This tutorial was written by Jonas Larsson, Royal Holloway University of London
jonas.larsson@rhul.ac.uk

Matlab Tutorial 1

Learning outcomes

By the end of this exercise, you will:

- know how to start and exit Matlab
- be familiar with the Matlab graphical user interface
- be able to navigate through folders and files using the graphical user interface and the command line
- have run a demo program demonstrating using Matlab to run and analyse a brief psychological experiment
- understand how to enter commands using the command line and be familiar with basic commands for listing files and variables and getting help
- understand the basic numeric variable types in Matlab (scalars, vectors, matrices, and arrays), how to enter variables and inspect variable values
- know how to load and save data into Matlab
- know how to do basic plotting in Matlab and how to save plots as images
- understand how to perform basic mathematical operations (adding, subtracting, computing mean and standard deviation etc) on variables in Matlab
- understand what scripts and functions are and how to call them from the command line
- optionally, have written your first program in Matlab and come across fundamental concepts of Matlab programming
- optionally, have read through a real Matlab program and understood what it does

Before you start
The aim of the tutorial is to give you hands-on experience, but it is important that you ask the instructor to explain anything that is unclear. At the end of the session, there will be a summary of what has been learnt. If you do not have time to complete all sections, you are encouraged to do so in your own time after the class – even if you finish in time repeating the exercise is a good way of consolidating what you have learnt.

Using Octave instead of Matlab
Octave is an open-source software package that is very similar in functionality to Matlab, but is free. Most of the commands are the same, but the user interface looks a bit different. If you don't have access to (or can afford) Matlab or want to try using open-source (and free!) software instead, much of this tutorial works on Octave as well – try it if you like but beware that some things might not work as expected (in particular, the user interface is more like a traditional programme with file menus etc – however it has the same functionality as Matlab's).

Connecting to the tutorial drive and copying files
The files for this course are located on Moodle. You need to copy the files to your home directory (Y:) in order to save any data and modify the files. To copy the files, do the following:

- Login to Moodle, and navigate to the course page
- Click on the MatlabTutorial.zip file and save it to your computer
- When it has downloaded, it should open in Windows; extract the files into your home drive

(Y:)

You are now ready to start the tutorial.

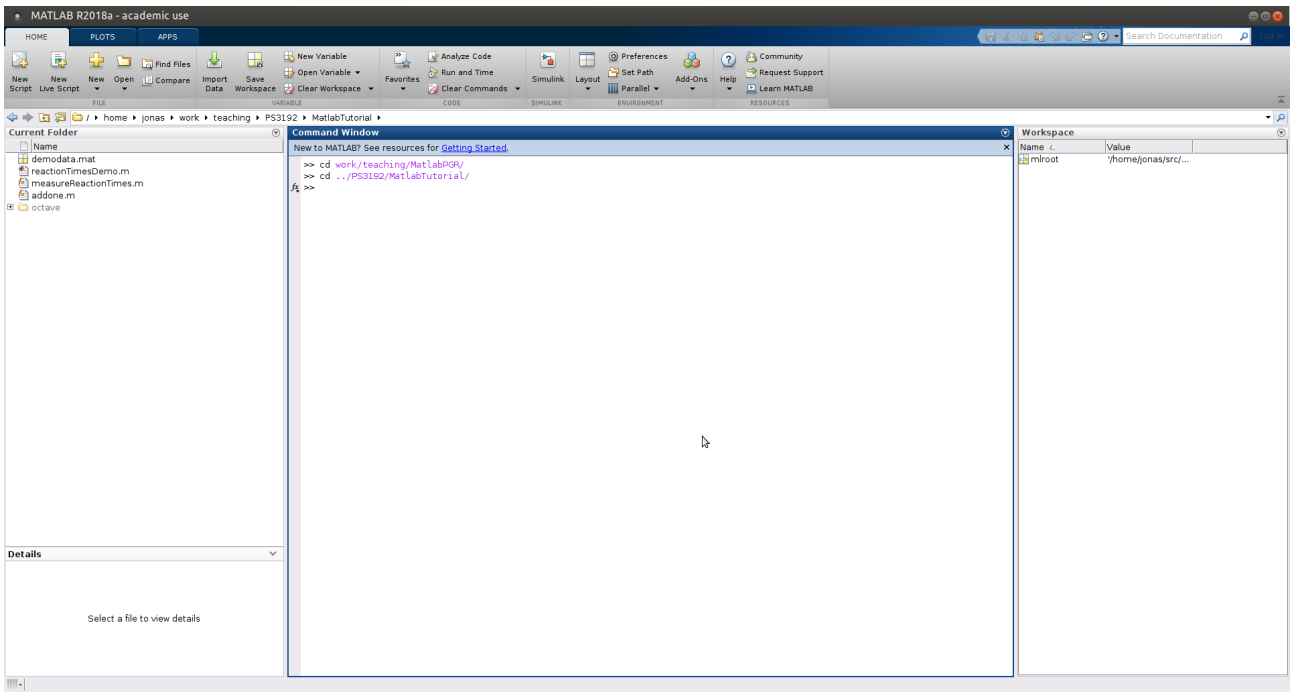
1. Starting and exiting Matlab
From the Windows menu bar, find the Matlab application folder and click on the Matlab program (it might be called something like “Matlab R2019a”). Matlab will now launch (it may take a little while).

To exit Matlab, either choose File->Exit, or type in

```
exit
```

at the command line (see section 4).

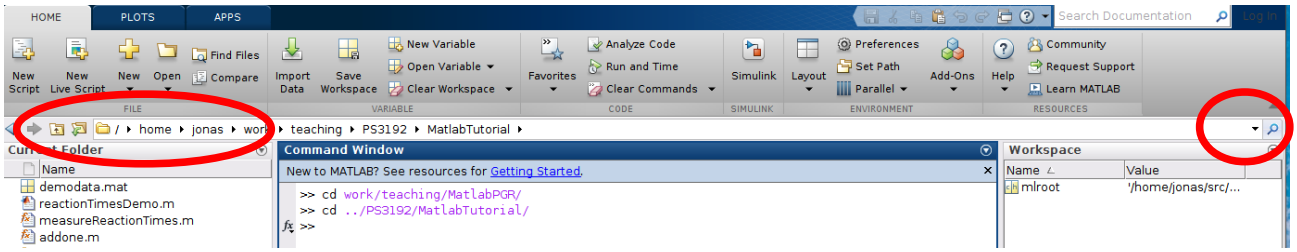
2. Overview of Matlab desktop
When Matlab opens, it will display a main window with several panels that look something like this – this is called the 'Matlab Desktop'. [Note that the exact look and feel of Matlab will depend on the version you use, but the basic elements described below are in all versions – ask the instructor if you are unsure.]



The middle panel is the 'command window'. This is where the user communicates with the program, by entering commands on the 'command line', which will be covered in section 4. The top left-hand panel shows the files and folders in the current folder (directory). The right-hand panel shows the *variables* in the current *workspace* (section 5). In the top menu, you can choose additional windows to display, for example a text editor to write scripts (section 9).

