Matlab tutorial for COMP61011

1 What is Matlab

Matlab is a software package which was developed for numerical analysis involving matrices (“matlab” as in matrix laboratory). It is now used for general technical computing. It is an interpreted language, which means that commands are run as they are entered in, without being compiled first. It also means that most of the commands, except the most basic ones, are defined in text files which are written in this interpreted language. So, you can read these files, and add to the language by making similar files. Many commands you will use have been created specifically for these labs.

Matlab is an imperative language and is like C in several respects. It has syntax similar to Java, but it is not object-oriented. The basic data element for Matlab is the matrix or array. This means that you can write programs which act on arrays easily and without the need for dimensioning and memory allocation. Because it is interpreted, it is slower than C, and to get the fastest performance, the matrix nature of Matlab must be used fully (the programs must be “vectorized”).

Matlab has built-in graphics and visualization tools. There are many add-on “toolboxes”. Matlab is an excellent prototyping language, because it has so many useful mathematics utilities, built-in. Many of the latest algorithms and research ideas in machine learning appear as Matlab packages before they are produced in other forms, such as C++.

When one get experience with the software, one can produce algorithms in matlab much more quickly than one could in JAVA, say. The downside is that these will run much more slowly than if they were written in C++, or even in JAVA. This is why Matlab is often used to prototype ideas. When the algorithms applied to large systems and need to run fast, they are often rewritten in a compiled language, such as C or C++.

2 Starting Matlab

Matlab runs on Linux machines in the computer science department. It also runs under Windows. If you are using Linux, do the following:

1. copy /opt/info/courses/COMP61011/startup.m to your working directory (the directory you from which you are going to do your lab work).
This file tells Matlab where to look for the special commands we have devised for the labs. When Matlab is invoked, it runs startup as its first command, if and only if startup.m is present in the current directory (from which Matlab was called). If you forget to move it to your current directory, you can always change to the directory and run startup from within Matlab.

2. The full path for matlab is (on Linux machines)

    /opt/matlab/bin/matlab

Typing this should start Matlab. You may put /opt/matlab/bin on your search path if you wish by adding the following to your .my_bashrc.all file,

    export MATLAB=/opt/matlab
    export PATH=$PATH:$MATLAB/bin

Then you will start matlab simply by typing

    matlab

You can also run it in the background by following matlab with &

Instead, if you are choosing to use Windows, the directory mentioned above is at L:/courses/COMP61011/ - navigate there and copy the startup.m as described above.

3 The Matlab Environment

3.1 The Matlab Environment

The Matlab environment consists of a window with three sub windows, as shown in Figure 1

The Command Window: This is the most important window. This is where the commands to Matlab will be entered. All of the commands described below are run by typing them at the prompt >> in the Command Window.

The Workspace Window: This shows the variables that currently exist in the workspace. Matlab is an interpreted language, and variables which you define and use remain until they are deleted or until Matlab is exited. You can click on the square icons to see their values.

Command History Window: This shows all of the commands you have ever entered. To rerun commands, you can cut and paste them from the Command History window to the Command window.
Figure 1: The Matlab Window. The Command Window is where you enter commands. The Workspace Window shows the variables defined in the workspace. The Command History window stores a history of the commands you have run. You can move this child windows around within the main Matlab window if you wish.
There is also a tab to a window which shows the files in the current working directory (the directory from which Matlab was invoked). You can remove all but the Command Window from the View menu on the menu bar.

**Exercise 1**

Start Matlab. If you want to see some of the things which can be done in Matlab, type `demos` at the prompt, and run some demos. Other Demos under menus Matlab/Demos is pretty good. Otherwise, carry on with the tutorial.

## 4 Getting Help

If you need more information about any Matlab function, there are several ways of getting it:

1. At the prompt, type `help` followed by the function name, e.g.

   ```matlab
   >> help sum
   ```

   (type `help` on its own to get a list of help topics.) N.B. Matlab online help entries use uppercase characters for the function and variable names to make them stand out from the rest of the text. When typing function names, however, always use the corresponding lowercase characters because Matlab is case sensitive and most function names are actually in lowercase.

2. To look for a command whose name you do not know, use "lookfor" (like `man -k` or `apropos` in UNIX)

   ```matlab
   >> lookfor sum
   ```

   This, however, can take a while.

3. The Help menu from the menu bar gives access to a huge range of documents and tutorials. Look under “MATLAB Help”. “Getting Started” which contains a good tutorial. “Using Matlab” is a useful reference.

