comment text explaining the contents of the script. Remember that comment lines begin with one or more % characters.

As you read the following notes, record all the MATLAB commands in your script. Run the script at any point to clarify concepts and answer questions in the notes.

Assignments  An assignment is when you put some data into a variable. The data is then stored by MATLAB in that variable, and the variable can be used to represent that data. For example,

```
>> A = 1
A =
    1
```

is an assignment. MATLAB now stores the number 1 in the variable A. We can check that it is stored there by saying

```
>> A
ans =
    1
```

Yep, still there. Good. We can use it in another assignment.

```
>> B = A + 10
B =
    11
```

Creating arrays  Arrays are structured groups of variables (usually numbers). They can be one dimensional:

```
>> a = [1 2 3 4]
a =
    1   2   3   4
```
Mathematical problem-solving for Earth sciences

>> b = [5; 6; 7; 8]
b =
   5
   6
   7
   8
They can also be two dimensional:
>> c = [1 2; 3 4]
c =
   1   2
   3   4
>> d = [a', b]
d =
   1   5
   2   6
   3   7
   4   8

In these examples we can see that use of a semi-colon ; between entries puts them on subsequent rows of the array. We also see, for the variable d, that we can concatenate two existing variables if their dimensions agree. In this case, a is 1 row by 4 columns, or 1 \times 4 and b is 4 columns by 1 row, or 4 \times 1. To make their dimensions agree, we take the transpose of a
>> a'
ans =
   1
   2
   3
   4

Note that the semi-colon has another effect. If you put it at the end of a line, it silences MATLAB's automatic output:
>> a = [1 2 3 4];

There are four common ways to create arrays in MATLAB. The first, shown above, is simply to list all the entries within square brackets []. To create larger arrays, this is impractical. Another approach is the colon :

>> e = 1:4
e =
   1   2   3   4
>> f = [5:2:11]'
   % note use of the transpose
f =
   5
   7
   9
  11
>> g = [10:2:16 ; 26:-2:20]
g =
   10   12   14   16
   26   24   22   20
The default increment of the colon operator is +1, but variables f and g show that we can
specify the increment to be whatever we like.

Another way to create arrays is with the \texttt{linspace()} function. Type \texttt{help linspace} for more information. Here's an example use
\begin{verbatim}
>> entries = 4*10 + 1;
>> h = linspace(1989, 1999, entries);
>> size(h)
ans =
   1 41
\end{verbatim}
In this example, we created a variable \texttt{entries} and set it to be 41. Then we created an array called \texttt{h} using \texttt{linspace()}. The first \textit{argument} to this function is the starting value of the array, the second argument is the ending value, the third argument is the number of entries. We then use the \texttt{size()} function to check that \texttt{h} has the expected number of entries. The result shows that it has 1 row and 41 columns. Each of these entries is equally spaced between 1989 and 1999.

Compare the contents of the following three arrays
\begin{verbatim}
>> tenth_increment_a = [1:0.1:2];
>> tenth_increment_b = linspace(1, 2, 10);
>> tenth_increment_c = linspace(1, 2, 11);
\end{verbatim}
The third common way is to use the functions \texttt{ones()} and \texttt{zeros()} to create an array. For example,
\begin{verbatim}
>> m = ones(2,3)
m =
 1 1 1
 1 1 1
\end{verbatim}
See the \texttt{help} entries for more information.

\textbf{Extract or modify entries of an array}  It is often useful to specify certain entries in an array. We might like to change one or more entries, or we'd like to create a new array from some entries in an old array. We do this by specifying the \textit{index} of the entry into an array. The index is an integer that identifies the row or column of the entry. Let's take a simple example using a variable created earlier.
\begin{verbatim}
>> a
a =
 1 2 3 4
>> a(1) = 10
a =
 10 2 3 4
>> a(2)=200
a =
 10 200 3 4
>> a(3:4)=[-3, -4]
a =
 10 200 -3 -4
\end{verbatim}
The index is specified within parentheses ( ) after the variable's name. We can specify a single

\footnote{The values that are passed into a function are called the arguments of the function.}
Index (e.g. \(a(2)\)), or we can specify a range of indices (e.g. \(a(3:4)\)). The use of indices to specify a part of an array is sometimes called subscripting (and the indices are called subscripts).