3. Navigating folders and files
Like most programs Matlab has the concept of a working directory or current folder, which is the directory (folder) where files are saved or loaded by default. When you start Matlab, the default folder will probably be your home folder. You can change the current folder in the menu at the top

of the main window:



Click on the drop-down menu arrow to the right to select your home drive Y:. Then click on the "Browse for folder" button (a folder with a green arrow) in the left upper window pane. This should open a file browser that list the files in your Y: drive. Navigate to where you downloaded the turorial data – it should be a folder called MatlabTutorial – double click on this to change the current folder to that folder and you will see a list of the files in that folder in the window pane below.

You can also change folder from the command line – see next section.

4. Using the command line

Unlike most conventional programs you may have used (like Word, Excel, etc), where the user interacts with the program by pointing the mouse and clicking, in Matlab users mainly interact with the program by typing in commands into the command window. While this can seem somewhat primitive, clunky and difficult at first, with a little practice it allows doing very complicated tasks quickly and much more easily than using a "point-and-click" interface - and importantly, it allows scripting (automating) any task easily (we will cover that in section 9).

When you start Matlab, in the command window you will see a startup message followed by the "command prompt" which might look something like this:

```
      This is a Classroom License for instructional use only.
      Research and commercial use is prohibited.
>>
```

The command prompt '>>' is where you communicate with Matlab. If you click the mouse inside the command window and then start typing, the text you enter will appear to the right of the command prompt. When you hit the Return (or Enter) key, whatever text you have typed will be interpreted by Matlab as a command. For example, if you type

```
dir
```

at the command prompt and hit return, Matlab will print a list of the files in the current directory, which might look like this:

```
.                          demodata.mat
..                         measureReactionTimes.m
plotReactionTimesData.m    reactionTimesDemo.m
```

In the following, we will use the font `Courier in normal type` to indicate text that Matlab

outputs in the command window, and **`Courier in bold type`** to indicate text and commands that **you** should enter using the keyboard. (We will not write out the command prompt in this manual, to make it easier for you to copy and paste commands from the manual into Matlab. A warning though: especially on Windows, copying and pasting text sometimes introduces invisible characters into the copied text that mess up the results; if things seem weird type in the commands instead.) For example, if you type the 'help' command, Matlab will list the main help categories as follows:

`help`

```
HELP topics:

matlab/general       -  General purpose commands.
matlab/ops           -  Operators and special characters.
matlab/lang          -  Language constructs and debugging.
matlab/elmat         -  Elementary matrices and matrix
manipulation.
matlab/specmat       -  Specialized matrices.
matlab/elfun         -  Elementary math functions.
matlab/specfun       -  Specialized math functions.
matlab/matfun        -  Matrix functions - numerical linear
algebra.
matlab/datafun       -  Data analysis and Fourier transform
functions.
matlab/polyfun       -  Polynomial and interpolation functions.
matlab/funfun        -  Function functions - nonlinear numerical
methods.
matlab/sparfun       -  Sparse matrix functions.
matlab/plotxy        -  Two dimensional graphics.
matlab/plotxyz       -  Three dimensional graphics.
matlab/graphics      -  General purpose graphics functions.
matlab/color         -  Color control and lighting model
functions.
matlab/sounds        -  Sound processing functions.
matlab/strfun        -  Character string functions.
matlab/iofun         -  Low-level file I/O functions.
matlab/demos         -  The MATLAB Expo and other demonstrations.
```

If you have any toolboxes installed (which are additional Matlab software) then those will be listed as well. If you type ‘help’ followed by a command name, Matlab will print a brief help text about how to use that command – try for example to type

`help dir`

You can also access Matlab’s documentation which is more detailed by typing ‘doc’ and the command name, e.g.:

`doc dir`

Matlab remembers the most recent commands entered (your command history). If you want to repeat a previous command, rather than typing it again, you can just enter the first few letters and

Matlab will display a pop-up window with the list of matching commands in your command history, You can scroll through this list using the up and down arrow keys, and then hit enter when the right command is selected – this is the same as typing it in again and saves a lot of time.

If you type in a number, Matlab will print that number:

```
4.56

ans =
     4.5600
```

`ans` means the result of most recent command (in this case the number you entered).

As mentioned above, you can change current folder from the command line (useful for example for scripting). To change to a directory inside the current folder, type **`cd`** (short for **c**hange **d**irectory) and the name of folder. So for example, suppose you had a folder called 'TopSecret' in your current folder, you would type

```
cd TopSecret
```

to change the current folder to 'TopSecret'. (What happens if you try this? Why?)

Note that Matlab is case-sensitive – you need to enter commands and directories using the right case (meaning typing **`CD topsecret`** would give an error message). To go back up one folder, you type **`cd ..`** (note the space after cd):

```
cd ..
```

The two dots .. mean “up one directory”. You can find out which is the current folder by typing **`pwd`** (short for **p**rint **w**orking **d**irectory):

```
pwd
```

To change to a directory not immediately above or below the current folder, you need to specify the full path. The path is the location of the directory on the computer, starting (on Windows) with the drive letter (e.g., Y:) followed by the directories separated by backslash (\). For example,

```
cd Y:\MatlabTutorial
```

will change the current folder to the MatlabTutorial folder in the Y: directory regardless of the current folder. NOTE: On Mac and Linux, which don’t suffer from the arcane concept of drive letters, a forward slash (/) is used to separate directories, e.g.

```
cd /home/myhomedir/MatlabTutorial
```

You might have noticed that the name of Matlab commands tend to reflect what they do, in a minimalistic sort of way. The terseness (**`cd`** instead of changedirectory for example) makes it easier to enter commands quickly – obviously once you have learnt them!

You now know enough of Matlab to run the demo program. Change directory to the MatlabTutorial

directory that you copied to your home directory – either using the menu on top or using the **cd** command. Then type

```
reactionTimesDemo
```

to run the demo, which runs a psychological test called a Stroop task. Follow the instructions – it will take about 10 min. You may need to adjust the volume on your PC for optimal results. Note that the script assumes you are two players - you can run this in groups of two if you like, or run yourself twice.

The demo should open a figure with five plots, showing:

- mean and standard deviation of reaction times for each player for the catch and non-catch trials (catch trials were those where the word GO could appear both in green and red and you were only supposed to respond to green GO; non-catch trials were those where GO always appeared in green). If there was a significant difference between catch and non-catch trials, an asterisk would be plotted above the non-catch trial bar. Was the difference significant?
- average reaction times for all players and conditions. Which player was fastest?
- individual reaction times for all correct trials. Do reaction times get shorter over time?
- the proportion of errors for each player on catch trials. Which player had the fewest errors?

The demo shows how Matlab can be used to deliver visual and auditory stimuli; measure reaction times; compute various statistics; plot the results; and save the data to a file. Later, once you have learn a few Matlab basics, you can open the demo program and inspect the code to see how each of these steps are done.