## 5 Entering Commands and Command Line Editing

In Matlab, commands are entered at the prompt, and run when the return key is entered. For example, if you wanted to know the value of $2\pi$, you could enter

```matlab
x=2*pi
```

at the prompt (`pi` is a built-in constant). Matlab will give the answer,
and create a variable $x$ in the workspace (if it did not already exist), and set its value to the above. If you put a semicolon after the command,

```
x=2*pi;
```

the answer will not be printed on the screen, but the value of $x$ will be set. This is useful when you don’t need to see the value (because it is part of an intermediate calculation), or when the variable being set is a large structure which would take many pages to print out. If you don’t give a variable name to the calculation, the variable is stored in a variable called `ans`. E.g.,

```
2*pi
```

results in

```
ans =
6.2832
```

**Exercise 2**

Use Matlab to calculate 15 factorial (i.e. 15!). Use help to find the command and how to use it, then type it in to get the answer.

---

### 6 Matrices

One of the most important aspects of Matlab how it handles matrices. Whereas other programming languages work with numbers one at a time, Matlab allows you to work with entire matrices, which can take some getting used to. A matrix is a rectangular array of numbers, like a two-dimensional array in C or JAVA. An “n by m” matrix has n rows and m columns. Special meaning is sometimes attached to 1 by 1 matrices, which are called “scalars” (ordinary numbers, basically), and to matrices with only one row or only one column, which are called “vectors”.

#### 6.1 Entering Matrices

There are several ways to enter matrices in Matlab. These include:

- Entering an explicit list of elements.
- Loading matrices from external data files.
- Generating matrices using functions.

To enter a matrix explicitly, there are a few basic rules to follow:
• Separate the elements of a row with blanks or commas.
• Use a semicolon, ; or carriage returns, to indicate the end of each row.
• Surround the entire list of elements with square brackets, [ ].

For example, to input a $4 \times 4$ magic square enter:

```
>> A = [16 3 2 13; 5 10 11 8; 9 6 7 12; 4 15 14 1]
```

at the prompt. This describes the matrix

```
16 3 2 13
5 10 11 8
9 6 7 12
4 15 14 1
```

and assigns it to the variable ‘A’. If you don’t know what a magic square is, this will be explained later in this document (exercise 6). Be careful - Matlab is case-sensitive, so it distinguishes between ‘A’ and ‘a’.

Matlab will echo the matrix back at you, unless you put a semi-colon (;) at the end of the line. This is very useful when the matrices are very large.

This matrix can be referred to as $A$ until the end of the session, unless you decide to alter it in some way. When it encounters a variable it hasn’t seen before, Matlab automatically creates one. To see which variables have been created so far, look in the Workspace submenu. If you type $A$ (or any defined variable name) at the Matlab prompt, it will print back the name with its contents. This is useful if you want to check the contents of a variable.

There are commands for creating special matrices. These take parameters which define the size of matrix they generate. Some of these are:

- **zeros**: makes a matrix of all zeros (for initialization, for example). For example, 
  ```matlab
  zeros(2,3)
  ```
  makes a 2 by 3 matrix of zeros.

- **ones**: makes a matrix of all ones. Used like the zeros command.

- **rand**: makes a matrix of random numbers uniformly distributed between 0 and 1. E.g. 
  ```matlab
  rand(1,10)
  ```
  makes a column of random numbers.

- **randn**: as rand, except the numbers are normally distributed.

- **eye**: makes an identity matrix. E.g. 
  ```matlab
  eye(10)
  ```
  makes 10 by 10 matrix with 1’s on the diagonal and 0’s off the diagonal.

**Exercise 3** Enter the magic square $A$ (further up this page). Set $B$ to be a $4 \times 4$ matrix with 1’s on the diagonal and 0’s elsewhere. Check to see that both variables are present.
6.2 Accessing Elements of a Matrix

An element of a matrix can be referred to by using the format M(row,column). For example, A(3,2) is 6. So to calculate the sum of the top row of A, one could use

\[ A(1,1) + A(1,2) + A(1,3) + A(1,4) \]

(there simpler ways to do this, as shown in the next section).

A range of the array can be referred to by using the colon operator. M(i:j,k) refers to the rows i through j of the kth column. For example,

\[
A(2:4,3)
\]

yields,

\[
\begin{array}{c}
\text{ans} = 11 \\
7 \\
14
\end{array}
\]

The colon by itself refers to the entire row or column. For example, A(:,2) is the entire second column of A,

\[
A(:,2)
\]

yields,

\[
\begin{array}{c}
\text{ans} = \\
3 \\
10 \\
6 \\
15
\end{array}
\]

Thus, A(:,2) is equivalent to A(1:4,2).

