Matlab Programming Module

Dr Peter Hansen
p.c.hansen@bham.ac.uk

School of Psychology / BUIC

Workshop 1
Getting Started with Matlab

Revision: 28 Sep 2017

Getting Started

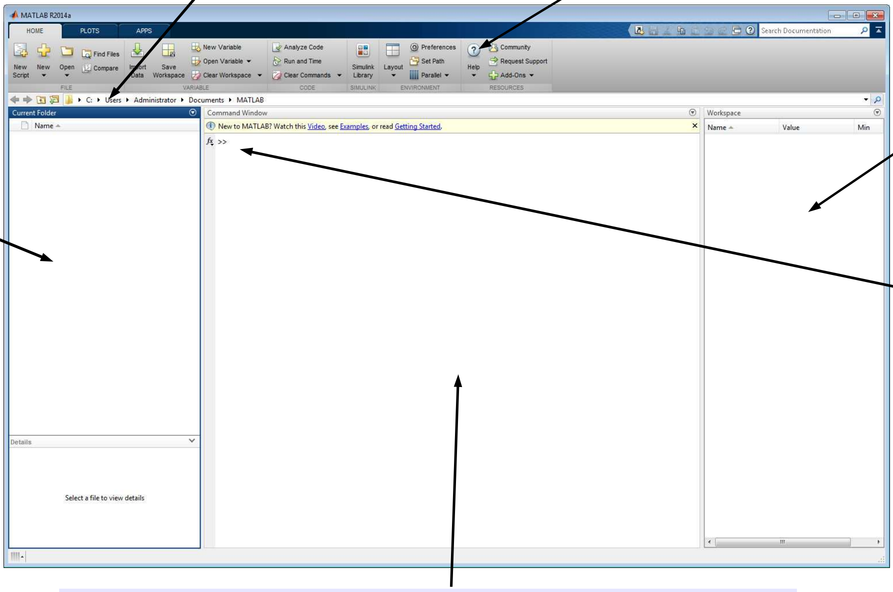
- Log in to the PC in front of you with your normal University username and password
- You should now have access to a Windows 10 desktop
- *If there is a problem with this step let us know*

- Start Matlab using the startup icon on the desktop or by finding Matlab R2017a via Windows Startup menu
- Matlab should now start up
- *If there is a problem with this step let us know*

The Default Matlab GUI Environment

View or change the current directory

Get help



New to MATLAB? Watch this Video, see Examples, or read Getting Started.

```
fx >>
```

Select a file to view details

This window shows the current directory contents

This window shows the workspace variables

>> is the Matlab command window prompt. You type commands here and Matlab will return its output here

This is the main interactive command window where you type commands

Quitting Matlab

In the Matlab command window type:

quit

Note here that items in red like this will generally be things <u>you</u> will need to type in the Matlab command window

To restart Matlab use the startup icon on the desktop or find Matlab on the Windows startup menu again. Make sure you are happy with starting up and closing Matlab once or twice.

One small modification to the default GUI that may be worth making is to show the command history. Through the GUI interface select:

Layout → Command History → Docked

If you find later you do not use or want this extra window then you can close it with:

Layout → Command History → Closed

Getting Help in Matlab

For general Help either (1) Hit F1 key (2) use “?” button on the Help tab on the menu bar or, (3) on the Matlab top menu go to:
Help → Documentation

In the command window, to find a command or function type:
lookfor average

Once you know the name of the command, to get help on it type:
help mean
The help will appear in command window.

To get help, with the help appearing in a separate help browser:
doc mean

Try running the above commands

Assigning a number to a variable

Type the following into the command window:
a = 10

a is the name we have chosen to give for a temporary variable to hold the number

= means assign the value on the RHS to the named variable on the LHS

10 is just the integer number ten

The overall result of this command is to take the number **10** and to **assign** it to a variable called **a**

Now try:
b = 10 + 2

In this command you are taking the number 10, adding 2 to it and assigning it to a variable called **b**

Now try:
c = 10 * 2

In this command you are taking the number 10, multiplying it by 2 and assigning it to a variable called **c**

This shows that you can use Matlab like a simple interactive calculator

Workshop Notation

As previously mentioned when you see a Matlab command printed in red as with the following command:

```
c = 10 * 2
```

then you should type this at the Matlab command prompt to help understand what it does. In this case the command calculates the value of 10 times 2 and assigns the result to a variable called **c**. Matlab will also show us the result of this computation in the same command window e.g. it will then print:

```
>> c =
         20
```

Whenever you see >> printed in these notes this is to help show you that Matlab has returned some result to you. This output should be what you see in your live Matlab session (if not something has gone **wrong**). You do **not** need to type this code in.

Arithmetic Operations

Common arithmetic operators include:

Operator	Meaning
=	Assign to
+	Addition operator
-	Subtraction operator
*	Multiply operator
/	Division operator
^	Power operator

Assigning a number to a variable

Now try:
d = 10 - 2

Here you are taking the number 10, subtracting 2 from it and assigning it to a variable called **d**

e = 10 / 2

Here you are taking the number 10, dividing it by 2 and assigning it to a variable called **e**

f = 10^2

Here you are taking the number 10, raising it to the power of 2 (i.e. squaring it) and assigning it to a variable called **f**

Or these can be combined together, for example:
g = 1 + 3*2 – 10/5

Once you have more complicated expressions there is room for user confusion about the order in which they are evaluated, e.g. does:
h = 2 * 10 + 5 give h a value of 25 or 30? Try it and see.

Matlab has a built in operator precedence so that it always evaluates expressions in the same way. Later we will see how to use brackets to remove any user ambiguity.

Useful Tip

When typing commands in the Matlab command window you can use the cursor (arrow) keys to navigate to previous commands:

- Use the up arrow to go back to a previous command
- Use the down arrow to come back to a later command
- Use the left and right arrows to move along a command
- In this way you can quickly retrieve, edit and run a modified version of an earlier command (without having to retype it all)

Try doing this now. Using the edit keys, go back to your earlier command to assign 10 + 2 to the variable b, and change this to assign instead 10 - 2 to b

Data in Different Dimensions I

- So far we have a assigned single, unique number when creating variables. e.g   a = 75
- The above might represent a single value from a data set, such as the score on Test 1 for one person
- But usually we are interested in data sets which comprise more than one number:

  We might have the scores on Tests 1, 2 and 3 for that same person

  Or we might have the scores on just Test 1 for all 10 people in a class

Data in Different Dimensions II

- For a single number held in a variable we can think of this as being “zero-dimensional” data
- Single numbers have only magnitude, not direction. Sometimes called “scalars”.
- For a set of numbers we can think of this as being like a line of numbers. This is “one-dimensional” data.
- 1-d data often goes by different names such as
  - vector
  - 1-d array
  - 1-d matrix
- These are all the same in Matlab. The names are interchangeable

Data in Different Dimensions III

- More specifically:

  The 1-d data might be written out in a line from left to right in a row. In this case it is called a **row array**

  Or the 1-d data might be written out in a column from top to bottom. In this case it is called a **column array**

- Remember both can still be called a 1-d array, a vector or a matrix

Creating 1-D Data I

Data can be created in different ways.
Try entering the following in the command window:

```
a=[ 5, 4, 3, 2, 1 ]
```

a is a row array of 5 nums

Note the use of **square [ ]** brackets here to assign these values

Note also use of the **comma ,** symbols here. When constructing an array, a comma separates elements in the same **row**

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

b is also a row of 5 nums
b is the same as **a** above

Note the commas have been omitted yet it still works as before. This is a <u>shortcut</u> when creating row arrays. Matlab has taken the commas to be **implicitly** still there.

Try:

```
c=[ 5, 4  3, 2  1 ]
```

What do you think will happen here?

Creating 1-D Data II

Now try entering the following:

```
d=[ 10; 9; 8; 7; 6 ]
```

d is a column array of 5 numbers

Note again the use of **square [ ]** brackets here to assign these values. *Square brackets are <u>always</u> used for assigning values.*

Note the use of the **semicolon ;** symbols here. When constructing an array a semicolon creates a new row.

There is no shortcut when creating column arrays – the semicolon must be used every time.

Try:

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

What do you think will happen here?

Using a Matlab function

Now try entering the following:

```
e = mean( d )
```

Note the use of round **()** brackets here

mean() is a built-in Matlab function

Functions **always** use round brackets

An array **d** containing numbers 10,9,8,7,6 has previously been created.

The **mean()** function has then been applied to this array.

The result has been saved into the variable **e**

Creating 1-D Data

Let us relate this back to the example above. Suppose student 1 in the class scores 75 on Test 1, 63 on Test 2 and 80 on Test 3. We could put this information into a single Matlab variable as follows:

```
Scores_Student1 = [ 75 63 80 ];
```

Column 1 = Test 1 value
Column 2 = Test 2 value
Column 3 = Test 3 value

Scores_Student1 is the name of the variable. This is a 1-d **row array** of 3 numbers

Or if we want to store the information on Test 1 for a class of all 10 students we could put this into one single Matlab variable as follows:

```
Scores_Test1 = [ 75; 50; 43; 82; 64; 67; 59; 71; 72; 81 ];
```

Scores_Test1 is the name of the variable. This is a 1-d **column array** of 10 numbers. Each entry (each row) represents the score on Test 1 for students 1-10.

Data in Different Dimensions IV

- Going slightly further we may want to represent in a single Matlab variable the data from both the set of tests conducted (Tests 1,2 and 3) and from all 10 of the students in the class
- If we put the data in a table format, with rows representing students and columns representing scores then this would be a two-dimensional variable
- In Matlab, these are called 2-d arrays or 2-d matrices

Data in Different Dimensions V

- Conceptually this 2d matrix will then look like this:

```
variable  =  [    Student1_Test1      Student1_Test2      Student1_Test3     ;
                  Student2_Test1      Student2_Test2      Student2_Test3     ;
                  Student3_Test1      Student3_Test2      Student3_Test3     ;
                                         …….
                 Student10_Test1     Student10_Test2     Student10_Test3     ]
```

Creating 2-D Data

Try entering the following:

a=[5 4 3 2 1; 10 9 8 7 6]
a is now a 2-D matrix with 2 rows and 5 cols

b=[5, 4, 3, 2, 1; 10, 9, 8, 7, 6]
b is the same 2-D matrix as **a** with 2 rows and 5 cols

c=[5 4; 3 2; 1 10; 9 8; 7 6]
c is now a 2-D matrix with 5 rows and 2 cols

d=mean(c)
The **mean()** function has been applied to the 2-D matrix **c** and saved in the variable **d.** By default **mean()** will return the mean of each column.

Creating 2-D Data

If we relate this back to the previous example of wanting to encode in a single Matlab variable the 3 test scores for all 10 of the students in a class we could do the following:

Column 1 = Test 1 Column 2 = Test 2 Column 3 = Test 3

```
Scores_All = [   75  63  80;
                 50  51  57;
                 43  50  52;
                 82  79  75;
                 64  67  71;
                 67  63  70;
                 59  53  62;
                 71  72  77;
                 72  59  80;
                 81  83  87 ]
```

Each row contains the 3 scores for a single student

Row 1 = Student 1
..
Row 10 = Student 10

More Ways of Creating Data I

It is often useful to create data in a regular sequence. In Matlab this can be achieved using the **colon** : operator. Enter the following code:

Only 2 numbers are used here so the step size is **implicitly** assumed to be 1.

a=1:10

a is a row of 10 numbers 1-10 in steps of 1

3 numbers are used here. The middle number is the step size. It is set to 2.

b=2:2:20

b is a row of 10 numbers 2-20 in steps of 2

Note that it does not matter here if the sequence is wrapped in square assignment brackets or not. Not using them here is another Matlab shortcut where the missing brackets are taken to be **implicitly** present. You will get the same result:

b=[2:2:20]

More Ways of Creating Data II

The step size can be anything:

c=1:0.1:10
c is a row of 91 numbers from 1-10 in steps of 0.1

d=10:-1:1
d is a row of 10 numbers from 10-1 in steps of -1 (going down)

e=0:pi:314
e is a row of 100 numbers in steps of pi (3.1415...) going up.