5. Introduction to Matlab variables

The core function of Matlab is to carry out calculations or manipulations of data provided by the user. Matlab can store these data in different types, the most important being numeric (numbers) and string (text) data. (There are also more complicated data types like structures and cell arrays, which we will cover later.) Data are stored in *variables*. Variables are symbols that can have almost any name (such as x, Y, myVariable, rt2, subject14...) and are assigned a value. For example, to assign the value 5 to the variable x, you would type:

```
x=5
```

```
x =
     5
```

Note that Matlab by default prints out the variable and its value on the command line – you can avoid this by adding a semicolon ';' after the command (the variable will still be saved):

```
x=5;
```

If you type the name of the variable, Matlab will print its value:

```
x
```

```
x =
     5
```

Although we will focus on numeric variables in this tutorial, Matlab can also store strings (text) – these are entered by typing in text within single quotes (').

```
mystring='This is a text, or string, variable'

mystring =
     'This is a text, or string, variable'
```

Numeric variables can contain a single value (a *scalar*) or multiple values in a row or column (a row or column *vector*, respectively). To enter several numbers into one variable, the numbers need to be enclosed in square brackets [] and separated by spaces or commas:

```
x=[1 2 3 4 5]

x =
     1  2  3  4  5

x=[1,2,3,4,5]

x =
     1  2  3  4  5
```

The variable x above is a *row vector* containing five numbers.
If numbers are separated by semicolons (;) instead of spaces or commas, they will be interpreted as being on different rows of one column, creating a *column vector*:

```
x=[1;2;3;4;5]

x =
     1
     2
     3
     4
     5
```

Now x is a column vector containing five numbers.
You can change a row vector to a column vector (and vice versa) by *transposing* it by adding a single quote (') after the variable name:

```
x=x'

x =
     1  2  3  4  5
```

You can create vectors from other vectors by concatenating them, as long as they are the same type (a row vector can be concatenated with a row vector but not a column vector, and vice versa) – this simply appends one vector to another:

```
x=[1 2 3 4 5];
y=[8 7 6 5];
```

```
xy=[x y]
```

produces the same result as

```
xy=[1 2 3 4 5 8 7 6 5]
```

To concatenate two column vectors, you need to use a semicolon instead of a space or comma:

```
w=[1;2;3;4;5];
z=[8;7;6;5];

wz=[w;z]
```

It is even posssible to have variables containing several rows of numbers in an array - such a variable is called a *matrix*:

```
m=[1 2 3;4 5 6;7 8 9]

m =
     1 2 3
     4 5 6
     7 8 9
```

You can also transpose a matrix – this exchanges rows and columns:

```
m=m'
```

Matrices can also be concatenated as long as their sizes match relative to the dimension you are concatenating them – so if you want to concatenate two matrices left-to-right, they need to have the same number of rows, and if you are concatenating two matrices top-to-bottom then they must have the same number of columns:

```
mx=[4 11 9 2; 8 6 15 0]
my=[3 7; 9 5]

mxmy=[mx my]

mxmy =

     4 11  9  2  3 7
     8  6 15  0  9 5
```

but concatenating them vertically using a semicolon doesn't work as mx and my have different numbers of columns:

```
mxmy=[mx;my]

Error using vertcat
Dimensions of arrays being concatenated are not consistent.
```

Matrices are very common in mathematics and numerical analysis, in fact the name 'Matlab' is a contraction of **Mat**rix **Lab**oratory. An intuitive example of a 'real-world' matrix is a digital grayscale image (such as a black-and-white photo) – each number in the matrix represents the lightness of the photo at the corresponding location in the photo. For example, if the top lefthand corner of the photo was black, this could be represented in a matrix by setting the number in the top row and leftmost column to 0 (meaning no light).

Both vectors and matrices are examples of *arrays*. Vectors can also be thought of as 1-dimensional arrays (having the single dimension of length) whereas matrices are 2-dimensional (having the two dimensions width, or number of columns, and height, or number of rows). In fact Matlab can be used to store arrays with more than 2 dimensions - 3, 4 or many more dimensions are possible. A 3-dimensional array, for example, could be used to store a number for each point in a volume (this is how MRI images are stored in Matlab).

Vectors are commonly used to store experimental variables. For example, suppose we have measured ten reaction times each for two volunteers - we could then store the reaction times in two variables that we call RTc1 and RTc2, each of which is a row vector 10 rows long. In fact, the demo program will have created two variables with these names, which you can inspect by typing their names:

```
RTc1
RTc2
```

Which subject was faster – player 1 or 2? You may want to compare RTc1 and RTc2 (the reaction times for the catch trials, in which you were only supposed to press a key when GO appeared in green but not in red) with RTnc1 and RTnc2 (the corresponding reaction times for the non-catch trials when GO was always shown in green). Was there a difference?

Individual numbers in an array are also referred to as *elements* of the array. You can access individual numbers, or elements, in a vector by *indexing*. For example, to display the second number in RTc1, type

```
RTc1(2)
```

This means 'show the second element (or number) in vector RTc1' – the number 2 is the *index* of that element. You can use indexing to look at several elements at the same time, for example to list the 1st, 3rd , and 7th element of RTc2:

```
RTc2([1 3 7])
```

Notice that the list of indexes [1 3 7] is itself a vector – so you can index a vector with a vector (so long as the indexes are integers – RTc1([1.2 4.5 6.7]) won't work – why?)

Matlab provides shortcuts that can be handy if you want to look at a smaller part of a vector – for example say you want to look at the 3rd to the 9th elements of RTc2. You could type

```
RTc2([3 4 5 6 7 8 9])
```

but it is much easier to use 'colon notation':

```
RTc2(3:9)
```

which gives the same result. The colon (:) here means 'create a sequence of numbers between 3 and 9 increasing by 1'. (Note you don't need to use the square brackets when using colon notation.) If you use a colon on its own as index, Matla will print out the entire vector or matrix as a column vector (this is a quick way to convert a matrix into a vector for instance):

```
RTc2(:)
```

You can modify parts of a vector using indexing, e.g. to set the 4th number of RTc1 to 0:

```
RTc1(4)=0
```

will print out the values of RTc1 with the fourth element set to 0. You can also modify all of the elements in a vector or matrix by using the colon operator:

```
RTc1(:)=0
```

means “set all elements of RTc1 to zero”. The colon is important – otherwise you would just set the variable to be a single zero – test this by checking what happens if you type (but beware this will overwrite the values in RTc1!)

```
RTc1=0
```

Accessing elements of matrices and arrays of more dimensions is done the same way as for vectors, but with additional indexes for the extra dimensions, separated by commas. For example, let's define a matrix M with 50 rows and 50 columns filled with random numbers between 0 and 1, using the command **rand**:

```
M=rand(50,50);
```

To show the number on the 23rd row and 15th column of M you would type

```
M(23,15)
```

Likewise, to access elements of a 3D array you would use 3 indexes separated by commas. First create a 10 x 20 x 30 3D array by the following command:

```
my3Dimage=rand(10,20,30);
```

then to access one of the elements type:

```
my3Dimage(4,15,24)
```

This would display the element on the 4th row, 15th column, and 24th slice of the 3D array called 'my3Dimage' . If you want to look at one “slice” of the 3D array you can use the colon operator (:) to show everything in a row and/or column:

```
my3Dimage(:,:,24)
```

means “display all elements in all rows and all columns of slice 24”.