Exercise 4 Get Matlab to compute the sum of the third column of A

6.3 More On The Colon Operator

The colon (:) is one of Matlab’s most important operators. It occurs in several different forms. The expression 1:10 is a row vector containing the integers from 1 to 10 (i.e. [1 2 3 4 5 6 7 8 9 10]).

To obtain non-unit spacing, specify an increment. For example:

\[
A(2:4,3)
\]

yields,

\[
\begin{array}{c}
\text{ans} = 11 \\
7 \\
14
\end{array}
\]

\[
A(:,2)
\]

yields,

\[
\begin{array}{c}
\text{ans} = \\
3 \\
10 \\
6 \\
15
\end{array}
\]

Thus, A(:,2) is equivalent to A(1:4,2).

Exercise 4 Get Matlab to compute the sum of the third column of A

6.3 More On The Colon Operator

The colon (:) is one of Matlab’s most important operators. It occurs in several different forms. The expression 1:10 is a row vector containing the integers from 1 to 10 (i.e. [1 2 3 4 5 6 7 8 9 10]).

To obtain non-unit spacing, specify an increment. For example:

\[
100:-7:50
\]

yields,

\[
\begin{array}{c}
\text{ans} = \\
100 \\
93 \\
86 \\
79 \\
72 \\
65 \\
58 \\
51
\end{array}
\]

\[
0:pi/4:pi
\]
ans =
  0  0.7854  1.5708  2.3562  3.1416

(pi is a built-in scalar constant).

Subscript expressions involving colons refer to portions of a matrix, as we have seen. So, \(A(1:3,4)\) means the same as \(A([1,2,3],4)\),

>> \(A(1:3,4)\)

ans =
  13  
   8  
  12  

>> \(A([1,2,3],4)\)

ans =
  13  
   8  
  12  

**Exercise 5** Generate a 4 by 2 matrix consisting of the every other column of \(A\) using the colon operator.

### 6.4 Functions on Matrices

Matlab has a number of built-in functions which can be performed on matrices. Here is a list of the more immediately useful ones.

**sum:** Returns the sum of the columns of a matrix, or, if used on a row vector, the sum of the row. For example,

>> \texttt{sum(A)}

sums the columns in the matrix, and returns the result as a 1 by 4 matrix. Notice how the special variable ‘ans’ is used to store the temporary result of the operation.

**mean:** Returns the mean (average) of the columns of a matrix.

**transpose:** To return the transpose of a matrix, append an apostrophe (or ”single-quote”) to the name. For example:
\[ A' \]
\[
\begin{array}{cccc}
16 & 5 & 9 & 4 \\
3 & 10 & 6 & 15 \\
2 & 11 & 7 & 14 \\
13 & 8 & 12 & 1 \\
\end{array}
\]

\[ sum(A')' \]
\[
\begin{array}{c}
34 \\
34 \\
34 \\
34 \\
\end{array}
\]

\textbf{diag:} Returns the diagonal of the matrix M.

\textbf{sqrt:} Returns the square root of the elements of matrix M.

\textbf{size} Returns the dimensions of the matrix M. This is returns a list of two values; the form is like this

\[ [\text{rows columns}] = \text{size}(A) \]

\begin{quote}
A magic square is a matrix in which the sum of each row, each column, and each diagonal is the same. The matrix A is a magic square. Check that for A, the sum of all rows and all columns are the same, using the \texttt{sum} command and the transpose operator.
\end{quote}

\section{7 Operators on Matrices}

You can add or subtract two matrices

\[ A + A \]
\[
\begin{array}{cccc}
32 & 6 & 4 & 26 \\
10 & 20 & 22 & 16 \\
18 & 12 & 14 & 24 \\
8 & 30 & 28 & 2 \\
\end{array}
\]

The result could easily have been stored in another variable, e.g.:

\[ S = A + A; \]
The multiplication operator *, division operator / and power operator ^ refer to matrix multiplication, division, and power respectively.

If a dot is put before these operators, the operator acts component by component. For example,

```plaintext
>> A.*A
```
returns the matrix containing the square of each component of A, whereas

```plaintext
>> A*A
```
performs matrix multiplication between the two. On scalars, both forms have the same meaning (which is the usual meaning).

Exercise 7: Since B is the identity matrix, multiplication of A by B should yield A. Check this. What does component by component multiplication give? Check this.