Indices can also be used on the right-hand side of an assignment:

```matlab
>> asub = a([1, 3, 4])
asub =
   10  -3  -4

>> asub(3) = 10*asub(3)
asub =
   10  -3  -40
```

Here’s a more complicated example. First we create the array \(A\) by concatenating four instances of the array \(a\):

```matlab
>> A = [a; a; a; a]
A =
   10  200  -3  -4
   10  200  -3  -4
   10  200  -3  -4
   10  200  -3  -4
```

Since \(A\) is a two-dimensional array, we need two indices to address its entries. The first index refers to the row; the second index refers to the column.

```matlab
>> A(:,3) = [1:4]'
A =
   10  200  1  -4
   10  200  2  -4
   10  200  3  -4
   10  200  4  -4
```

Here we have selected all entries of the third column of \(A\) and replaced them by the column-array \([1:4]'\). Now let’s replace a row,

```matlab
>> A(2,:) = b'
A =
   10  200  1  -4
   5   6   7   8
   10  200  3  -4
   10  200  4  -4
```

And a single entry,

```matlab
>> A(3,2) = 201
A =
   10  200  1  -4
   5   6   7   8
   10  201  3  -4
   10  200  4  -4
```

What we can see from the above examples is that when we wish to select an entire row or column of an array, we can use the naked `:` as the index.

**Doing arithmetic with arrays** It is often useful to add (+), subtract (-), multiply (*), or divide (/) two arrays of numbers (or a number and an array of numbers). By default, MATLAB treats arrays as vectors and matrices. For example:
because MATLAB did a vector-vector multiplication $1 \times 1 + 1 \times 1 + 1 \times 1 + 2 \times 1 = 5$. To tell MATLAB to do the multiplication \textit{element-wise}, we use the .\*$ multiplication operator

```matlab
>> [1 1 1 2].*[1 1 1 1]  % no transpose this time
ans =
     1   1   1   2
```

Try this line with a transpose ' on the second array to see what happens.

We might like to find the reciprocal of entries in an array. For example,

```matlab
>> n = [0:5];
>> k = 2*pi./n
k =
     Inf   6.2832   3.1416   2.0944   1.5708   1.2566
```

Notice that we use the ./ operator to ensure entry-wise division (also note that MATLAB correctly finds that $2\pi/0 = \infty$).

When doing addition or subtraction, MATLAB doesn’t require a dot before the operator because vector addition is already entry-wise.

```matlab
>> [1 2 3 4] + 5
ans =
     6   7   8   9
```

But one must be careful that the two objects to be added are compatible. Try entering each of the following

```matlab
>> [1 2 3 4] + [10 20]
>> [1 2 3 4] + [1 2 3 4]'
>> [1 2 3 4] + [1 2 3 4]
```

and note the results.

Raising an array to a power also requires you to specify whether MATLAB should work entry-wise. Recall that the exponentiation operator is the caret (^), and so the entry-wise exponentiation operator is the dot-caret (.^). For example,

```matlab
>> p = [2 4].^2
p =
     4   16
```

\textbf{Solving linear systems of equations} MATLAB stands for “Matrix Laboratory” and so it should come with no surprise that one of the most powerful and attractive aspect of MATLAB is its ability to solve linear systems of equations.

For example, consider the following equations:

\begin{align*}
x + y &= 1 \\
x \cdot 35 + y \cdot 2 &= 21
\end{align*}

This can be easily written in matrix-vector form:

```matlab
A=[1 1; 35 2]  \text{a matrix containing the coefficients;}
X=[x; y]  \text{a column vector containing the unknowns;}
```
The problem is then simply $AX = B$. In MATLAB, the solution of such systems can be easily obtained as

```matlab
>> X = A \ B
X =
0.57576
0.42424
```

The `backslash` operator is also called left division in MATLAB. What goes on internally in MATLAB when this operator is used is quite complicated. Suffice to say for now that the `backslash` operator of MATLAB is a powerful tool to solve linear systems and a whole class of problems known as “inverse problems”. To check that the above solution is indeed the solution, just compute the “forward problem”:

```matlab
>> A * X
ans =
    1
    21
```

**Basic plotting**  The plotting commands in MATLAB are extensive and powerful—what follows is a basic introduction, sufficient to get you started in this course. You will need to use `help plot` to learn more (and to find related commands).