You can create arrays and matrices of any size filled with zeros or ones using the commands **zeros** or **ones**:

```
myzeroarray=zeros(4,5) % creates a 4x5 matrix filled with zeros
myonearray=ones(2,4,6) % creates a 2x4x6 array filled with ones
```

Another way to create an array is to use the **repmat** command, which replicates a value in each dimension that you request as follows:

```
myfivearray=repmat(5, 4, 7, 9, 2);
```

creates an array that is 4 rows by 7 columns by 9 slices by 2 ‘hyperslices’ in which each element has the value 5.

Often it is easier to look at a matrix as an image where colours represent the value of each element. You can use the command **imagesc** for this. To look at the slice above you can type

```
imagesc(my3Dimage(:,:,24))
```

You can add a colour bar to give a numeric scale to the colours used by typing (note US spelling)

```
colorbar
```

To remove the colour bar, type

```
colorbar off
```

The colours used to display data is called a *color map*. This is simply a matrix of colour values from low to high, normally 64 rows long. Each colour is a row vector 3 elements long specifying the red, green, and blue values of each colour (as numbers between 0 and 1). So red would be [1 0 0], green [0 1 0] and blue [0 0 1] – light red would be something like [1 0.5 0.5].

The default color map is called ‘parula’. You can set a different color map using the command **colormap** followed by the color map you want to use in brackets:

```
colormap(hot)
```

To see the available built-in color maps, type

```
help graph3d
```

The best way to understand Matlab variables is to try manipulating them. The demo program will have defined a number of variables, which are shown in the workspace window pane or you can list using the command **whos**:

```
whos
```

```
Name                    Size              Bytes  Class
```

```
Attributes

  Pcond1                   1x1                   8  double
  Pcond2                   1x1                   8  double
  Psubj12c                 1x1                   8  double
  Psubj12nc                1x1                   8  double
  RTall                   10x4                 320  double
  RTc                      1x10                 80  double
  RTc1                     1x10                 80  double
  RTc2                     1x10                 80  double
  RTnc                     1x10                 80  double
  RTnc1                    1x10                 80  double
  RTnc2                    1x10                 80  double
  alpha                    1x1                   8  double
  catchtrialfraction       1x1                   8  double
  meanRTall                2x2                  32  double
  meanRTc1                 1x1                   8  double
  meanRTc2                 1x1                   8  double
  meanRTnc1                1x1                   8  double
  meanRTnc2                1x1                   8  double
  n                        1x1                   8  double
  nErrc                    1x1                   8  double
  nErrc1                   1x1                   8  double
  nErrc2                   1x1                   8  double
  nErrnc                   1x1                   8  double
  nPlayers                 1x1                   8  double
  player                   1x2                 136  cell
  stdRTc1                  1x1                   8  double
  stdRTc2                  1x1                   8  double
  stdRTnc1                 1x1                   8  double
  stdRTnc2                 1x1                   8  double
  t                        1x1                  72  cell
  trialNumber              1x10                 80  double
```

Later on, you can look in the file **reactionTimesDemo** for definitions of what some of these variables represent (but don't try to understand what that code does just yet).

The size column shows the array size of each variable – so 1x10 means 1 row, 10 columns. A quicker way of determining the size of a variable is to use the **`size`** command, which returns the same information for a particular variable:

```
size(RTall)
ans =
    10     4
```

If you want to know the size of a variable along a particular dimension, just specify that dimension as a number after the input variable, e.g., to show the number of columns (second dimension) type

```
size(RTall,2)
ans =
     4
```

All of the variables listed by **`whos`** make up your *workspace*. This data is stored in memory and will be erased when you exit Matlab unless you save them – see next section.

Exercises
a. Create a random matrix with the name **`matrix1`** 8 columns wide and 6 rows long.
b. Create another matrix called **`matrix2`** that is the same size as **`matrix1`** using the command **`ones.`**
c. Type **`matrix1+matrix2`**. What is the result?
d. Create a new matrix **`matrix3`** that is the transpose of **`matrix1`**
e. Type **`matrix3+matrix2`**. What happens? Why?
f. Create a 3-dimensional array variable with the name **`array1`** 10 rows long, 5 columns wide, and 13 slices thick filled with ones.

6. Loading and saving data
All the variables currently in your workspace can be saved from the command line by the **`save`** command:

```
save
```

This will save everything in the workspace to the file 'matlab.mat' in the current folder (the folder you are in when you enter the command). If you want to save the workspace to a different file name, such as 'mydata' just type that after the command:

```
save mydata
```

and the data will now be in the file 'mydata.mat' in the current folder. Note that Matlab will add the extension '.mat' to the file name, which is required so that the file can be recognised as a Matlab file – you should not add that yourself.

Loading data is done the same way using the **`load`** command:

```
load
```

loads the file matlab.mat if it exists, or prints an error message if it doesn't. This will restore all the variables saved in the matlab.mat file (and potentially overwrite any existing variables with the same name in the current workspace). To load a file with a different name, type the name after the command:

```
load mydata
```

loads the file 'mydata.mat' in your current folder. The demo program will have saved the resulting data in the file demodata – should you overwrite any variables you can type

```
load demodata
```

at any time to reload the data.

7. Plotting in Matlab

A great strength of Matlab is its ability to create almost any kind of data plot – not so much because the plots are particularly visually appealing (although with considerable effort they can be made to look quite good) but because they can be automated very easily. The demo program showed a number of basic plot types which we will briefly cover here – for more detail, look at the **`reactionTimesDemo`** script and see how the plot commands are used.

Line plots. Line plots show data points connected by a line. The line can be solid, dashed or dotted:

Plot RTc1 against RTnc1 in red solid lines with squares as plot marker symbols:

```
plot(RTc1,RTnc1,'rs-')
```

'rs-' means **r**ed **s**quares in a solid line – type **`help plot`** for a list of abbreviations for colours, line styles, and marker symbols. Try using different colours, line styles, and markers and see the effect.

Same but in blue dotted line with diamonds as markers:

```
plot(RTc1,RTnc1,'bd:')
```

You can leave out the x-axis variable – Matlab will then plot each of the numbers in the input in sequence as if the x-axis started at 1 and increased in steps of 1:

```
plot(RTc1)
```

will give the same result as

```
plot(1:10,RTc1)
```

Scatter plots. Scatter plots are useful when you want to plot one set of data against another – for example, if you want to see whether there is a correlation between two sets of data (in which case the plot symbols should fall roughly along a diagonal line). Scatter plots are similar to line plots but have no connecting lines and can be created using the command **`scatter`**. Examples:

Plot a scatter plot of RTc1 against RTc2 in blue circles:

```
scatter(RTc1,RTc2)
```

This should give the same result as

```
scatter(RTc1,RTc2,40,'bo')
```

40 is the size of the marker, 'bo' means **b**lue **o**'s (same abbreviations as used by 'plot' command) (NOTE: the default marker size may differ in your version of Matlab – adjust the value 40 to see what the plot looks like)

Plot a scatter plot of RTc1 against RTc2 in large solid (filled) red squares:

```
scatter(RTc1,Rtc2,100,'rs','filled')
```

'rs' – **r**ed **s**quares

Bar plots. Bar plots display each data value as a vertical or horizontal bar and are usually used when the data in the plot correspond to categories (e.g. catch vs non-catch trials, male vs female).