This example also shows some mathematical constants (like pi here) are preprogrammed into Matlab

Creating 3-D Data

For creating large or higher dimensional matrices it is usually easier to create the matrix first with a function and then to fill it in as needed:

There are 3 arguments to this function. That tells you this will be a 3-D matrix.

a=zeros(2,2,2)
a is now a 3-D matrix of size 2x2x2 with all elements set to 0
The function **zeros()** has been used to create this.

There are 4 arguments to this function. This tells you this will be a 4-D matrix

b=ones(4,3,5,6)
b is now a 4-D matrix of size 4x3x5x6 with all elements set to 1
The function **ones()** has been used to create this.

Matlab and Errors

You will find Matlab is very particular about its syntax. It is very easy to make a small mistake and then Matlab will complain by generating an error. The error text may not always be entirely obvious. For example:

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

Here I am trying to create a 2-D 3x3 matrix but I have mistakenly left out one element (no 6) on row 2. This won't work – I need 9 elements in a 3x3 matrix, not 8 as I've tried to use here. Oops.

Matlab will generate the error:

```
>> Error using vertcat
>> CAT arguments dimensions are not consistent.
```

Do not panic when such errors appear. Everything can be fixed.

Using the Editor Window

- Later in the workshops we will use the Editor window to create/save scripts and functions
- For now we can use it as a temporary buffer from where you can cut and paste commands
- In the main Matlab GUI, go to

  Home tab → New Script

  (or press CTRL-N)
- A new Editor window should open

Using the Editor Window

- In the main Matlab command window enter the command: a=1:10 (**a** is a row of 10 numbers 1-10 in steps of 1)
- Use the mouse to highlight the command
- Either mouse right-click and select the **copy** option, or hit CTRL-C on the keyboard
- Select the Editor Window
- Either mouse right-click and select the **paste** option, or hit CTRL-V on the keyboard
- In this way commands can be temporarily saved, copied and pasted between the Editor window and the Matlab command window

Creating Linear Sequences

Data can be created in a linear sequence using built-in Matlab functions

linspace(x1, x2, N) ←

x1, x2 and N are known as the **arguments** to the function
If N is not defined it defaults to 100

This function generates N points linearly between x1 and x2.

Note again the use of **round ( ) brackets** here.

linspace() is a Matlab function. Recall that functions in Matlab **always** use round brackets when passing arguments.

Arguments are **always** separated by **commas**.

For example:

```
a = linspace(1,19,10)
```

```
>> a =  1    3    5    7    9   11   13   15   17   19
```

Creating Nonlinear Sequences

Data can also be created in a nonlinear log sequence using the function:

logspace(x1, x2, N) ← If N is not defined it defaults to 50

This function generates N points logarithmically equally spaced points between 10^x1 and 10^x2.

For example:

This function is asking for 5 equally log-spaced points between 10^1 (10) and 10^2 (100)

```
a = logspace(1, 2, 5)

>> a = 10.0000   17.7828   31.6228   56.2341  100.0000
```

Suppressing Display Output

You can use a final semicolon **;** to suppress output from being shown in the command window. For example try the following:

```
x=1:10
y=1:10;
```

← **no** semicolon here

← But there **is** a semicolon here

Notice how output from the second command is not echoed to the Matlab command window. To see the values of y in the command window just type the variable name without a final semicolon

```
y
```

One side effect of this ability is that you can use a semicolon to put more than one command on the same command line. All commands will be run but nothing will be echoed to the screen. For example:

```
a = 1; b = 2; c = 3;
```

Overloading the ; symbol

You have just seen an example of something that is often seen in programming languages. This is where a particular language symbol or feature is used to provide one function in one context and another function in a different context. So, with the semicolon symbol, we saw earlier that it can be used to separate rows when creating an array:

```
d=[ 10; 9; 8; 7; 6 ]
```

But we have just seen in the previous slide that it can also be used to suppress output from being shown in the command window.

```
y=1:10;
```

There is no link between the function of the ; in the first command to the second command. Rather the same symbol is being re-used to do something different. This concept is called **overloading**. There are several such cases in Matlab. It can be a bit confusing.

Manipulating Data - Transpose

Use an apostrophe **'** to transpose a matrix (swap around its axes)

For example type these commands:

```
x = [ 1 2 3; 4 5 6 ];
x
x'
```

You should see that x is different in the two cases, one being the transpose of the other (i.e. the rows and columns are flipped around)

Make sure you understand the difference between x and x'

Variables I

Data in Matlab are stored in variables. These have names that you as the programmer can choose. For example, can choose to name this variable **a**:

```
a=1:10
```

a is a row of 10 numbers from 1 to 10

This data can be assigned (copied) to another variable with a different name. For example:

```
b=a;
```

b now has the same values as **a**

```
b
>> b = 1 2 3 4 5 6 7 8 9 10
```

Note however that **b** is a ***copy*** of **a** and does not link to the same data as in **a**.

If you change the values in **a** or delete **a**, the values in **b** are unchanged.

Variables II

Variables can also be safely assigned to themselves, overwriting their current values:

```
a = 1
a = a + 1
a = 2
```

The way this works it as if Matlab calculates an internal temporary variable and then assigns the value of that back to **a** thus overwriting the old value.

It is as if Matlab internally does this:

```
a = 1
tmp = a + 1
a = tmp
```

```
b = [1:5]
b = b - 1

>> b = 0 1 2 3 4
```

There is another Matlab shortcut in operation here. Rather than forcing you to create an equal sized array to **b** containing only 1s before the subtraction:

```
b = b - ones(1,5)
```

Matlab has understood that you cannot subtract a scalar number (1) from the 1-d array (1:5) and it has therefore **implicity** expanded the 1 to be an equal sized array to that of array **b**. Sometimes this kind of assumption can create subtle bugs in Matlab code.

Variable Names

Matlab has strict rules about allowable names for variables.
Variable names cannot start with a number or a symbol:

```
a123 = 5;    ← OK
123a = 5;    ← Error!
[a = 5;      ← Error!
```

Matlab will allow you to define a variable with the same name as an existing Matlab command. This is a really bad idea, and will likely create an error or confuse you later on:

```
quit = 5;    ← Legal but very dangerous. Do  not do this!
clear = 2;
```

Matlab is also case sensitive:

```
g = 5;    ← G and g are two different variables. Be careful to remember this.
G = 6;
```

Other Useful Commands

who	Lists the variables in the current workspace
whos	Gives more information about variables
clear	Remove variable from workspace e.g. clear a
clear all	Removes all variables from workspace
pwd	Shows current directory
ls	Lists files in current dir
clc	Clear command window

Other Useful Functions

length() Returns the length (no of items) for 1D vector e.g.

```
a=1:10
length(a)
>> ans = 10
```

If the input is a matrix, length() returns the length of the biggest dimension e.g.

```
b=ones(2,5)
length(b)
>> ans = 5
```

size() Report exact size of matrix (no of rows, no of cols) e.g.

```
size(b)
>> ans = 2 5
```

Some More Useful Functions

```
x = [ -1 3 4 ];
```

Try using some of the following functions. Try to understand what each function does and what the return values mean (use **help** if you need to):

abs(x)	Absolute value
min(x)	Minimum
max(x)	Maximum
sqrt(x)	Square root
sum(x)	Sum
mean(x)	Mean
median(x)	Median
range(x)	Range
std(x)	Standard deviation
var(x)	Variance

Forgotten how to get **help()** for a particular function? Just type:

```
help abs
```

at the command line prompt

A word on complex numbers

If you are unfamiliar with complex / imaginary numbers skip over this

Some of the previous examples will return imaginary or complex numbers. The simplest example is:

```
sqrt(-1)
>> ans =  0 + 1.0000i
```

Complex numbers can be created with same syntax, e.g.:

```
c1 = 3 + 4i
```

Note that j can be used as well as i for the imaginary no:

```
c2 = 3 + 4j
```

Be very careful therefore if your code may need to handle or generate complex numbers. You should avoid variables called i or j

For next week

- Make sure you have read and understood the Matlab Background handout. We can discuss any questions arising next week
- Make sure you can do all the Exercises in this workshop
- If using “MATLAB for Behavioural Scientists” (Rosenbaum), make sure you have read and understand the content to the end of Chapter 2
- Have a look at Workshop 2 handout on Canvas ***before*** the next workshop

Matlab Workshop 1

Exercises

Revision: 28 Sep 2017

Exercises 1.1

- Have a play interacting with the Matlab environment
- Explore what is on each of the pull down menus
- See if you can understand what the different sub-windows do
- Try changing the current directory

Exercises 1.2

- Investigate the Matlab GUI and make sure you know where the different parts can be found. For example, using the GUI:
  - What (if any) variables are currently in your workspace?
  - What files are in your current directory?
  - What was your last but one command?

- Investigate the help available in matlab. Using the help:
  - Find the function for standard deviation
  - Find the function for variance
  - Read the help on the standard deviation and variance functions
  - Apply the above two functions to the matrix **a=[1:10]**
  - Do you get 3.0277 and 9.1667?

Exercises 1.3

- What is the equals symbol `=` used for?
- What are square brackets `[]` used for?
- What are round brackets `()` used for?
- What is the comma symbol `,` used for?
- What is the semi-colon symbol `;` used for?
- What is the colon symbol `:` used for?
- What is the apostrophe symbol `'` used for?
- What is the caret symbol `^` used for?

Exercises 1.4

- Create a row matrix A that increases in steps of 1 from 3 up to 99
- How many columns are in vector A?

- Create a column matrix B that decreases in steps of 0.5 from 333 down to -10
- What are the dimensions of matrix B?

- Create a 2d matrix called C of size 100 rows by 100 columns containing only the number 1
- Now create a 2d matrix called D of size 100 rows by 100 columns containing only the number 7

Exercises 1.4

- Add 2 to all the values in C and store it in a new matrix called X1. Divide all the values of D by 14 and store it in a matrix called X2. Add X1 and X2 together, divide the result by 2 (to create the mean of X1 and X2) and then store the result in variable Xmean. Check the size of Xmean is as expected.

- Try defining:
  E = [1, 2, 3, 4; 5 6 7]
  What did you do wrong?

- If I define F = [10:1] what would you expect **F** to be? Try it.
  Can you understand what has happened?

Exercises 1.5

- Recall the table of Student Test results you created called **Scores_All**
- What is the average score per Test?
- What is the average score per Student?
- What is the maximum score (over all students over all tests)?

Exercises 1.6

- What examples have you seen so far of **overloading** in Matlab?
- What examples have you seen so far of **implicit shortcutting** in Matlab?