Let’s first create a synthetic dataset

```matlab
>> t = linspace(0,2,201);
>> f = 10 + 5*sin(2*pi*t);
```

where $t$ represents time in days, the independent variable, and $f$ represents a dependent variable, say ground temperature in degrees Celcius. We could plot this as follows

```matlab
>> plot(f,'-k');
>> title('Ground temperature over two days.')
>> ylabel('temperature, degrees C')
>> xlabel('data index')
```

Run these commands in MATLAB. You’ll see that the plot it makes doesn’t have time as the independent variable. To fix this, first clear the figure

```matlab
>> clf
```
then run the following commands

```matlab
>> plot(t,f,'-k');
>> title('Ground temperature over two days.')
>> ylabel('temperature, degrees C')
>> xlabel('time, days')
```

So when you plot something, the first argument is the independent variable, the second argument is the dependent variable, and the other arguments are plot-type specifications. In this case we specified a solid line (`'-'`) that is black (`'k'`).

We can overlay things on our plot. First let’s overlay red circles for each quarter of the day.

```matlab
>> hold on; % this holds the previous plot
>> plot(t(1:25:end),f(1:25:end),'or','MarkerSize',10,'LineWidth',2);
```

Let’s specify the axis limits so we can see the full plot

```matlab
>> axis([0 2 0 20])
```

We could overlay another line. For example,

```matlab
>> plot(t,10+3*cos(2*pi*t),'--b','LineWidth',2);
```
Ground temperature over two days.

Figure 2: Plot created using Matlab.

Your plot should resemble that shown in Figure 2.

Plot properties can be changed on the command line, and of course in scripts, using the `get` and `set` commands. For example, you can increase the fontsize of the plot annotations using the following command:

```matlab
>> set(gca,'FontSize',30);
```

`gca` gets the handle to the current axis. To see all available options, simply type

```matlab
>> gca
```

in the command line and see what happens.

It is also a good idea to annotate plots with a legend. This can be done in simply with the command `legend`. As a start, try

```matlab
>> legend('sine curve','subsampled sine','cosine','Location','SouthWest')
```

Conditional execution: if-statements  “if-statements” are a way to do something only if something is true (or false). Judicious uses of “if-statements” can be used to save computational time. For example, if we need to do something one million times, but we only need to save the output every 1000 times, it would be quite wasteful to save all one million iterations and then delete the iterations that are not needed. “if-statements” work as follows:

```matlab
if condition
    do something here
elseif another condition
    do something else here
else
    if the previous clauses didn’t work, then do this
end
```

Here is a small example. Vary the value of `a` and make sure you understand how this works.

```matlab
a=-2;
if a>=0
    b=a
elseif a==0
```
b=1000
else
  b=a^2
end

Note that it can be useful and efficient to combine if-statements with so-called “logical operators”, such as & (AND), | (OR), ~ (NOT), == (EQUAL), etc... For example:

a=-2;
if (a<=0 & a>-4)
  b=a
else
  b=a^2
end

Repeating tasks: for-loops “for-loops” are used to tell the computer to automatically repeat a command a finite number of times. A related construct, called a “while-loop” works in a similar way, but instead of telling the number of times that the loop needs to iterate over, “while-loops” need to be given a condition to be met for the loop to end. Note that this can be tricky since errors in how conditions are coded can result in never ending loops.

“for-loops” have the following structure.

for variable=start:end
   do something here that depends on the variable
       ... and maybe do even more things ...
end

As a test, type the following on the command line:

for i=1:3
   i
end

Here, the value assigned to the variable i will change at each iteration. You will see the numbers 1, 2 and 3 printed out. Warning: in loops, unless you absolutely want to see the output of every iteration printed on the screen, don’t forget to add the ;. Keep in mind, though, that when troubleshooting one’s code, it can be very useful to run the loop a small number of times and to print the intermediate results of each iteration to the screen to check how/if the loop is working.