Logical operations are also allowed. These are the same as in most other languages: &, |, ~, xor have their usual meanings when applied to scalars. Any non-zero number represents True, and zero represents False. They can also be applied to matrices, and the result is a matrix of 0's and 1's. For example:

```plaintext
>> L = [0 0 1 1; 0 1 0 1];
>> L(3,:) = L(1,:) & L(2,:)
```

```plaintext
L =
0 0 1 1
0 1 0 1
0 0 0 1
```

```plaintext
>> L(3,:) = xor(L(1,:), L(2,:))
```

```plaintext
L =
0 0 1 1
0 1 0 1
0 1 1 0
```

Relation operators are similar to that of other languages: ==, <, >, <=, >=, ~=.

These return either 1 or 0, in much the same way as the logical operators:

```plaintext
>> 5<3
ans =
0
```

```plaintext
>> 4==2*2
```
However, these can be used with matrices as well:

```
>> A>10
```

```
ans =

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

Exercise 8 Use `sum` and logical operators to count the number of values of `A` which are greater than 10.

8 Graphics

8.1 Graphing functions

Matlab has range of built-in graphics and plotting routines. The command `plot(x,y)` makes a two-dimensional plot of `x` against `y`. For example,

```
x=-20:0.01:20;
y=sin(x)./x;
plot(x,y);
```

graphs the function sin function divided by `x` between $-20$ and $20$. (Try it.) The points of the `x` axis are separated by 0.01; for different spacing, you would use a different increment in the colon operator in the definition of `x`.

Type `help plot` at the prompt to get a description of the use of the plot function. Your math teacher may have taught you to always label your axes; here is how: `xlabel` and `ylabel` puts strings on the axes, like so,

```
xlabel('x');
ylabel('sin(x)/x');
```

To get rid of the figure, type `close` at the Matlab prompt.

There are a load of options to `plot`, which `help plot` will show. It is possible to control whether the plots are lines or points, the line style, color, and other properties of the plots. The basic form is `plot(x,y,str)` where `str` is a string of one to three characters denoting color, symbol plotted at each point (if any) and line type (if any). Here is a table of options,
COLOR POINT STYLE LINE STYLE
character color character symbol character line style
b blue . point - solid
r red x x-mark -. dashdot
c cyan + plus – dashed
k black d diamond
y yellow s square
For example, if we wanted to plot the graphs above as crosses, and in red, the command would be

plot(x,y,’r+’);

This is useful for plotting data. For example, suppose we had some data arranged in columns.

data =

5.1000 3.5000
4.9000 3.0000
4.7000 3.2000
4.6000 3.1000
5.0000 3.6000
5.4000 3.9000
4.6000 3.4000
5.0000 3.4000
4.4000 2.9000
4.9000 3.1000

We could plot it using the command, plot(data(:,1),data(:,2),’+’).

Exercise 9 Plot the log of the integers from 1 to 100.

8.2 Multiple Plots

If you want to compare plots of two different functions, calling plot twice in succession will not be satisfactory, because the second plot will overwrite the first. You can ensure a new plot window for a plot by calling figure first.

If you want to put multiple plots on the same figure, we set hold on The default is hold off, which makes a new figure overwrite the current one.
Here is an example. Suppose we want to compare the log function with the square root function graphically. We can put them on the same plot. By default, both plots will appear in blue, so we will not know which is which. We could make them different colors using options. Here is the final answer,

\[
x=1:100; \\
y=log(x); \\
z=sqrt(x); \\
plot(x,y,'r'); % plot log in red \\
hold on; \\
plot(x,z,'b'); % plot log in blue \\
hold off;
\]

What appears after the \% on a line is a comment. The options can also be used to plot the values as points rather than lines. For example, ‘+’ plots a cross at each point, ‘*’ a star and so forth. So,

\[
x=1:100; \\
y=log(x); \\
z=sqrt(x); \\
plot(x,y,'r+'); % plot log as red crosses \\
hold on; \\
plot(x,z,'b*'); % plot log as blue stars \\
hold off;
\]

### 8.3 Three-Dimensional Plots

These will not be used for this course, but you can also make three dimensional plots. If you have three dimensional data, such as

\[
data = \\
\begin{array}{ccc}
5.1000 & 3.5000 & 1.4000 \\
4.9000 & 3.0000 & 1.4000 \\
4.7000 & 3.2000 & 1.3000 \\
4.6000 & 3.1000 & 1.5000 \\
5.0000 & 3.6000 & 1.4000 \\
5.4000 & 3.9000 & 1.7000 \\
4.6000 & 3.4000 & 1.4000 \\
5.0000 & 3.4000 & 1.5000
\end{array}
\]

This could be plotted as points in 3-d, using the \texttt{plot3} command,

\[
plot3(data(:,1),data(:,2),data(:,3),'+')
\]

You can rotate the plot around.

You can also plot functions of two variables. An example of this can be seen with the following
[x,y] = meshgrid(-8:.5:8);
r = sqrt(x.^2+y.^2) + eps;
z = sin(r)./r;
surf(x,y,z);

You can click on the icon which looks like a circular arrow, and rotate the plot around using the mouse. To learn about using Matlab to make all types of plots, see “Graphics” under the “Getting Starting” and under the “Using Matlab” in the on-line help. For a list of commands, type help graph2d or graph3d at the Matlab prompt.

9 Control Structures

There are two important control structures for these labs.

9.1 If

The form for if-else constructs is

if condition
    statements
else
    more statements
end

where the else part is optional. Likewise, there is an elseif keyword,

if condition1
    statements
elseif condition2
    more statements
elseif condition3
    even more statements
    ...
else
    yet more statements
end

9.2 For

There are loop structures using for. The basic construction is,

for index = j:k
    statements
end

As an example, lets sum the main diagonal of A:
```matlab
>> c=0;
>> for i=1:4
    c=c+A(i,i);
end;
>> c