Matlab Workshop 1

Suggestions for Answers to Exercises

Revision: 28 Sep 2017

Exercises 1.3

- What is the equals symbol **=** used for?
  Assigning the value on RHS of = to a variable on LHS
- What are square brackets **[ ]** used for?
  When creating an array they go around the contents
- What are round brackets **( )** used for?
  When making a function call they go around the arguments passed to the function
  (Note that you will see other uses for brackets in the next workshop)

Exercises 1.3

- What is the comma symbol `,` used for?
  (1) When creating an array, for separating out the elements within a row.
  (2) When used with function calls, to separate out the arguments to the function.

- What is the semi-colon symbol `;` used for?
  (1) When creating an array, for creating a new row.
  (2) To suppress output being shown in the command window

Exercises 1.3

- What is the colon symbol `:` used for?
  As a shortcut to create a regular sequence.

- What is the apostrophe symbol `'` used for?
  To transpose a matrix

- What is the caret symbol `^` used for?
  To raise to the power of (as in 10^2 = 10^2 = 100)

Exercises 1.4

- Create a row matrix A that increases in steps of 1 from 3 up to 99
  A = 3:99 or A = [3:99] or A = 3:1:99 or A = [3:1:99]

- How many columns are in vector A?
  length(A)
  >> ans = 97

- Create a column matrix B that decreases in steps of 0.5 from 333 down to -10
  B = [333:-0.5:-10]'

- What are the dimensions of matrix B?
  size(B)
  >> ans = 687 1

Exercises 1.4

- Create a 2d matrix called C of size 100 rows by 100 columns containing only the number 1
  C = ones(100,100)

- Now create a 2d matrix called D of size 100 rows by 100 columns containing only the number 7
  D = 6 + ones(100,100) or
  D = 7*ones(100,100) or
  D = C + C + C + C + C + C + C

Exercises 1.4

- Add 2 to all the values in C and store it in a new matrix called X1. Divide all the values of D by 14 and store it in a matrix called X2. Add X1 and X2 together, divide the result by 2 (to create the mean of X1 and X2) and then store the result in variable Xmean. Check the size of Xmean is as expected.

```
X1 = C + 2;
X2 = D / 14;
tmp = X1 + X2;
Xmean = tmp/2;
size(Xmean)

>> ans =   100   100
```

Exercises 1.4

- Try defining:
  E = [1, 2, 3, 4; 5 6 7]
  What did I do wrong?
  The 1st row of E has 4 elements (4 columns) but the 2nd row has only 3 elements (3 columns). That's not valid. A valid matrix must have an entry for each (row, col) entry with no gaps.

- If I define F = [10:1] what would you expect **F** to be? Try it.
  Can you understand what has happened?
  F = [ ] - the empty or null array. This has happened because we've asked Matlab to do something that is impossible – to create an array starting at value 10 and going in steps of +1 up to 1. But you can't go from 10 to 1 in +1 steps. Matlab realises there's a problem and returns its best effort - an empty array.
  What was probably intended was:
  F = [10: -1: 1]
  Other languages might return an error here. Matlab does not.

Exercises 1.5

- Recall the table of Student Test results you created called **Scores_All**

- What is the average score per Test?
  mean(Scores_All) or mean(Scores_All,1)

```
>> ans =  66.4000   64.0000   71.1000
```

Exercises 1.5

- What is the average score per Student?

  mean(Scores_All,2)

```
>> ans =
   72.6667
   52.6667
   48.3333
   78.6667
   67.3333
   66.6667
   58.0000
   73.3333
   70.3333
   83.6667
```

Exercises 1.5

- What is the maximum score (over all students over all tests)?

  max(Scores_All) or max(Scores_All,1)
  will find the max score by Test:

  ```
  >> ans =    82    83    87
  ```

  and then

  max(max(Scores_All)) or max(max(Scores_All,1))
  will find the single biggest score from these three:

  ```
  >> ans =   87
  ```

Exercises 1.6

- What examples have you seen so far of **overloading** in Matlab?

  (1) The semi-colon ; symbol. It can be used to create new rows in an array. Or it can be used to suppress display output in the command window

  (2) The comma , symbol. It can be used to create new columns in an array. Or it can be used to separate the arguments being passed to a Matlab function.

  (3) The use of round brackets ( ). So far you have seen these been used to envelope the arguments to a Matlab function. In the next workshop you will see they are also used for other purposes too.

Exercises 1.6

- What examples have you seen so far of **implicit shortcutting** in Matlab?

  (1) When generating a 1-d column array the comma symbol can be omitted.

  (2) When generating an array using the colon : operator if you do not wrap the array in square brackets, the square brackets are implicitly assumed to be present.

  (3) When generating an array using the colon : operator if you do not specify the step size it is implicitly assumed to be equal to 1.

  (4) When manipulating scalars and matrices Matlab implicitly resizes the scalar to be the size of the matrix, if it can.

Matlab Programming Module

Dr Peter Hansen
p.c.hansen@bham.ac.uk

School of Psychology / BUIC

Workshop 2
Matlab Concepts

Revision: 2 Oct 2017

Recap on brackets - [ ] vs ( )

Try and remember that round brackets **( )** and square brackets **[ ]** are used for different purposes in Matlab and cannot be used out of context. This is one of the most commonly confused aspects when using Matlab.

Square brackets are **only** ever used for **creating** arrays when you want to **assign** a set of values. Try the following commands:

```
x = [ 10 11 12 ]        ← OK
x = ( 10 11 12 )        ← Error!
```

In the first case the numbers 10, 11 and 12 are assigned to an array called **x**

In the second case an error is generated because this is not valid Matlab syntax. Matlab uses round brackets ( ) for several other purposes but they are **never** used for assigning values into an array.

Assigning Data I

You can assign variables within other variables. For example:

Define:

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

You can then set

```
z = [ x y ]
>> z =
        1 2 3 4 5 6
```

Do you recall from last weeks workshop what the semicolon operator **;** does?

If you assign:

```
z = [x ; y]
```

what would you expect to be the value of z? Try it and see.

Assigning Data II

As you saw last week you can apply simple arithmetic operations like + (add operator) and – (subtract operator) to variables:

```
a = 1
b = a + 1
>> b = 2
```

You can also apply arithmetic operations with numbers to matrices. The operation will act on **each and all** elements of the matrix:

```
x = [ 1 2 3 ]

x - 2
>> ans =
         -1 0 1

x + 10
>> ans =
        11  12   13
```

If you recall from last week, Matlab is performing an **implicit shortcut** here by allowing the scalar number to operate with the matrix without needing it to be a matrix of the same size i.e. we do not need to write code like this:

```
x – 2*ones(size(x))
>> ans =
        -1 0 1
```

Assigning Data III

You can apply arithmetic operations between matrices. For this to work, the matrices **must** be matched in size. The operation then acts one by one on each the **corresponding** elements of the matrices:

For example, if:

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

x + y
>> ans =
        5  7  9
```

i.e. [x_1+y_1 x_2+y_2 x_3+y_3] or [1+4 2+5 3+6]

```
y - x
>> ans =
       3  3  3
```

i.e. [y_1-x_1 y_2-x_2 y_3-x_3] or [4-1 5-2 6-3]

NOTE: If the matrices are of different size and Matlab cannot use an implicit shortcut (such as when one of the variables is a scalar) then an error will be generated.
Try:

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

Assigning Data IV

Multiplication and division are treated in a special way in Matlab. Matlab needs to know whether you intend to do standard matrix multiplication (or division) or whether you intend to multiply each element in one matrix by the corresponding element in the other.

For example, if:

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

Matrix multiplication uses just the standard multiply symbol (asterisk *****):

```
x*y'
>> ans = 32
```

Note the use of the transpose symbol here.
It is not a valid matrix multiplication to use
`x*y`
Try it!

Aside - Matrix Multiplication

If it is not obvious why x*y' equals 32 then remember that what you are really doing here is multiplying a row matrix by a column matrix. The default rule for this is:

$$\begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix} * \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = (x_1 * y_1 + x_2 * y_2 + x_3 * y_3)$$

So with x=[1 2 3], y=[4 5 6]:

x*y' = 1*4 + 2*5 + 3*6 = 32

Assigning Data IV

If you want to multiply each element in one matrix by the corresponding element in the other, then element-based multiplication uses the symbol “dot asterisk” or .*

The dot **.** operator is a special operator that is used in Matlab in the context of matrix operations.

It means something like 'apply by element'.

For example, if:

```
x = [ 1 2 3 ]
y = [ 4 5 6 ]
x.*y
>> ans = 4  10 18
```

This comes from:
[x_1*y_1 x_2*y_2 x_3*y_3]
=
[1*4   2*5   3*6]

Assigning Data V

Multiplication and division by a scalar number (not a matrix) are the same whether you use .* or just *
For example:

```
x*2
>> ans = 2 4 6
x.*2
>> ans = 2 4 6
```

This is true whether or not the number is 'hidden' inside a variable e.g.

```
n=2
x*n
>> ans = 2 4 6

x.*n
>> ans = 2 4 6
```

This may look like matrix multiplication but it isn't

Assigning Data VI

Some operations are ambiguous as to whether they are to be applied to each element within the matrix or to the whole matrix. To make the operation apply to each element the dot operator needs to be used in front of the other operator. For example with The power operator **^** :

```
m=[ 1  2; 3  4];
m.^2
>> ans = 1 4
         9 16
```

Square (raise to the power 2) each individual element of m

Compare this with:

```
m^2
>> ans = 7    10
        15    22
```

Square (i.e. multiply by itself) the entire matrix m

Note carefully here. With matrices:
m.^2 is **not** the same as **m^2**

However, note also that m^2 is the same as:

```
m*m
```

Aside - Matrix Multiplication II

If it is not obvious where the result comes from in calculating **m^2** or **m*m** then you need to appreciate the basic matrix multiplication rule:

m*n = $\begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} * \begin{pmatrix} n_{11} & n_{12} \\ n_{21} & n_{22} \end{pmatrix} = \begin{pmatrix} m_{11}*n_{11}+m_{12}*n_{21} & m_{11}*n_{12}+m_{12}*n_{22} \\ m_{21}*n_{11}+m_{22}*n_{21} & m_{21}*n_{12}+m_{22}*n_{22} \end{pmatrix}$

Or in the case when the second matrix is the same as the first:

m*m = $\begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} * \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} = \begin{pmatrix} m_{11}^2+m_{12}*m_{21} & m_{11}*m_{12}+m_{12}*m_{22} \\ m_{21}*m_{11}+m_{22}*m_{21} & m_{21}*m_{12}+m_{22}^2 \end{pmatrix}$

Overloading of ( ) Brackets

Round brackets are a source of confusion in Matlab as they appear in three different contexts. In each case they look the same but are used for a different purpose:

(1) Passing arguments to a function

(2) Arithmetic simplification

(3) Selecting data from an array

Matlab could have chosen to have 3 different symbol pairs for each of these 3 cases.

But as with many other programming languages they chose to re-use the same symbols for different situations.

Try to memorize each of these usages as being for a separate special case. Also – just because round brackets are used so often does **not** mean that they can be used in other contexts, such as creating arrays (where square brackets need to be used).

( ) Brackets – Function Calls

As you have already seen, round brackets are used in Matlab to pass variables into a function so that the function has something to work with.

For example:

```
k=1:10;
mean(k)
```

Here the function is called **mean**

( ) is the mechanism by which data is passed into the function

k is the **argument** (in the form of a variable called **k**) passed to it

```
>> ans = 5.5
```

( ) Brackets – Arithmetic Simplification

When you have a complicated arithmetic expression in Matlab you may want to ensure that it is evaluated in the way you expect. Matlab allows you to do this by putting in round ( ) brackets as needed.

For example:

```
a = (22 + 7) * (11 + 4)

b = 22 / (11 + 12)

c = (1 + 10)^2

d = 10^(1 + 2)
```

This corresponds exactly to how you would use ( ) brackets in standard mathematical algebra.

( ) Brackets – Selecting Data

Round brackets can be also used for **selecting** data from variables. This is a special Matlab use of the brackets. For example:

```
a = [ 7  3  1  8  ];
a(2)
```

will retrieve the 2nd element of **a**

```
>> ans = 3
```

Note: When selecting data from a Matlab array only ever use round ( ) brackets.

Never use square brackets!