Example: to plot the mean reaction times for catch (meanRTc1) and non-catch trials (meanRTnc1) for player 1 as red vertical bars, type

```
x=[meanRTc1 meanRTnc1]
bar(x,'r')
```

A simpler way to do the same thing would be

```
bar([meanRTc1 meanRTnc1],'r')
```

To plot individual reaction times to catch trials for player 2 as blue vertical bars, type

```
bar(RTc2,'b')
```

Histograms. Histograms are a useful way of looking at data distributions. They are easy to make in Matlab using the command **`hist`**, e.g.:

```
hist(RTc1)
```

More recent versions of Matlab (not Octave) have the command **`histogram`** which does much the same thing but has more functionality:

```
histogram(RTc1)
```

Error bar plots. Error bars are used to indicate variability in a value, for example to show the standard deviation of a number.

Example: to plot the mean reaction times for catch and non-catch trials for player 1 with standard deviation error bars, using a magenta line and square plot marker symbols:

```
errorbar([meanRTc1 meanRTnc1],[stdRTc1 stdRTc2],'ms')
```

Multiple overlay plots. By default, any plotting command will erase what is in the figure already. To draw multiple plots on top of each other – for example, a bar plot with error bars on top – type

```
hold on
```

Any plotting command that follows will then add to what is already in the plot, without erasing it first. To go back to the default behaviour – so that new plot commands erase the current plot – type

```
hold off
```

Multiple plots in one figure (subplots). It is sometimes useful to show several plots in different locations in one figure. To do this, use the command **`subplot`** to select a portion of the figure to

draw to – any plot commands will then apply to that part of the figure only.

For example, to create two subplots in one rows and two columns and plot a scatter plot in the first subplot and a line plot in the second, type

```
subplot(2,3,1)
scatter(RTc1,RTc2,25,'bo')
subplot(2,3,2)
plot(RTnc1,RTnc2,'rs')
```

Saving figures as images. It is sometimes useful to save figures as images (JPG, GIF, or PNG format) that can be pasted into Word documents. To do this, on the Figure window menu select File->Save as and then select the right file type (PNG or JPG) and then enter a name for the image. This will save th figure as a bitmap (image) file that you can open with Photoshop or Paint or paste into Word or PowerPoint documents. You can also save them as encapsulated postscript (EPS) files that can be imported into vector drawing programs such as Illustrator or Inkscape; this will allow you to edit the figure elements (lines, symbols, axes etc) and will look a lot nicer.

8. Modifying variables: basic mathematical operations

Much of the power of Matlab lies in its ability to manipulate large sets of numbers at once. For example, if you want to measure the pairwise difference between two sets of reaction times by hand, you would have to calculate the difference between each pair of reaction time one by one. If instead you had the reaction times stored in two Matlab vector variables called RTc1 and RTc2, then you could compute this difference in one go by typing

```
RTcdiff=RTc1-RTc2
```

Assuming the two vectors were the same size, this would print out the pairwise difference between the two sets of reaction times and assign the result to a new variable called 'RTcdiff'. Similarly you can compute the pairwise sum between two variables using +:

```
RTcsum=RTc1+RTc2
```

To multiply or divide two variables, you need to use the symbols '.*' and './', respectively:

```
RTcproduct=RTc1.*RTc2
RTcratio=RTc1./RTc2
```

(You need to use the .* to indicate pairwise multiplication; the * or / symbols on their own indicate matrix multiplication and division, respectively, which are mathematical operations with a specific meaning which we won't cover in this introduction.)

It is often useful to compute the mean and standard deviation of a series of numbers. If you recall your first-year statistics, you will remember that the mean is defined as the sum of the numbers divided by how many numbers there are. You could do this in Matlab by typing

```
RTc1mean=sum(RTc1)./length(RTc1)
```

Here we have used the command **sum**, which sums the numbers of a vector, and divided the result

with the output of the command **length** which returns how many numbers a vector contains. Note that for both these commands, the variable you want to compute the sum or length of needs to be put inside parentheses () after the command. (If you have used formulas in Excel, this way of calling a command should be familiar.)

An easier way to compute the mean is to use the **mean** command directly, which does the same thing:

```
RTc1mean=mean(RTc1)
```

Similarly, to compute the standard deviation you would use the 'std' command:

```
RTc1std=std(RTc1)
```

In the demo program, **mean** and **std** were used to compute mean and standard deviation of reaction times for catch and non-catch trials and compare their difference. Use **mean** to see whether non-catch trials (RTnc1 and RTnc2) were faster than catch trials (RTc1 and RTc2) for each of the subjects. What was the result?

There are numerous similar commands for computing descriptive statistics in Matlab – try applying the following on Rtc1:

min
max
median
var

Type `help` followed by the command name to find out more what each command does.

The demo also compared whether the difference between catch and non-catch trials was significant using a command called **ttest2** – this command does a two-sample t-test. Type **help ttest2** to see what it does (look in the **reactionTimesDemo** code for an example of how it is used). (NOTE: On Octave this command is called **t_test_2**).

Statistics
This tutorial will not cover the many statistics tools available in Matlab (at least if you have the Statistics toolbox installed). However, you can do virtually any kind of statistical analysis possible with Matlab, although it is done rather differently than you would in e.g. SPSS. A particular strength of Matlab is that it lends itself well to doing bootstrap statistics which are powerful when your data doesn't conform well to normality assumptions – the drawback is you need to program the actual bootstrapping yourself to some degree. However, this type of statistical analysis is almost impossible to do with SPSS and similar software. For more information, look at the Statistical Toolbox in Matlab's online help.

Exercises
a. Compute the mean of each column of one of the matrix variables created in the first set of exercises in section 5.
b. Compute the median for the same
c. Plot the mean and the median in the same figure in red and green, respectively.

9. Scripts and functions

Commands and variable entered through the command line can be saved in a ***script*** to allow redoing them without having to type them in again one by one. A script is just a list (sequence) of commands entered in a text file with the extension .m. For example, you might want to create a Matlab script to output the current date, current folder, and list the files in that directory. Rather than typing the three commands that would do this in sequence:

```
date
pwd
dir
```

you could create a text file containing these three commands and save the file as 'myscript.m'. If you then type

```
myscript
```

Matlab would run the three commands exactly as if you had entered them separately. You can write scripts in Matlab's built-in editor (kind of like a simple word processing program) which you access through the top menu. You do not need to add the .m extension (which is necessary for Matlab to treat the file as a script) – Matlab will do this automatically.