Loops can also be used for subscripting. For example, let’s imagine we are given a vector of integers and we need to add all even integers in that vector.

For the sake of illustration, let’s create a vector containing N entries, with each entry corresponding to a random integer between 1 and 1000.

```
>> N=100; % let’s say we want 100 numbers
>> items=randi([1,1000],1,N); % generate the vector of random integers;
>> eventotal=0; % initialize the grand total to 0

for i=1:length(items)
   test=mod(items(i),2);
   if(test==0)
      eventotal=eventotal+items(i);
   end
```
How much do you get?

**Input and output** Input/Output (I/O) can be quite complicated if the data format is not in MATLAB format. Since in science we tend to use data from multiple sources and these data are more often than not saved in different formats, there is not one single command that can be used to read all data types into MATLAB.

MATLAB developpers are doing a very good job at it, though, and there is a convenient “Import Data” button to click that will help you in most cases. If the data are in MATLAB format, they can be read simply using the `load` command.

MATLAB variables/arrays/results/etc can be saved to disk in MATLAB format using the `save` command. For example, let’s imagine we have two variables with results we need to save, a and b

```matlab
a=1:100;
b=201:300;
```

and we want to save them in a file called `importantvariables.mat`. The following command will do this:

```matlab
>> save('importantvariables.mat','a','b');
```

To test that this worked, clear the workspace and then try to read in the file.

```matlab
>> clear all
   >> load('importantvariables.mat'); Both a and b have been loaded. In some cases, for example when the file is very large or when we don’t need all the variables in the file, then it may be a good idea to only save the variables that are needed. This is achieved simply telling load which variables we want, e.g. a:
   >> load('importantvariables.mat','a');
```

**Writing and using functions** MATLAB scripts contain lines of code contained in an m-file that is recognisable by MATLAB. Typing the name of the file at the command prompt caused MATLAB to execute the code. While this is useful, the is not always the best way or organizing computer code.

In a script, the variables that you create are available in your workspace after the script has finished. In a function, the variables are all local to that function. They are created when the function runs, and destroyed when it completes. Like a mathematical function, a MATLAB function has arguments and outputs; these are the variables that go into and out of the function. The function has no knowledge of any outside variables besides those that are passed in as arguments. Furthermore, the arguments are copied into the function; changing the value of an argument within a function does NOT change its value outside the function.

Here’s an example:

```matlab
function mag = VectorRMS(v)
    % VectorRMS The root-mean-square magnitude of a vector.
    % argument: v -- a vector of real numbers
    % output: mag -- the RMS magnitude of the vector
    vsq = v.^2;
    mag = sqrt(mean(vsq));
```
This function takes a vector of real numbers as an input and finds their root-mean-square. An example of the usage of this function:

\[
\begin{align*}
\texttt{>> x} &= \texttt{[2:2:8];} \\
\texttt{>> M} &= \texttt{VectorRMS(x)} \\
\texttt{M} &= 5.4772
\end{align*}
\]

Although the function created and used the variable \texttt{vsq}, this variable is not present in the workspace after the function returns (in programming, “returns” means completes running and reports the output).

Some rules to be aware of when creating a function:

- A function must begin with the word \texttt{function}.
- A function must be stored in a file. The file and the function must have the same name. For example, the function \texttt{VectorRMS} must be stored in the file \texttt{VectorRMS.m}.
- Commented lines at the beginning of a function are printed as a \texttt{help} message.
- MATLAB only knows about your function if you are in the same directory as the .m file that contains it.

To get more information about writing new functions, type \texttt{help function} at the MATLAB command prompt.

Since it is slightly easier to use scripts than to write and use functions, beginners are often tempted to use scripts as “kitchen sinks” of computer code. That is, when people start coding, they will perform all tasks in a single script, including reading that data, manipulating the data, do calculations and plotting. While this approach will most likely work at the beginning, you will soon find that it is a very good idea to split the work into different units and to save calculations results. How the different units are separated will usually be subjective, but as a rule of thumb, try to write different functions for calculations and for plotting. This is so the plots can be redrawn, changed, etc. as many times as needed without having to recompute everything. Computations can take a long time so saving computational results and loading them in a separate function for plotting can actually result in substantial time savings!