```
b=0:5:50;
b(5)
```

will retrieve the 5th element of **b**

```
>> ans = 20
```

Or with multidimensional data:

```
c=[1 2 3 4; 5 6 7 8];
c(2,3)
```

Note the use of 2 arguments here (one for each dimension), used with a comma to separate them

will retrieve the element in the 2nd row and 3rd column of **c**

```
>> ans = 7
```

Matlab Functions

Matlab has a huge range of mathematical functions that can be applied to all elements of a matrix. You saw some of these in Workshop 1. For example:

```
mean(x)
median(x)
var(x)
sqrt(x)
```

Note again - all have round brackets after the function name.

Functions may take several arguments. For example the **rem()** function returns the remainder of its first argument after dividing by its second argument:

```
k = 1:10;
rem(k,3)
>> ans =    1    2    0    1    2    0    1    2    0    1
```

Selecting Data I

More advanced ways of selecting data. If:

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

At the moment you know how to select one item from the array .e.g.

```
b = a(2)
>> b = 4
```

Select only element 2

You can also select several items in one go, by using an array:

```
c=a([ 1 2 3 ])
>> c = 5 4 3
```

Select elements 1, 2 and 3 at same time

Or the same result as above could have been achieved in 2 steps:

```
idx = 1:3
d=a(idx)
>> d = 5 4 3
```

Predefine variable that contains numbers 1,2 and 3

Select only the elements in **a** whose values are in **idx**

Selecting Data II

Finally note carefully that the 1st element in any array is always numbered from 1 (and not 0 as in some languages). But the last element in an array could be anything, depending on how large the array is. There is a special symbol name for the last element of an array in Matlab. This name is **end**. For example if:

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

f=a(end)
>> f = 1

g=a(4:end)
>> g = 2  1

h=a(end-1)
>> h = 2
```

In these examples **end** is a special symbol meaning the last element in the array

Selecting Data III

You might think that one alternate way of selecting elements from one array could be the following:

b=[a(1) a(2) a(3)]

Select elements 1, 2 and 3 of **a**

This is perfectly legal Matab syntax. But it becomes unworkable for a large no of selections. Suppose that:

a= [1000000: -1: 1];

If we now need to select many elements it will be very tedious to have to write out:

b = [a(30) a(31) ...all the way up to... a(100)]

Avoid this!

Instead, it would be easier to define a subset of the array using:

idx = [30:100];
b = a(idx)

Defining your own index variable and then applying it to select the subset

or

b=a(30:100)

Doing the above operation in 1 line

Selecting Data IV

For selecting data from higher order matrices you need to specify as many numbers (separated by commas) as there are dimensions .e.g.

a=ones(3,3,3) + 1
b=a(2,3,1)
>> b = 2

This will retrieve the element in the 2nd position of dim1 (row), 3rd position of dim2 (col) and 1st position of dim3 (depth)

If one number is replaced by a colon **:** symbol then **all** of the corresponding row or column is selected:

a=[5 4 3 2 1; 10 9 8 7 6];
b=a(2, :)
>> b = 10 9 8 7 6

Select row 2, all columns

b=a(: ,3)
>> b = 3
        8

Select column 3, all rows

Selecting Data V

Data can also be selected by means of Boolean logic operations. e.g:

```
a = [ 5 4 3 2 1];
b = a>3
>> b = 1 1 0 0 0
```

If you don't know what Boolean logic is then it will help you to read up on this. Google it.

Each value in matrix **b** has been generated by applying the test “is the value >3” to each corresponding value in **a**. If the test is true then a value of true (1) is set in the corresponding position in **b**. If it is false, then a value of false (0) is set in **b**.

The previous line of code is basically the same as writing:

```
b = [ a(1)>3  a(2)>3  a(3)>3  a(4)>3  a(5)>3 ]
```

However, this form is much longer and very clunky to write out. Matlab is again performing an implicit shortcut here and in this example the benefit to us, in being able to write short concise code, is clear.

Logical Test Operations

Common logical test operators include:

Operator	**Alternative function(s)**	**Meaning**

==	eq()
	isequal()
	isequaln()
		Equal to

~=		Not equal to
>	gt()	Greater than
>=		Greater than or equal to
<	lt()	Less than
<=		Less than or equal to
~	not()	Logical not
|	or() **||**	Logical or
&	and() &&	Logical and

Selecting Data VI

Any logical test operation can be used to help select data e.g.:

```
a = [ 5 4 3 2 1];
b = a==3
>> b = 0 0 1 0 0
```

Test each value in **a** to see if it is equal to 3 and if so assign true (1), if not assign false (0) to each corresponding element in variable **b**

Note above that == means “is equal to”.
It is a Boolean test operator.

This is different to the = sign which means “assign value to”.

```
b = a~=3
>> b = 1 1 0 1 1
```

Note above that ~= means “not equal to”

Selecting Data VII

Values in matrix **a** can be selected by using matrix **b** to select them:

```
a = 1:10;
b = a>5
>> ans = 0 0 0 0 0 1 1 1 1 1
a(b)
>> ans = 6 7 8 9 10
```

Or these operations could be combined into a single step:

```
a=1:10;
a(a>5)
>> ans = 6 7 8 9 10
```

Or going one step further and applying a function to the result:

```
mean(a(a>5))
>> ans = 8
```

Do make sure you can follow this. The steps here may seem obvious one at time. But you will eventually need to be able to generate constructs like this yourself.

Selecting Data VIII

A related Matlab function, which often causes confusion, is the **find()** function:

```
a = [ 1  10  2  9  3  8  4  7  5  6]
idx=find(a>5)
idx = 2 4 6 8 10
a(idx)
>> ans = 10 9 8 7 6
```

Here the **find()** function returns those **positions** in matrix **a** where the logical test condition is satisfied. **idx** will only contain as many numbers as there are satisfied test conditions. If none, **idx** will be a null empty matrix [ ].

Contrast this with a simple logical test on **a**:

```
ltest = a>5
>> ltest = 0 1 0 1 0 1 0 1 0 1
```

ltest will always be the same size as **a**

```
a(ltest)
>> ans = 10 9 8 7 6
```

Make sure you are really clear on how the **find()** command works and how this is related to the previous logical test used to locate elements in an array.

Selecting Data VIII – Overloading

If you were keen eyed you might have spotted the subtle overloading that Matlab performed in doing data selection on the previous slides.

If we have:

```
a = [ 1  10  2  9  3  8  4  7  5  6 ]
idx = [ 2 4 6 8 10 ]
```

and

```
ltest = [ 0 1 0 1 0 1 0 1 0 1 ]
```

We saw:

```
a(idx)
```

and

```
a(ltest)
```

both generate:

```
>> ans = 10 9 8 7 6
```

Here Matlab is selecting from **a** using input **idx** where **idx** can be of any length (greater than 0) provided that each element of **idx** is in the range 1 to length(**a**).

Here Matlab is selecting from **a** using input **ltest** where **ltest** is of the same length as **a** and each element of **idx** is either true (1) or false (0).

If neither of these two specific selection choices are met then Matlab will generate an error. For example try these:

```
a( [ 2 4 6 11] )
ltest(10)=[ ]; a(ltest)
```

Why don't they work?

Selecting and Assigning Data

As with selection, using the colon : operator, assignment can be made to **all** of the elements in the dimension selected by the colon at the same time i.e. to all of one row or all of one column. For example:

```
a=ones(2,2);
a(1,:)=[3 4]
>> a =   3   4
         1   1
```

a(1,:) means select from **a** row 1, all columns

```
a=ones(2,2);
a(1,:)=7
>> a =  7    7
        1    1
```

Note here that Matlab is using implicit shortcutting to automatically assigned the number 7 to **all** the elements of the row

```
a(1,:) = [ 7 7 ]
```

You might have thought you needed to do this. It is correct too but the above form is simpler

Removing Data

Part of a matrix can be removed (deleted) by assigning empty values to it:

```
x = [ 1 2 3; 4 5 6 ]
>> x =     1     2     3
           4     5     6
```

Define an array with 2 rows and 3 cols

```
x(:,3)=[  ]
>> x =    1     2
          4     5
```

[ ] is the empty or null array

x(:,3) selects all rows, column 3 only.
This will remove the third column of the array

Selecting and Assigning Data

If we have a large matrix e.g.:
```
a=zeros(2,2,2);
```

a is now a 3-D matrix of size 2x2x2 full of zeros.

We can fill this in by selecting just those elements we want to substitute e.g.:

```
a(1,:,:)=ones(2,2)

a(2,:,:)=[ 11 12; 13 14 ]

a(1,2,3)=11
```

For next week

- Make sure you can do the Workshop Exercises. Ideally, play around with them and make sure you understand them.
- If using “MATLAB for Behavioural Scientists” (Rosenbaum), make sure you have read and fully understand to the end of Chapter 3
- Have a look at Workshop 3 handout on Canvas before next weeks workshop

Matlab Workshop 2

Exercises

Revision: 2 Oct 2017

Exercises 2.1

- What are square brackets **[ ]** used for?
- What are round brackets **( )** used for?
  (give 3 answers)
- What are semicolons **;** used for?
  (give 2 answers)
- What is the apostrophe symbol **'** used for?

Exercises 2.2

Clear the workspace using

```
clear all
```

Now define a data variable **x** as:

```
x = [ 10 11 12 ]
```

Predict what you think will happen in each of these cases:

(1) `x(1)`
(2) `x(3)`
(3) `x(10)`
(4) `x[1]`

Now test your predictions by typing the commands into Matlab.

Make sure you clearly understand what is happening in each case.

Exercises 2.3

- Create a matrix **m** that contains two rows of data. The top row are the numbers 4, 5 and 6 and the bottom row 9, 6 and 3.

- Create a matrix **n** that contains two columns of data. The 1st column are the numbers 1, 3, 5 and 7 and the 2nd column are 2, 4, 6 and 8.

- What are:

(1) `m.^2`
(2) `n.^2`
(3) `n*m`
(4) `m'*n'`

Exercises 2.4

- Clear the workspace
  clear all
- Create matrix **X**:
  X = [333:-3:3];
- How many rows and columns does **X** consist of?
- Create a new matrix called **Y** using **X** that has 3 times as many columns as **X** but is still only 1 row
- Check this is right with:
  size(Y) should return 1 333
- Create a new matrix called **Z** using **X** that has the same no of columns as **X** but which has 3 rows

  Check this is right with:
  size(Z) should return 3 111

Exercises 2.5

- Create the following 3x3 matrices:

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

  H = [11 12 13; 14 15 16; 17 18 19]

- Now replace column 1 of **G** with row 3 of **H**

Exercises 2.6

- Create matrix **J**:

  J = [1:10; 11:20; 21:30]

- Remove the last 5 columns of **J** and save the resultant matrix in a new variable called **K**

- Remove the first 2 rows of **J** and save the resultant matrix in a new variable called **L**

Matlab Workshop 2

Suggestions for Answers to Exercises

Revision: 2 Oct 2017

Exercises 2.1

- What are square brackets **[ ]** used for?