A ***function*** is similar to a script in many ways – it is also a text file containing a list of commands. Unlike a script, a function takes some input (such as a variable or a number) and returns an output. In computing jargon, the input and output variables are often referred to as the "arguments" of the function. Most Matlab commands are actually functions – for example, the command **mean** is a function that takes an input (a vector of numbers) and returns an output (the mean of the vector). When you call a function, you have to provide the input to the function within parentheses () after the name of the function, and usually you want to provide a return variable that the result of the function will be saved into. For example, to compute the mean of a variable called x and save the result in the variable mx, you would type

```
mx=mean(x)
```

Many functions take more than one input variable (or argument) and many also return more than one output variable. For example the function **min,** which finds the minimum value of an array, can return both the minimum value and the index of that element (i.e., so that you could find where in the array the minimum value was). To return more than one output variable, you have to call the function with the names of the output variables *separated by commas* within *square brackets*:

```
myarray=rand(1,10);
[myminvalue,myminindex] = min(myarray)
```

To call a function with more than one input variable, you simply enter them separated by commas (but not within square brackets). For example the function **mean** can take a second input determining along which dimension to compute the mean value:

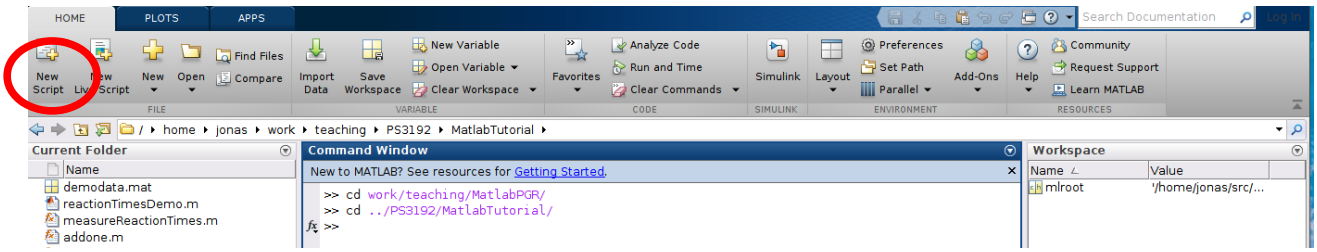
```
myarray=rand(5,10);
meanrows=mean(myarray, 1) % this means to compute the mean down
```

```
each row (dimension 1)
meancols=mean(myarray, 2) % this means to compute the mean down
each column (dimension 2)
```

A function is just a text file with the extension .m, just like a script, but differs in that the first line MUST begin with the text

function <output>=<filename>(<input>)

where filename is the name of the function and the file it is stored in, and this is followed by a parentheses and the list of input variables separated by commas, and finally a closing parenthesis. This is a easier to understand by demonstration: Let's make a function called 'addone' which will take a number as input and return that number plus 1. Open the editor as follows:
In the top menu, select New Script:

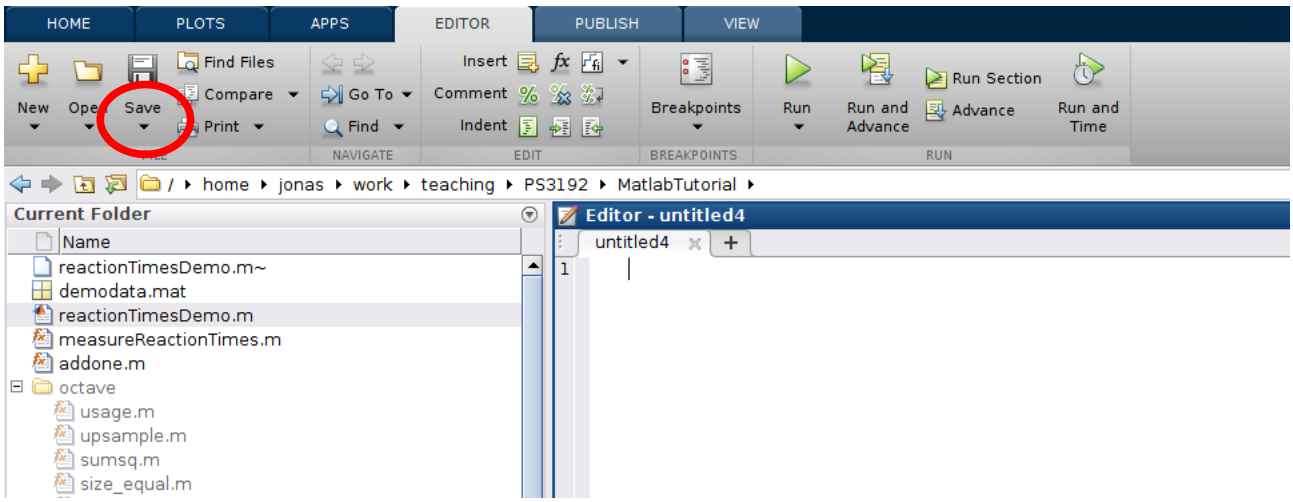


This will open a new window pane (the editor) above the command window. Text you enter into this window can be saved as functions, which are text files that must have the name of the function with the.m extension.

Now type the following into the editor window:

```
function output=addone(input)
% This function adds 1 to its input.
output=input+1;
```

Now save this as 'addone' (click on Save which opens a file menu and save the file as `addone.m` – Matlab will likely already have suggested this name in the file menu).



Now try typing in the command window

```
addone(4)
```

What is the result? Note that you didn't specify whether the input was a single number, a vector, or a matrix; the function will work equally well with any (because in Matlab a single number such as 1 can be added to any numeric variable regardless of size). Try typing

```
addone([1 2 3 4 5 6 7])
```

and

```
addone(ones(4,5))
```

and see what happens. (Recall that the **ones** command creates an array consisting of 1.) Obviously, you could call the function with a variable as input, e.g.

```
x=[-1 3 99]
addone(x)
```

You may wonder what is the purpose of the text after the % in the addone.m function. Normally, any line beginning with % is treated like a comment that is ignored by Matlab, but if it appears directly below the first line in a function, it is interpreted as help text for that function. To see this, type

```
help addone
```

which should return

```
This function adds 1 to its input.
```

Try to modify the text after the % and see what happens when you type **help addone** (remember to save the changes to **addone** first).

For an example of what a 'real' Matlab function looks like, you can type the command **type** followed by the function name; for some commands this will print the function text in the command window. Try

```
type mean
```

to see what the mean function does (don't worry if you don't understand much – it is just to give you an idea of what happens 'under the hood'). Notice the help text starting with % at the top of the function (compare the output of **type mean** with **help mean**). For some functions, this won't work – you will get the message 'function xxx is a built-in function', which simply means it is hard coded in machine language and can't be printed in text format.

When to use functions and when to use scripts: Scripts are best used when you have a sequence of commands you want to run over and over – basically saving you the effort of typing the commands into the command window every time. Scripts have access to all the variables in your workspace –

exactly as if you were typing the commands manually. Functions, on the other hand, only have access to their input variables you pass them when they're called, and the output variables they return. A function can't "see" the variables in your workspace, nor can you access the variables inside a function. This makes functions much safer than scripts, because they can't modify or delete any variables that you have in your workspace. For example, let's say you have a variable in your workspace called myvar that you assign the value 1:

```
myvar=1;
```

Suppose you have a script called myscript.m that adds 1 to the myvar variable:

```
myvar=myvar+1;
```

Every time you run `myscript`, `myvar` in your workspace will increase in value by 1. If instead you make a function called `myfun` that also adds a 1 to a variable to `myvar`, the result will be quite different:

```
function myfun
myvar=myvar+1;
```

When you run `myfun`, the variable `myvar` in your workspace will not change. This is because the variable `myvar` *inside* the `myfun` function is treated as a completely different variable to the one in your workspace; the function `myfun` has no access to the variables outside the function unless you pass them directly to the function as inputs (and even then it can't change them – it only gets a copy of them). Because functions can't inadvertently change variables outside of the functions, they are much safer to use than scripts and are ideal for dividing your code into logically separate and independent sections.