34
```

Of course, this is more easily done using `sum(diag(A))`.

**Exercise 10** Use a `for` loop to compute the sum of the columns containing the reciprocal of each element of `A`. Can you do this using matrix commands, without a `for` loop?

### 10 Running Commands From a File

#### 10.1 Scripts

You can type a set of commands into a file and read that file into Matlab. Matlab will run these just as if you had typed them. Such files are called scripts. Matlab looks for files in directories defined by a `path` variable. The directory from which Matlab was invoked is in the path. To add a directory to the path, use

```
addpath directorypath .
```

#### 10.2 User-defined functions

You can create your own functions for use in Matlab. These are defined in separate files, somewhere on your search path (usually they will be in your current directory). The best way to show this is for you to try it yourself: in a text editor or in the Matlab Editor, make a file in your current directory, calling it `test.m`, with the following text in it:

```matlab
function result = test(m)
    \% A test function for Matlab
    result = sum(diag(m))
```

Save it, and then in Matlab type

```
>> test(A)
```

The returned value (here stored in ‘ans’) will be the sum of the diagonal of `A`. Here’s a brief explanation of the file you typed in:
• ‘function’ tells Matlab that this is a function file

• ‘result = test(m)’ means that the function is called ‘test’, and will accept
  one input parameter and one output parameter, with the specified names.
  The file must have the same name as the function and the extension “.m”.

• Anything between a % and the end of the line is treated as a comment.

• ‘result = sum(diag(m))’ does the actual calculation. The answer is stored
  in ‘result’, and as there are no more lines of code, the function ends. The
  return parameter is the last value that ‘result’ had.

Matlab contains a built-in editor which you can invoke with the command
edit or by using Open or New from File on the menu bar. You can also use
textedit, emacs, or whatever other UNIX editor you are familiar with.

Here is an example to test whether the sum of the diagonal from top left to
bottom right of a matrix is the same as that from top right to bottom left.

function result = diagonalSum(m)
% DIAGONALSUM returns 1 if sum if left to right diagonal of a square
% matrix is the same as that of the right to left diagonal.
[height width]=size(m);
if (width ~=height)
    error('Only works for square matrices'); % print error message and
        % terminate
end
lefright=0;
rightleft=0;
for i=1:width
    lefright=lefright+m(i,i);
    rightleft=rightleft+m(i,width-i+1);
end
result=(lefright==rightleft)

10.3 Vectorization

Built-in matrix commands are compiled (along with the rest of Matlab) and
thereby run faster. Thus, commands will be faster if they are written as built-in
matrix operations. For example, the function diagonalSum above would run
faster if the loop is replaced with a matrix command,

function result = diagonalSum(m)
% DIAGONALSUM returns 1 if sum if left to right diagonal of a square
% matrix is the same as that of the right to left diagonal.
[height width]=size(m);
if (width ~=height)
error('Only works for square matrices'); % print error message and
% terminate

end

lefrtright= sum(diag(m));
rightleft=sum(diag(m(:,width:-1:1))); % sum of diagonal of column
% reversed matrix

result=(lefrtright==rightleft)

This is called “vectorization”. It is not always so easy to see how to do it. You
need only worry about this if speed of your programs becomes an issue. If it
does, you may like the profiler: do help profile.