Data assignment.	e.g. X = [1 2 3 4]

- What are round brackets **( )** used for?

  (give 3 answers)

(1) Arithmetic simplification.	e.g. X = (1+2+3)*(4+5)
(2) Selecting data.	e.g. X(3)
(3) Function calls.	e.g. mean(X)

Exercises 2.1

- What are semicolons **;** used for?

  (give 2 answers)

  (1) When creating an array, for creating a new row.

  (2) When typing in the command window, to suppress output being displayed. This also allows multiple commands to be written on any one command line.

- What is the apostrophe symbol **'** used for?

  To transpose (swap the axes) of an array

  You will also find later it is used to create string data

Exercises 2.2

x = [10 11 12]

What will happen:

x(1)
10

x(3)
12

x(10)
An error is generated. We are attempting to get the 10th element of array x. But x has only 3 elements.

x[1]
An error is generated. This is not valid Matlab syntax.
Square brackets?

Exercises 2.3

- Create a matrix **m** that contains two rows of data. The top row are the numbers 4, 5 and 6 and the bottom row 9, 6 and 3.

  m = [4 5 6; 9 6 3]

- Create a matrix **n** that contains two columns of data. The 1st column are the numbers 1, 3, 5 and 7 and the 2nd column are 2, 4, 6 and 8.

  n = [1 2; 3 4; 5 6; 7 8]

  or

  n = [1 3 5 7; 2 4 6 8]'

Exercises 2.3 (cont)

- What are:

  (1) m.^2

```
>>  16   25   36
    81   36    9
```

  (2) n.^2

```
>>   1    4
     9   16
    25   36
    49   64
```

Exercises 2.3 (cont)

- What are:

(3) n*m

```
>>  22   17   12
    48   39   30
    74   61   48
   100   83   66
```

(4) m'*n'

```
>> 22   48   74  100
   17   39   61   83
   12   30   48   66
```

Exercises 2.4

- Create matrix **X**:

  X = [333:-3:3];

- How many rows and columns does **X** consist of?

  1 row and 111 columns as shown by size(X)

- Create a new matrix called **Y** using **X** that has 3 times as many columns as **X** but is still only 1 row

  Y = [X X X];

- Create a new matrix called **Z** using **X** that has the same no of columns as **X** but which has 3 rows

  Z = [X; X; X];

Exercises 2.5

- Create the following 3x3 matrices:

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

  H = [11 12 13; 14 15 16; 17 18 19]

- Now replace column 1 of **G** with row 3 of **H**

  G(:,1) = H(3,:)

Exercises 2.6

- Create matrix J:

  J = [1:10; 11:20; 21:30]

- Remove the last 5 columns of **J** and save the resultant matrix in a new variable called **K**

  J(:,end-4:end)=[ ]

  K = J

- Remove the first 2 rows of **J** and save the resultant matrix in a new variable called **L**

  J(1:2,:)=[ ]

  L = J

Matlab Programming Module

Dr Peter Hansen
p.c.hansen@bham.ac.uk

School of Psychology / BUIC

Workshop 3
More Matlab Concepts
Data Types

Revision: 17 Oct 2017* (updated since workshop)

More Matlab Concepts

Random Permutations

A permuted sequence of integer numbers can be generated using the **randperm()** function. **randperm(n)** creates a pseudo-random permutation of all the integers from 1 to n. For example:

```
r1 = randperm(10)
>> r1 =   1     7     5     2     6     4    10     3     8     9
```

Each time you run the **randperm()** function you will generate a different pseudo-random sequence:

```
r1 = randperm(10)
>> r1 =   6     3     7     8     5     1     2     4     9    10
```

However, note that the randomization is not truly random as it relies on underlying predictable algorithms. That is why the sequence is described as being pseudo-random rather than being truly random. We will follow up the implications of this shortly.

Uniformly Distributed Random Numbers

A set of uniform distributed pseudo-random decimal numbers can be created by using the **rand()** function. For example:

```
r2 = rand(1000,1)
```

This will generate a 1000x1 column array of pseudo-random numbers that are uniformly distributed in the range between 0.0 and 1.0.
As with the **randperm()** function, if you run the command multiple times you will get different pseudo-random uniform distributions.

A uniform distribution is one that looks flat across its range. You can check it looks uniform with a histogram plot:

```
hist(r2)
```

Uniformly Distributed Integer Random Numbers

A more useful special case of selecting numbers from a uniform random distribution is when we only want the numbers to be integers.

There is a special function for this in Matlab called **randi()** (note that the 'i' stands here for 'integer'). The syntax for this function has one more initial parameter than for **rand()** alone as we also need to tell it what the maximum integer size will be. For example:

```
r3 = randi(10,1000,1)
```

The first parameter (10) here tells the function to only generate integers in the range 1:10.

This will generate a 1000x1 column array of pseudo-random **integer** numbers that are uniformly distributed in the range between 1-10. Again you can check that it looks uniform and has the correct range with a histogram plot:

```
hist(r3,1:10)
```

Normally Distributed Random Numbers

A set of Normally/Gaussian distributed pseudo-random decimal numbers can be created by using the **randn()** function. For example:

```
r4 = randn(1000,1)
```

The **n** in **randn()** stands for Normal distribution.

This will generate a 1000x1 column array of pseudo-random numbers that are Normally/Gaussian distributed with mean 0 and standard deviation 1. As with the **randperm()** function, if you run the command multiple times you will get different pseudo-random Normal distributions.

A Normal/Gaussian distribution is one that looks bell-shaped across its range. You can check that it looks like this with a histogram plot:

```
hist(r4)
```

Randomization I

Often we want to ensure that the randomization we generate using Matlab's various random routines is **repeatable**. This may seem counter-intuitive. But if we are debugging code or running numerical simulations it is important that we can rerun the same code and generate exactly the same results. This can be achieved by seeding the internal Matlab RNG (Random Number Generator) with a particular fixed starting value. For example:

```
rng(1001)
randperm(5)
>> ans = 5    3    2    1    4
```

Seed value. Here I chose 1001 but it could be any non-negative value. What is important is that it is a **known** value so that we can reset the RNG to the **same** known value if needed.

Running the above again:

```
rng(1001)
randperm(5)
>> ans = 5    3    2    1    4
```

We see here that having reset the RNG with the same seed value, we do indeed generate the same “random” output again.

Randomization II

However, we may instead want to ensure that the randomization we generate using Matlab's various random routines is as **variable** as possible. This is the opposite of what we wanted to achieve in the previous slide. For example, we may be running an experiment in Matlab with code to permute the order in which stimuli are presented. We might want to do our best to ensure that no two runs will have an identical sequence of stimuli. This can be achieved by seeding the internal Matlab RNG with a starting value that is always different. This can be achieved with:

```
rng('shuffle')
```

This sets the RNG based on the current time. If this command is put at the beginning of your Matlab code then whenever your program runs it will randomize things differently.

Note: Users of older versions of Matlab may remember achieving the above result using commands such as **rand('state',x)** or **rand('seed',x)** with x being some number returned by the **clock()** function. *This form of usage is now deprecated and should not be used*.

Clock function

The Matlab **clock()** function can be used to access the current time and date:

```
c = clock()
```

This returns a six element vector containing the current date and time in the form:

[year month day hour minute seconds]

Although it is not obvious, the first five elements will always be whole numbers (year for example can only ever be an integer). The sixth element (seconds) is a decimal number and is accurate to many digits beyond the decimal point.

For a more readable display (rounding to the nearest second) use the rounding function **fix()**:

```
c= fix(clock())
>>   2012        10        10        17        40        27
```

For more info, use `help clock`

Pearson Correlation Coefficient

The **corrcoef()** function is an example of a slightly more sophisticated Matlab function. It can be used in various ways (see **help corrcoef**). Here is an example:

Remember what the rand() function does?

```
a = rand(100,1);
b = a/2 + rand(100,1);
plot(a,b,'.')
[r,p] = corrcoef(a,b)
```

From this, **a** and **b** should be correlated
Here we create a scatterplot of **a** vs **b**
Notice here we return 2 values, not 1

r is a matrix containing the correlation values: r(1,2) will be the correlation between a and b.

p is a matrix containing the significance (p) values: p(1,2) will be the p-value for the correlation between a and b.

Sorting Data I

The simplest use is with the basic **sort()** function. For example:

```
a = [ 10 4 2 8 ]
sort(a)
>> ans = 2 4 8 10
```

Look at the help on **sort()** for more details

Or we could sort in reverse order:

```
sort(a,'descend')
>> ans = 10 8 4 2
```

Note that **sort()** takes the extra argument about which direction to sort in as a string 'ascend' or 'descend'

Can also sort each column in a matrix using a single command:

```
b = [ 5 2; 4 3; 4 2; 1 2; 5 1; 2 4; 4 1; 5 3; 3 1; 1 1]
sort(b)
```

Note that the two columns have been sorted **independently** so any previous relationship between rows has been lost

Sorting Data II

To preserve the relationship between rows when sorting, rather than **sort()**, use the **sortrows()** function:

```
sortrows(b)
```

Sort rows by default (ascending) order

```
sortrows(b,1)
```

Sort by column 1 in ascending order

```
sortrows(b,-2)
```

Sort by column 2 in descending order

Look at the help on **sortrows()** for more details.

Note that **sortrows()** takes an extra argument about which column to sort as a **number** rather than as a string 'ascend' or 'descend' as with **sort()**. If the number is positive the sort is ascending. If the number is negative, the sort is descending.

```
sortrows(b,[1 -2])
```

Here the extra argument is an array, with as many columns as there are in b. This means: sort **b firstly** by ascending order in column 1, then **secondly** (when there are two or more tied identical values in column 1) by descending order in column 2

Rounding Data I

Rounding to the nearest integer (round down if <0.5, round up if >=0.5) uses the **round**() function:

```
c = 10*rand(10,1)
round(c)
```

Rounding always ***down*** to the nearest integer uses **floor()**:

```
floor(c)
```

Rounding always ***up*** to the nearest integer uses **ceil**() (as in short for 'ceiling'):

```
ceil(c)
```

Rounding Data II

If rounding towards zero also includes negative numbers then **floor()** may not behave as desired as it will round negative numbers to be ***more*** negative (away from zero).