10. Programming in Matlab

Programming is simply a way of instructing a computer to perform a series of commands in a sequence, so in fact the **addone** function you wrote above is a very simple programming example. There is not enough time in this workshop to go into great details of programming but we will touch on two essential concepts – using loops to repeat a set of commands several times, and using 'flow control' commands to do modify what the program does depending on its current state. Open the editor (see previous section), create a new m-file, and type in the following:

```
for n=[1 2 3 4 5 6 7 8 9 10]
     disp(n)
end
```

Save this in a file called 'myfirstprogram'. Then type

```
myfirstprogram
```

in the command window. What is the result?

The first line of this program defines a for-end loop. A loop is a sequence of commands that are repeated a number of times (or *iterations*). Here, we tell Matlab that the variable n should be set to the list of numbers following the = sign; on the first run (iteration) of the loop, n is set to 1, the second iteration n is set to 2, etc until the end of the list is reached. On each loop, Matlab runs all

the commands between 'for' and 'end'. In the example, this is simply using the 'disp' command to show the value of n on the command line. (Normally, we would use Matlab's built-in method for creating a vector variable increasing in steps of one using a colon, i.e. **`for n=1:10`**, rather than typing all of the numbers in a sequence.)

We will now add a *flow control* command – 'if-else-end' – to the script. Modify the code to look as follows:

```
for n=1:10
     disp(n)
     if (n<5)
          disp('n is smaller than 5')
     else
          disp('n is larger than 5')
     end
end
```

Run the script. How is the result different from the first time?

Like before, the script will display the value of n on each iteration, but then it will also check each time whether the current value of n is smaller than 5 – if it is, it will print 'n is smaller than 5', if not (`else`) it will print 'n is larger than 5'.

The example above also demonstrates how to do logical comparisons in Matlab – i.e. testing if a variable is smaller than or equal to some value. These are very frequently used in programming and are largely self-explanatory, below is a list of the common comparators:

> greater than
< smaller than
>= greater than or equal to
<= smaller than or equal to
== equal to (NOTE that it is two = signs; a single = means assignment, so x=y means ‘set x to the value of y’, NOT ‘is x equal to y’ – this is a common typo that can cause major confusion)
~= not equal to
~ not (e.g. **`~(x==y)`** is equivalent to **`x~=y`**)

Another useful command for loops is **`while`**, which is used to run a loop as long as a condition is met – it combines looping with flow control. The following code does the same as the one above but uses while instead of **`for`**:

```
n=1;
while (n<=10) % Runs the code between ‘while’ and ‘end’ until n>10
     disp(n)
     if (n<5)
          disp('n is smaller than 5')
     else
          disp('n is larger than 5')
     end
     n=n+1; % Increases n by 1 each iteration
end
```

You have now written your first proper program in Matlab. If you are interested in learning more, consult the web resources at the end of this document and Matlab's documentation which has tutorials on programming. Don't worry if it seems daunting; programming takes time and practice to learn.

Exercises
a. Write a program that creates a 3x4 matrix of random numbers and prints each element by row and column
b. Modify the program so the matrix is passed as a variable to the program
c. Modify the program so the matrix can be any size (hint: use the command **`size`** to figure out how many rows and columns the matrix has)

11. Writing efficient code: vectorisation
Loops are handy and intuitive for carrying out an operation to each element of a vector or array. So for instance, if you wanted to add two vectors to each other, element by element, you could write

```
vec1=rand(1,5000);
vec2=rand(1,5000);
for n=1:length(vec1)
     vec3(n)=vec1(n)+vec2(n);
end
```

While this works, this is actually not the most efficient way to do this in Matlab. Matlab is built around the concept of "vectorisation", which means that it is has built-in functions to manipulate vectors and arrays as single variables, rather than element-by-element, and these built-in functions are much faster. For example, the code above could be rewritten as follows in vectorised form:

```
vec1=rand(1,5000);
vec2=rand(1,5000);
vec3=vec1+vec2;
```

Not only does this speed things up, it also gets around the problem of Matlab having to change the size of vec3 on every iteration (because Matlab doesn't know from the start how many elements will go into vec3, it has to increase the size of vec3 by one element each iteration of the loop, a very time consuming operation). If you use large data sets, vectorisation can have significant benefits in speed and performance. For example, try entering the following into a script (not at the command line!) and save it as `vectest.m`, then type **`vectest`** in the command window and see the difference in speed between looped code and vectorised code:

```
vec1=rand(1,500000);
vec2=rand(1,500000);
disp('Running in a loop')
tic
for n=1:length(vec1)
     vec3(n)=vec1(n)+vec2(n);
end
t1=toc;
disp(['Elapsed time is ' num2str(t1) ' seconds'])

disp('Running vectorised')
```

```
tic
vec3=vec1+vec2;
t2=toc;
disp(['Elapsed time is ' num2str(t2) ' seconds'])

disp(['Vectorised code is ' num2str(t1/t2) ' times faster'])
```

Vectorisation is particularly important when dealing with large data sets and when using Octave – recent versions of Matlab are quite efficient even when running loops, although vectorised code is still significantly faster.

The code above introduced a couple of useful commands for checking how optimised your code is. First, you have the timer functions **`tic`** and **`toc`**; **`tic`** starts the timer, and **`toc`** ends it; **`toc`** on its own prints the elapsed time since the last time **`tic`** was called, but if you call it with an output argument (e.g., **`t1=toc`**) then it assigns the elapsed time (in seconds) as a value to that argument. So you can measure how long any code took to run by preceding it by **`tic`** and then following it by **`toc`**. Note that to be meaningful, this must be done in a script, otherwise **`toc`** will measure the time it took you to enter each command on the command line which is usually much longer than the time to actually run any meaningful code.

The other two useful functions here are **`disp`**, which displays a text string, and **`num2str`** that converts a number (such as a variable) to a text string that **`disp`** can print. Type **`help disp`** and **`help num2str`** to get more information about how they work.

12. Advanced variables: strings, structures and cell arrays

So far, we've looked at numeric variables (scalars, vectors, matrices, and arrays) but Matlab also supports many other types of variables that can be very useful. Some of the most useful ones are *strings*, *cell arrays* and *structures* (or *structs*).