For example:

```
floor(-5.5)
>> ans = -6
```

Instead, Matlab has the **fix()** function:

```
fix(-5.5)
>> ans = -5
```

To see the difference between the functions look at:

```
d = [c floor(c) fix(c) -c floor(-c) fix(-c)]
```

Missing Data I

Matlab has a special value to indicate missing or invalid data.
This value is known as **NaN** (short for Not-a-Number)

NaN may represent a missing data element in a data set.
Imagine we have 4 people and some incomplete measures on them:

```
age = [ 30    22   18    33  ]
sex = [   1      2     1      2  ]
IQ  =  [ 120 130 NaN 140 ]
```

Age in years
coded as 1=female, 2=male
Scaled IQ measure

NaN will also be returned by functions that return an indeterminate value. Try:

```
v = 0/0
v = NaN
```

Missing Data II

Matlab has the function **isnan()** to check if a value is NaN:

```
isnan(v)
>> ans = 1
isnan(10)
>> ans = 0
```

In Matlab, a 1 is the same as **true**

In Matlab, a 0 is the same as **false**

The stats toolbox in Matlab also has a set of NaN functions that work on matrices containing NaN values in a similar way to their non-NaN counterparts. e.g.

```
nanmean(IQ)
>> ans = 130
```

nanmean() calculates the mean of the array excluding any elements that have value **NaN**

If you try and apply standard functions to data containing NaNs e.g.:

```
mean(IQ)
>> NaN
```

you will find a problem - it will just return **NaN**

Missing Data III

There are therefore other NaN Matlab functions that work with NaNs such as:

nanmax(IQ)	Max excluding NaNs
nanmean(IQ)	Mean excluding NaNs
nanmedian(IQ)	Median excluding NaNs
nanmin(IQ)	Min excluding NaNs
nansum(IQ)	Sum excluding NaNs
nanstd(IQ)	Standard Dev excluding NaNs
nanvar(IQ)	Variance excluding NaNs

Try running:

```
nanmean(IQ)
```

More high level functions may give you different options for NaNs

Infinities

Matlab also has a special value to indicate infinity. This value is **inf** and will be returned by operations that inadvertently generate infinities.

```
a = [ 1 2 3 4]
b = [ 1 1 0 1]
a./b
>> ans = 1     2   Inf     4
```

There is also a function **isinf()** to check if a value is inf:

```
isinf(a./b)
>> ans = 0 0 1 0
```

Note again that 1 = true, 0 = false

Matlab Basic Data Types

Different Types of Data I

- So far most data we have seen has been in the form of numbers.
- Matlab can represent numbers internally in different ways. You can choose different number **types** when defining a numeric variable.
- The number may be a decimal number (such 2.718). These are known as **floating point** numbers. Floating point numbers can have values other than integers and can be positive or negative.
- Floating point numbers can also be represented with more or less precision (accuracy) depending on how storage memory is set aside to represent each numeric variable. The more memory (precision), the more decimal point values can be encoded. Similarly with more memory very much larger or smaller numbers can be encoded.

Different Types of Data II

- By default, in Matlab, numbers are of type **double** (double precision floating point). These are the most accurately represented floating point numbers. Define:
  d = double(2/3)

  Because double is the default, this is the same as
  d = 2/3

- Numbers can also be of type **single** (single precision floating point). They are less precise than **double** but can be used to save memory (or disk space when stored) when you do not need high accuracy. Define:
  s = single(2/3)

  Now try:
  format long
  d
  s
  format

  Tells Matlab to use long form decimal display format

  This shows the double version **d** of 2/3 stores more information (further down in the decimal points) than the single version **s**

  Tells Matlab to revert back to default display format

Different Types of Data III

- Numbers can also be **integers** rather than floating point. Integers are whole numbers with no decimal parts (e.g. 1 or 7).
- Integers can be **signed** (positive or negative whole numbers) or **unsigned** (positive whole numbers only)
- Integers can be of different size, depending on how much memory we need to allocate per variable. The sizes are 1, 2, 4 or 8 bytes (a byte is 8 bits). The bigger the byte size, the larger the maximum number represented can be, but the more memory it takes up.
- It makes sense to choose a particular integer type if we know what size and range of data we want to represent and if we want to specifically constrain numbers to be only integers, or only positive.

Different Types of Data IV

Besides the different sorts of numbers, Matlab has some other data types too. The main other ones you will encounter are:

- **char type (character) and string type (array of chars)**
  There is only one sort of char type. Chars are easy to recognise as they are always defined between pairs of single quotes e.g.:
  ```
  letter = 'a'
  string = 'hello'
  ```
  Note the single quote symbol is the same symbol used to transpose an array. This is another example of Matlab reusing the same symbol in different contexts.

- **Boolean type (logical)**
  There is only one sort of logical type – either **true** or **false**. e.g.
  ```
  test = true
  ```
  Note that **true** and **false** are special predefined words in Matlab. 1 and 0 can also be substituted for true and false when working with variables of type Boolean.

Different Types of Data V

```
clear all

a = 1
b = 1.00
c = 1.4e-12
d = single(b)
e = double(d)
f = int16(b)
g = uint32(b)

A = 'z'
B = 'abcd'
C = true
D = [true false false true]
```

Enter the following variables

The results of these will be reviewed on the next few slides

Different Types of Data VI

```
whos
>>  Name      Size          Bytes  Class      Attributes

>>   A        1x1            2  char
>>   B        1x4            8  char
>>   C        1x1            1  logical
>>   D        1x4            4  logical

>>   a        1x1            8  double
>>   b        1x1            8  double
>>   c        1x1            8  double
>>   d        1x1            4  single
>>   e        1x1            8  double
>>   f        1x1            2  int16
>>   g        1x1            4  uint32
```

Different Types of Data VII

a, b, c and e are numbers of type double (double precision). Each uses 8 bytes of memory.

d is a number of type single (single precision). It uses 4 bytes of memory.

f is a number of type int (signed integer) and of size 16 (bits). It uses 2 bytes of memory.

g is a number of type uint (unsigned integer) and of size 32 (bits). It uses 4 bytes of memory.

Different Types of Data VIII

A is a variable containing a single char (character).
It uses 2 bytes of memory.

B is an array of 4 chars. This is also known as a **string**.
It uses 8 bytes of memory.

C is a variable containing a single Boolean (logical value).
It uses 1 byte of memory.

D is an array of 4 Booleans.
It uses 4 bytes of memory.

Different Types of Data IX

It is easy to convert between different data types.

Possible conversion commands can be found with:

```
help datatypes
```

Each data type has a function of the same name associated with the data type. This function is used to convert **to** that data type. For example:

```
double(10)
single(10)
int16('abcd')
uint8([true false])
logical(1)
char(68)
```

Different Types of Data X

Here is a fuller listing of the different integer conversion functions (and their associated data type):

uint8()	Convert to unsigned 8-bit integer	(range 0 – 255)
uint16()	Convert to unsigned 16-bit integer	(range 0 – 65535)
uint32()	Convert to unsigned 32-bit integer	(range 0 – 4×10^{9})
uint64()	Convert to unsigned 64-bit integer	(range 0 – 18×10^{18})
int8()	Convert to signed 8-bit integer	(range -128 – 127)
int16()	Convert to signed 16-bit integer	(range -32768 – 32767)
int32()	Convert to signed 32-bit integer	(range -2×10^{9} – 2×10^{9})
int64()	Convert to signed 64-bit integer	(range -9×10^{18} – 9×10^{18})

Different Types of Data XI

The particular data type for a given variable can be learned by using the **class()** function:

```
class(A)
>> ans = char

class(C)
>> ans = logical

class(d)
>> ans = single
```

Different Types of Data XII

There are various Matlab functions, fairly logically named, to test whether data is of a particular data type. For example:

isnumeric()	- True if numeric
isfloat()	- True if floating point (single and double)
isinteger()	- True if integer (of any type)
islogical()	- True if logical
ischar()	- True if char

```
isnumeric(C)
>> ans = 0        (false)
isnumeric(d)
>> ans = 1        (true)
isinteger(d)
>> ans = 0        (false)
isfloat(d)
>> ans = 1        (true)
```

Note again here the equivalence between 0 and false, and 1 and true.

In older versions of Matlab there was no specific logical data type and integers 0 and 1 were used in their place. They can still be used interchangeably.

Don't let this confuse you.

0 is the same as false and 1 is the same as true.

String Manipulation I

The simplest way to use strings in Matlab is as a regular row array of characters (type char). Strings should be assigned using single quotes:

```
a = 'Hello world!'
```

a is a 1x12 row array

Do not use double quotes to define char arrays:

```
z = “Hello world!”
```

This will appear to work but really it creates a new Matlab data type introduced into Matlab R2016b called, confusingly, a “string array”. **z** is here a size 1x1 object, not the same as **a**. Different functions are used to manipulate string arrays. We won't discuss this data type further as it is not backwards compatible with earlier Matlab versions or with the char arrays we are using here.

Char arrays can be concatenated like any other matrix using **[ ]**:

```
b = '   ';  c = 'This is a test.';
d = [ a b b b b c ]
>> 'Hello world!    This is a test.'
```

```
d = strcat(a,b,c,b)
>> 'Hello world!This is a test.'
```

This has almost the same effect as the **strcat()** function except **strcat()** also removes any blank spaces that are present.

String Manipulation II

Just as with arrays containing numbers, values in string arrays can be selected using round ( ) brackets. For example:

```
a(7:11)
>> ans = 'world'
```

Or the values can be overwritten:

```
a(7:end) = 'earth!'
a
>> 'Hello earth!'
```

Or the values can be removed:

```
a(1:6) = [ ]
a
>> 'earth!'
```

String Manipulation III

It is possible to create a 2d matrix of strings. However, all the strings need to be the same length.

```
a = 'Hello world!'
b = 'Hello Peter!'
x = [a ; b]
```

See what happens if the strings are of different lengths:

```
c = 'Goodbye'
x = [a; c]
```

Error

Just like with numeric matrices, matrix elements sizes must match.

It is possible to clunkily get around this by padding strings with spaces:

```
c = 'Goodbye     '
x = [a; c]
```

Need to have exactly 5 spaces here after final 'e'

String Manipulation IV

Matlab has an inbuilt function to do this without you having to work out exactly how many spaces to pad each string with. The function Is called **strvcat()** and it automatically pads each string with spaces In order to form a valid matrix:

```
a = 'Hello'
b = 'Goodbye'
c = 'OK'
x = strvcat(a,b,c)
```

This is the same as:

```
x = ['Hello  '; 'Goodbye'; 'OK     ']
```

5 spaces here

2 spaces here

In **strvcat()** each text input parameter can itself be a string matrix:

```
y = strvcat(x,'Yes',x,'No')
```

String Manipulation V

There are many functions that work on strings. For example:

strfind(S1, S2) will search inside string S1 and return the starting indices of any occurrences of the search string S2

```
s = 'How much wood would a woodchuck chuck?';
strfind(s,'wood')
>> ans =10 23
```

findstr(S1, S2) is similar to **strfind()**, but initially it finds the shorter of S1 and S2, before returning the starting indices of it inside the longer string. This means the order of the arguments does not matter:

```
findstr(s,'wood')
>> ans = 10 23
findstr('wood',s)
>> ans = 10 23
```

String Manipulation VI

strcmp() will compare two strings, case-sensitively, and return logical 1 (true) if they are identical or logical 0 (false) otherwise:

```
a = 'Good day'
b = 'good day'
strcmp(a,b)
>> ans = 0        (false)
```

strcmpi() will compare two strings, ignoring case, and return true if they are identical or false otherwise:

```
strcmpi(a,b)
>> ans = 1        (true)
```

String Manipulation VII

strncmp() will compare the first **n** chars of two strings, case-sensitively, and return true if they are identical or false otherwise:

```
a = 'Good day'
b = 'good dog'
strncmp(a,b,4)
>> ans = 0        (false)
```

strncmpi() will compare the first **n** chars of two strings, ignoring case, and true if they are identical or false otherwise:

```
strncmpi(a,b,4)
>> ans = 1        (true)
```

Data Type Conversion I

One source of confusion is that Matlab data of type char also have numerical equivalents known as ASCII codes. (Google “ASCII codes” in a browser for more info if this is completely new to you).

The **double()** command can reveal the underlying ASCII codes for chars. Applying it a number variable has no effect:

```
double(4.23)
>> ans = 4.2300
```

Whereas on a char or string:

```
double('H')
>> ans = 72
double('Hello world')
>> ans = 72  101  108  108  111  32  119  111  114  108  100
```

72 is the ASCII code for H

For a full list see: www.asciitable.com

This may seem rather counter-intuitive.

Data Type Conversion II

Based on the previous slide, be very careful when working with chars and strings (char arrays) not to inadvertently use numeric functions on them unless you intend to. Usually this will be a mistake.

Doing so may accidentally create a Matlab expression that has legal Matlab syntax but almost certainly does not do what you intend. This is because there may be an implicit data conversion going on to make the numeric function work. This is an example of Matlab not helping us!

By way of illustration, run this:

```
mean('Hello world')
>> ans = 98.5455
```

Why is the **mean()** of this string 98.5? Make sure you really understand why this strange result happens.

Data Type Conversion III

While **double()** can show the numbers associated with characters (ASCII codes), the **char()** command performs the reverse conversion and shows the character equivalent of ASCII codes. For example:

```
a = [ 72 101 108 108 111 32 80 101 116 101 114  33 ];
char(a)
>> ans = 'Hello Peter!'
```

Be aware that a variable of type char may contain the *character* for a number but that is <u>not</u> the same as the actual number (as it is not a variable of type number). A numeric conversion of that variable will show something else:

```
a = '1'
double(a)
>> ans =  49
```

Make sure you understand this result.
The ASCII code for 1 is 49.

Data Type Conversion IV

This is a common problem so there is a separate function in Matlab specifically to convert numbers in variables of data type char into data type numeric. The function is called **str2double()**:

```
a = '1'
double(a)
>> ans =  49
str2double(a)
>> ans = 1
```

The reverse function, to convert numbers into string type, is called **num2str()**:

```
num2str(1)
>> ans = '1'
```

For next week

- Make sure you can do the Exercises here
- If using “MATLAB for Behavioural Scientists” (Rosenbaum), make sure you read and understand up to the end of Chapter 4
- Have a look at Workshop 4 handout on Canvas before next weeks workshop

Matlab Workshop 3

Exercises

Revision: 09 Oct 2017

Exercises 3.1

- Create an array r1 of size 1x1000 (1 row, 1000 cols) consisting of the numbers 1 to 1000 in a random order

- Create an array r2 consisting of 1x1000 random numbers sampled from a uniform distribution in the range from 1.0 to 2.0.
  Plot the histogram of this and verify r2 is as specified by visual inspection of the data

- Create an array r3 consisting of 1x1000 random numbers sampled from a Normal distribution with mean 10.0 and std dev 2.0.
  Plot the histogram of this and verify r3 is as specified by visual inspection of the data

Exercises 3.2

- Sort r1 firstly in ascending and then in descending numerical order. Repeat this exercise with r2, then r3

- Create the matrix:
  `r4 = [ 4:-1:1 1:4 4:-1:1 ; 1:6 6:-1:1 ; 1:12 ]';`

  Note the final transpose operator. Check this is a 12x3 matrix. Now, keeping the row order intact and using only a single Matlab command, sort r4 by:
  (i) descending order of column 1,
  (ii) ascending order of column 2,
  (iii) ascending order of column 3

Exercises 3.2

- Using the **randi()** function, create a matrix r5 consisting of 10,000 integer random numbers sampled from a uniform distribution in the range from 1 to 10.

  Verify this by plotting the data and visual inspection.

- Now repeat the above without using the **randi()** function. Using only the **round()** function, create a matrix r6 consisting of 10,000 integer random numbers sampled from a uniform distribution in the range from 1 to 10.

  Again, verify this by plotting the data and visual inspection.

Exercises 3.3

- What is the mean and standard deviation of:
  `m = [1:10 NaN 11 4 20 10 50];`
- Calculate using a logical expression how many non-NaN numbers are in matrix m
- Is one NaN equal to another NaN?
  Test this with a logical expression.
- Is one Inf equal to another Inf?
- What do you get if you divide a NaN by infinity?

Exercises 3.4

- Clear the workspace and load some sample data:
  ```
  clear all
  load count.dat
  ```
  (variable count will be loaded)
- Calculate the Pearson correlation coefficient between columns 1 and 2 of variable **count**. What is the p value of this correlation?
- Is this the largest correlation?
  Or is the col1 vs col3 or col2 vs col3 correlation bigger?

Exercises 3.5

- Clear the workspace. Create an array (single row) called **A** of type double with numbers 77, 97, 116, 108, 97 and 98
- Convert **A** into **B**, an array of unsigned 8 bit integers containing the same numbers
- Convert **B** into **C**, an array of chars converted from **B**
- Define:
  D = 1 + 3i
  Using a function call, what is the data type of **D**?
- Define:
  E = 17
  Using a function call, is **E** a floating point number?

Exercises 3.6

- Clear the workspace and define:
  s1 = 'Matlab '
  s2 = 'I '
  s3 = 'love '
- Use strings s1, s2 and s3 to create a new string (1d array of type char) s4 that contains the phrase 'I love Matlab'
- Combine the strings s1,s2 and s3 into a new 2d matrix of strings called s5

Exercises 3.7

- Clear the workspace and create a new string s1 containing the phrase 'I really love Matlab'
- Transform s1 into s2 by replacing 'love' with 'hate' so that s2 contains the phrase 'I really hate Matlab'
- Transform s1 into s3 by replacing 'really' with 'totally' so that s3 contains the phrase 'I totally love Matlab'
- Create an uppercase version of s1, called s4, by replacing all letters in s1 with their uppercase equivalents. Hint: **lookfor** a function that might do this.

Exercises 3.8

- Define:
  s = 'How much wood would Woody Woodchuck chuck?';
- Find all the occurrences in s where the char 'w' or the char 'W' occurs
- Using this result, calculate the no of times that 'w' or 'W' occurs in the string **s**
- Calculate the mean distance between occurrences of 'w' or 'W' in the string **s**. Hint: use the **diff()** function. If you don't know what this function does, look up the help on it.

Matlab Workshop 3

Suggestions for Answers to Exercises

Revision: 09 Oct 2017

Exercises 3.1

- Create an array r1 of size 1x1000 consisting of the numbers 1 to 1000 in a random order
  r1 = randperm(1000);

- Create an array r2 consisting of 1x1000 random numbers sampled from a uniform distribution in the range from 1.0 to 2.0
  r2 = 1 + rand(1,1000);

  Plot the histogram of this and verify r2 is as specified by visual inspection of the data
  hist(r2)

Exercises 3.1 (cont)

- Create an array r3 consisting of 1x1000 random numbers sampled from a Normal distribution with mean 10.0 and std dev 2.0
  `r3 = 10 + 2.0.*randn(1,1000);`

  Plot the histogram of this and verify r3 is as specified by visual inspection of the data
  `hist(r3)`

Exercises 3.2

- Sort r1 firstly in ascending and then in descending numerical order. Repeat this exercise with r2, then r3

```
sort(r1)        or      sort(r1,'ascend')
sort(r1,'descend')

sort(r2)        or      sort(r2,'ascend')
sort(r2,'descend')

sort(r3)        or      sort(r3,'ascend')
sort(r3,'descend')
```

Exercises 3.2 (cont)

- Create the matrix:
  `r4 = [ 4:-1:1 1:4 4:-1:1 ; 1:6 6:-1:1 ; 1:12 ]';`

  Note the final transpose operator.
  Check this is a 12x3 matrix.

```
4   1   1
3   2   2
2   3   3
1   4   4
1   5   5
2   6   6
3   6   7
4   5   8
4   4   9
3   3   10
2   2   11
1   1   12
```

- Now, keeping the row order intact and using only a single Matlab command, sort r4 by:
  (i) descending order of column 1,
  (ii) ascending order of column 2,
  (iii) ascending order of column 3
  `sortrows(r4, [-1 2 3] )`

```
4   1   1
4   4   9
4   5   8
3   2   2
3   3   10
3   6   7
2   2   11
2   3   3
2   6   6
1   1   12
1   4   4
1   5   5
```

Exercises 3.2 (cont)

- Using the **randi()** function, create a matrix r5 consisting of 10,000 integer random numbers sampled from a uniform distribution in the range from 1 to 10.

  `r5 = randi(10,1,10000)`

- Verify this by plotting the data and visual inspection.
  `hist(r5)`

Exercises 3.2 (cont)

- Now repeat the above <u>without</u> using the **randi()** function. Using only the **round()** function, create a matrix r6 consisting of 10,000 integer random numbers sampled from a uniform distribution in the range from 1 to 10.

  r6 = round(10*rand(1,10000) +0.5)

  Why does this work?

rand(1,10000)	gives floating point numbers between 0 to 1
10*rand(1,10000)	gives floating point numbers between 0 to 10
10*rand(1,10000) + 0.5	gives floating point numbers between 0.5 to 10.5
round(10*rand(1,10000) + 0.5)	gives integers between 1 and 10

- Again, verify this by plotting the data and visual inspection.
  hist(r6)

Exercises 3.2 (further explanation)

You may find the previous result difficult to follow. You might think that if **rand()** returns numbers in the range 0 to 1 then the maximum value of **rand(1,10000)** should be **1**. In which case the max value of **10*rand(1,10000)** should be **10** and the max value of **10*rand(1,10000)+0.5** should be **10.5**. At which point applying the **round()** function should round this value up to **11**. Which we do not want.

Why does this not happen? The answer relies on subtle understanding of how Matlab works. Although **rand()** does returns numbers in the range 0-1 because this is a continuous distribution it will <u>not</u> return the actual end values of 0 or 1 themselves. The max value will instead be a value very close to 1 (like 0.999999999) but not 1 itself. Then:

rand(1,10000)	will have a max value of 0.9999999
10*rand(1,10000)	will have a max value of 9.9999999
10*rand(1,10000) + 0.5	will have a max value of 10.499999999
round(10*rand(1,10000) + 0.5)	will round **down** this max value to **10**

Exercises 3.3

- What is the mean and standard deviation of:
  m = [1:10 NaN 11 4 20 10 50];

  nanmean(m)   OR   mean(m(~isnan(m)))
  >> ans = 10

  nanstd(m)   OR   std(m(~isnan(m)))
  >> ans = 12.0178

- Calculate using a logical expression how many non-NaN numbers are in matrix m
  sum(~isnan(m))
  >> ans = 15

Exercises 3.3 (cont)

- Is one NaN equal to another NaN?
  Test this with a logical expression.
  NaN==NaN
  >> ans = 0   (false)
  No, NaNs are not equal

- Is one Inf equal to another Inf?
  Inf==Inf
  >> ans = 1   (true)
  Yes, Infs are equal

- What do you get if you divide a NaN by infinity?
  NaN/Inf
  >> ans = NaN
  Another NaN

Exercises 3.4

- Clear the workspace and load some sample data:
  ```
  clear all
  load count.dat
  ```
  (variable count will be loaded)

- Calculate the Pearson correlation coefficient between columns 1 and 2 of variable **count**.
  ```
  [r,p] = corrcoef(count(:,1),count(:,2));
  r(1,2)
  >> ans =  0.9331
  ```

  What is the p value of this correlation?
  ```
  p(1,2)
  >> ans =  3.0231e-11
  ```

Exercises 3.4 (cont)

- Is this the largest correlation?
  Or is the col1 vs col3 or col2 vs col3 correlation bigger?

  ```
  [r,p]=corrcoef(count);
  r
  ```

  Visual inspection of the matrix r containing the correlation values shows r(1,3) (the col1 vs col3 correlation) to have the biggest value at 0.9599.