Strings

Strings are used to store text information, such as names and labels. They're not really intended for storing large chunks of text, although in principle you could store an arbitrarily large text inn a single string variable. To create a string, you simply enter the text within single quotes, as discussed previously:

```
mystring='Hello world!'

mystring=
     'Hello world!'
```

Strictly speaking, the string above is a *character array*: an array (a vector to be precise) of characters (the letters). Character arrays are in some ways like numeric arrays; you can index them in the same way, e.g. to print the 7th to the 11th element of `mystring` above you could type

```
mystring(7:11)

ans = 'world'
```

More recent versions of Matlab has a similar data type called *string arrays* which can hold strings of different lengths. These are created using double quotes instead of single quotes:

```
mystring="Hello world!"
```

Note that Octave does not at the present support string arrays – the above command would just generate a simple character array, the same as if you had used single quotes.

Matlab has numerous functions to manipulate both character arrays and string arrays – search for "Characters and strings" in the documentation. Some useful functions are **`strfind`** (to find one string in another), **`strcmp`** (to compare if two strings are the same), etc.

Cell arrays

Although Matlab can store single strings in single variables, often you have lists of strings that you want to store. For example you might have collected data for four subjects called Simon, Louis, Cheryl, and Mel and you might want to put them into a list. To do this you use what is called cell arrays in Matlab, which is simply a way of storing arbitrary data in a single variable. (In recent versions of Matlab, you can also use string arrays, see above.) So to create your list you would type:

```
mysubjects=cell(4,1);
```

This means "create a cell array that is 4 rows and one column wide".

Then you can put the name of each participant into the array:

```
mysubjects{1}='Simon';
mysubjects{2}='Louis';
mysubjects{3}='Cheryl';
mysubjects{4}='Mel';
```

or, more compactly:

```
mysubjects={'Simon';'Louis';'Cheryl';'Mel'}
```

Note the use of curly brackets – these must be used when indexing cells. To display the third subject you type

```
mysubjects{3}
```

A cell array can hold anything and each element of a cell array can be any type of variable. So each element can be anything – not just strings. You could for example add a 3D numeric array to your list without Matlab complaining:

```
mysubjects{5}=rand(5,7,3)
```

although this wouldn't be particularly useful!

Structures

It is sometimes useful to group together related data rather than having lots of separate variables. For example, you might have collected data on the names, heights, and favourite colour for several subjects. You could create one cell array for the names, another for the colours, and a third for the heights. Alternatively, you could put them all into a single cell array where the first column is the

name, the second the colour, and the third the height:

```
mydata=cell(4,3);
mydata{1,1}='Simon';
mydata{2,1}='Louis';
mydata{3,1}='Cheryl';
mydata{4,1}='Mel';
mydata{1,2}='pink';
mydata{2,2}='turquoise';
mydata{3,2}='red';
mydata{4,2}='black';
mydata{1,3}=175;
mydata{2,3}=170;
mydata{3,3}=165;
mydata{4,3}=170;
```

Then to show what Simon's favourite colour is you can type

```
mydata{1,2}
```

However, this way you can't tell what each column stands for as easily as when each data set is kept in a single variable. Sometimes it can be more useful to store related data in complex variables called structures (or structs) which can hold different types of data but with names. A struct holds its data in named **fields.** For instance, in the above example we could have a struct with the fields **name**, **colour** and **height** to hold the information we need. For example if the data was held in a struct called mystruct we would access the fields by typing the name of the struct, a dot . and the field name:

```
mystruct.name
mystruct.colour
mystruct.height
```

This won't work, since we haven't yet created the struct. To do this, we create a struct holding information about name, colour and height for the first subject:

```
mystruct.name='Simon'
mystruct.colour='pink'
mystruct.height=175
```

However, this just holds information for Simon – luckily in Matlab, structures can form arrays of any size, so we can just add a second struct for Louis:

```
mystruct(2).name='Louis'
mystruct(2).colour='turquoise'
mystruct(2).height=170
```

and so on. Note that when we created the first struct, we could equally have written

```
mystruct(1).name='Simon'
mystruct(1).colour='pink'
mystruct(1).height=175
```

because Matlab considers any variable with only one member as part of an array that is 1 elements long (so if you are creating a new variable called q, there is no difference between writing **q=3** or **q(1)=3** – the result is the same). A drawback with structs is that it is a bit more difficult to look at all the information in an array of structs as it will only be shown in detail for individual elements, and it will depend on the application whether a cell or a struct is the most suitable.

13. Data management: finding and sorting values in arrays

When you get into writing scripts and functions it is often useful to be able to find the indices of particular values in arrays – for example you might have a vector with correlation coefficients (r) and you want to find the elements that are above some threshold (say r>0.5). You can find which elements meet this criterion simply by using the > symbol to the whole vector:

```
rvalues=[0.45 0.23 0.67 0.11 0.76 0.23 0.14 0.34 0.63];
rvalues>0.5

ans =

  1×9 logical array

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

This returns a *logical array* – a vector where 1 means the element was greater than 0.5, and 0 otherwise. To retrieve the indices of those elements above threshold, use the command **find**:

```
find(rvalues>0.5)
ans =

     3     5     9
```

find can also return the indices of arrays with multiple dimensions, in which case you call it with multiple output arguments, one for each dimension. Type **help find** for details.

Other useful commands for managing data arrays are sorting (**sort** and **sortrows**), finding unique values in an array (i.e. values without duplication) using **unique**, intersections (finding the elements in common) and unions (finding elements in either array) between different arrays with the commands **intersect** and **union** respectively. Use the documentation to learn more about how these commands work.

14. A practical application: deconstructing the demo

You have now come across many of the main features of Matlab. To learn more you will need to practice using the program and exploring different commands. It is often useful to start with an existing script or function and go through it line by line to understand what it does, and try modifying it bit by bit to see what happens. If you have come this far, you may want to open the demo **reactionTimesDemo** in the Matlab editor and go through each line and see if you can follow what it does. Note that lines beginning with % are comments – these are just there to make the functioning of the script easier to understand but are otherwise ignored by Matlab. Note that the script makes use of some features not covered by this tutorial, so don't worry if some things seem obscure – focus on understanding the overall structure and logic instead.

15. Next steps

If you want to continue from here, there are numerous web sites with Matlab tutorials – simply search for “matlab tutorials” using your favourite search engine. The following sites are a good start:

www.cyclismo.org/tutorial/matlab/
www.math.utah.edu/lab/ms/matlab/matlab.html
www.math.mtu.edu/~msgocken/intro/intro.html
ubcmatlabguide.github.io/

Also, the company that makes Matlab, Mathworks, has tutorials and even live demonstrations (that allow you to run Matlab in a web browser for training) on their website:

www.mathworks.com

If you want to try out Octave (a free alternative to Matlab with similar commands and capabilities, although not quite as good at plotting), go to:

www.gnu.org/software/octave/

Octave has loads of toolboxes that replicate much of the functionality of Matlab – on Linux these are easily installed using the package management system, on Windows it requires you to download and install these manually, see Octave documentation for details.

Googling 'Matlab tutorial' will bring up a wealth of resources that may be helpful.

Matlab has lots of built-in demos, type

```
demo
```

which will open a window listing all available examples – click on MATLAB to find more basic demos to begin with. The documentation (accessed by typing `doc` in the command window) is useful if you want to know exactly how a specific command is used.