Exercises 3.5

- Clear the workspace. Create an array (single row) called **A** of type double with numbers 77, 97, 116, 108, 97 and 98
  clear all
  A = [77 97 116 108 97 98];

- Convert **A** into **B**, an array of unsigned 8 bit integers containing the same numbers
  B = uint8(A)
  >> B =   77   97 116 108   97   98

- Convert **B** into **C**, an array of chars converted from **B**
  C = char(B)
  >> C = Matlab

Exercises 3.5 (cont)

- Define:
  D = 1 + 3i
  Using a function call, what is the data type of **D**?
  class(D)
  >> ans = double

- Define:
  E = 17
  Using a function call, is **E** a floating point number?
  isfloat(E)
  >> ans = 1   (true)

Exercises 3.6

- Clear the workspace and define:
  s1 = 'Matlab '
  s2 = 'I '
  s3 = 'love '

- Use strings s1, s2 and s3 to create a new string (array of type char) s4 that contains the phrase 'I love Matlab'
  s4 = [s2 s3 s1]
  >> s4 = I love Matlab

- Combine the strings s1,s2 and s3 into a new 2d matrix of strings called s5
  s5 = strvcat(s1, s2, s3)
  >> s5 =

```
              Matlab
              I
              love
```

Exercises 3.7

- Clear the workspace and create a new string s1 containing the phrase 'I really love Matlab'
  clear all
  s1 = 'I really love Matlab'

- Transform s1 into s2 by replacing 'love' with 'hate' so that s2 contains the phrase 'I really hate Matlab'
  s2 = s1;
  s2(10:13)='hate'
  or
  s2 = [s1(1:9) 'hate' s1(14:end)]

  >> s2 = 'I really hate Matlab'

Exercises 3.7 (cont)

- Transform s1 into s3 by replacing 'really' with 'totally' so that s3 contains the phrase 'I totally love Matlab'
  s3 = [s1(1:2) 'totally' s1(9:end)]
  >> s3 = 'I totally love Matlab'

- Create an uppercase version of s1, called s4, by replacing all letters in s1 with their uppercase equivalents. Hint: **lookfor** a function that might do this.

  lookfor uppercase
  >> UPPER Convert string to uppercase

  s4 = upper(s1)
  >> s4 = 'I REALLY LOVE MATLAB'

17

Exercises 3.8

- Define:
  s = 'How much wood would Woody Woodchuck chuck?';

- Find all the occurrences in s where the char 'w' or the char 'W' occurs
  sw = findstr(upper(s),'W')
  >> sw = 3   10   15   21   27

  or

  sw = sort([findstr(s,'w') findstr(s,'W')])
  >> sw = 3   10   15   21   27

18

Exercises 3.8 (cont)

- Using this result, calculate the no of times that 'w' or 'W' occurs in the string **s**
  length(sw)   or   max(size(sw))
  >> ans = 5

- Calculate the mean distance between occurrences of 'w' or 'W' in the string **s**. Hint: use the **diff()** function. If you don't know what this function does, look up the help on it.
  mean(diff(sw))
  >> ans = 6

Matlab Programming Module

Dr Peter Hansen
p.c.hansen@bham.ac.uk

School of Psychology / BUIC

Workshop 4
Contingencies
Advanced Data Types

Revision: 23 Oct 2017 (revised post Workshop)

Contingencies

if … elseif ... else … end

The ability to use contingent logic (different actions follow different conditions) is a defining characteristic of a computer language. Matlab supports this with different constructs. One of the most important is **if... elseif… else... end**. This has the specific syntax:

```
if TEST1
   Matlab code1
elseif TEST2
   Matlab code2
elseif TEST3
   Matlab code3
else
   Matlab code4
end
```

In this example, TEST1, TEST2, TEST3 are logical Matlab tests.

If TEST1 evaluates to true then the Matlab code immediately following it (code1) is run. Matlab will then exit the **if** construct and jump to the line following the final **end** statement. If TEST1 evaluates to false then TEST2 will be evaluated and the process repeated.

If TEST2 is true, code2 is run. If false, TEST3 will be evaluated.

If none of TEST1-3 evaluate as true then the code after the final **else** clause (code4) will be run.

There must be a single **if** statement and a single **end** statement. There can be as many **elseif** statements as needed. The final **else** clause is optional but there can be only one of these.

if … elseif ... else … end

Here is a more specific example. Try and work out what is going on before you type in any code:

```
a = 1;
b = 0;

if a==1
  b = 10;
elseif a==0
  b = 20;
else
  b = 30;
end

b
>> b = 10
```

Two variables are initially defined here before the **if... end** loop is started.

Logical Matlab test. Is **a** equal to 1? Remember that == means "**is equal to?**" as opposed to = which means "**assign value to**"

Suppose, instead, that at the top **a** has value of 0 (and not 1)
What would be the value of **b** at the end?

Suppose that at the top **a** has value of 3 (and not 1)
What would be the value of **b** at the end?

if … elseif … else … end

Another example:

```
a = 2.7; b = 2;
if  (a>=1) & (a<=3)
   b = 2*b;
elseif (a<1)
   b = 0;
else
   b = 5;
end
```

This means:
"is **a** greater than or equal to 1"
AND
"is **a** less than or equal to 3"

Remember: **&** is the symbol for a logical AND

What is the value of b ?

if … elseif … else … end

Another example

```
a = 3.7; b = 2;
if  (a<=1) | (a>=3)
   b = -b;
else
   b = 4;
end

b
>> b = -2
```

Remember: **|** is the symbol for a logical OR

Suppose, instead, that a=2.7 (not 3.7).
What would be the value of b at the end?

if … elseif … else … end

Nesting of **if** statements allows for more complex contingencies:

```
a = 2.3; b = 10;
if  a <= 0
   if b <= -5
      c = 1;
   else
      c = 2;
   end
else
   if b <= -5
      c = 3;
   else
      c = 4;
   end
end
```

Test with **a** - is **a** less than or equal to 0?

If the test with **a** was met (so **a** <= 0), then follow the instructions here

If the test with **a** was not met (so **a** was **not** <= 0), then follow the instructions here

What is the the value of c?

switch … case … end

An alternative, but similar, construct to **if** is **switch... case... end**. For example:

```
x = 2;
switch x
  case 1
    y = -1;
  case 2
    y = -2*x;
  case 3
    y = -3*x;
  otherwise
    y = 0;
end
```

'switch x' means 'switch to the case below where x is equal to that value'

'case 1' is short for 'in the case that x==1'

'case 2' is short for 'in the case that x==2'

'case 3' is short for 'in the case that x==3'

'otherwise' is short for 'in the case that x is not equal to 1 or 2 or 3". This is like the **else** clause in the **if** construct

What is the value of y?

for … end

The **for … end** loop allows operations to be repeated. For example suppose we want to print numbers 1 to 5. Here the variable i incrementally takes each of the values in the array [1:5] i.e. 1 to 5:

```
for i=1:5
  disp(i)
end

>> 1
>> 2
>> 3
>> 4
>> 5
```

disp(x) displays the array, without printing the array name. In all other ways it's the same as leaving the semicolon off an expression except that empty arrays don't display.

Multiple uses for **end**. Firstly, note here how **end** has been used as the final closing statement in each of the **if,** the **switch** and the **for** constructs above.

But remember also that **end** can also be used in a different way – as an item selector when used inside an array to select its final element, as in **x(end)**.

for … end

Suppose we want to create a vector and step through its values:

```
Vec=[12:-3:0];
for idx=Vec
  disp( idx )
end

>> 12
>> 9
>> 6
>> 3
>> 0
```

Vec contains 12 9 6 3 0.

idx is set to be equal to **Vec**

We then step through its values, one by one.

The **idx** variable defined here takes on the values of **Vec**, one by one, each time we go around the for loop:

idx: 12 9   6   3   0

This is probably the simplest form of the **for... end** use.

for … end

Or, alternatively, we could do this:

```
Vec=[12:-3:0];
for idx=1:length(Vec)
  disp( Vec(idx) )
end

>> 12
>> 9
>> 6
>> 3
>> 0
```

Vec still contains 12 9 6 3 0 but unlike before, the **idx** variable defined here is now effectively an index for the column no corresponding to the elements in **Vec** i.e:

idx:	**1**	**2**	**3**	**4**	**5**
Vec(idx):	**12**	**9**	**6**	**3**	**0**

Look carefully at this example and the preceding example and make sure you understand how and why they are slightly different.

for … end

Another example, constructing a matrix, showing nesting of one **for** loop within another **for** loop:

```
M=zeros(4,4);
for r=1:4
   for c=1:4
      M(r,c)=r * c;
   end
end

M
M =
     1      2      3      4
     2      4      6      8
     3      6      9     12
     4      8     12     16
```

Here we are populating the matrix M by putting in a value of r*c in the r^th row and the c^th column

for … end

In a **for** loop the construct can be any array:

```
for i=2:2:8
   disp(i)
end
```

Note how this is an array that steps in jumps of 2

Or we could create the array first and then step through it:

```
Z=2:2:8;
for i=Z
   disp(i)
end
```

Or the array could be a char array:

```
for i='abcdefg'
  disp(i)
end
```

while … end

The final Matlab contingency construct is **while … end** which loops for as long as a particular condition holds. For example:

```
a = 1; b = 0.25;
count = 1;
while count <= 10
   a = a + a*b;
   count = count + 1;
end
```

We only stay in the loop whilst **count** is less than or equal to 10

When **count+1** is equal to 11 we drop out of the loop

```
a
>> a = 9.3132
count
>> count = 11
```

The value of the variable **count** as 11 shows us that the loop has gone around 10 times before exiting the **while** loop

Note **end** is again re-used by Matlab, this time as the final statement in the **while...** construct.

breaking out of loops

The **break** command can be used to escape from inside **for** and **while** loops. Matlab will jump to line immediately <u>following</u> the final **end** statement containing the **for** or **while** loop. For example:

```
a = 1; b = 0.25;
count = 0;
while a >= 1
   a = a + a*b;
   count = count + 1;
    if count>100
       break;
    end
end

a
count
```

Note here that because **a** initially equals 1 and **a** can only ever get bigger inside the loop this test will always be true

This **if** statement is basically a sanity check to stop the program looping forever

When **count** > 100 (i.e. **count** equals 101) then break out

If the **break** statement is reached then jump to the line <u>following</u> the final **end** of the **while** loop. Here that line is effectively outside (at the end of) the program so **break** exits the program.

continuing within a loop

Sometimes you may want to stay within a **for** or **while** loop but pass control to the <u>next</u> iteration of the loop, skipping over any remaining statements in the body of the loop. This can be achieved with a **continue** statement. For example:

```
for r=1:4
   for c=1:4
      if r*c>10
        continue
      end
      N(r,c)=r*c;
   end
end

N
```

If the **continue** statement is reached then the program will loop back to the next value of **c** in the **for c=1:4** line. It will skip over the N(r,c)=r*c; line.

Compare this output matrix, N, with the very similar one, M, from 4 slides back.

Make sure you understand how and why these matrices differ.

Vectorizing

Often Matlab can do things in several ways. For example creating a matrix and populating it in a systematic way. This could be done with:

```
M = [ 5:5:500 ];
```

Or in a **for... end** loop as:

```
for i=1:100
  M(i)=i*5;
end
```

The former version is known as the *vectorized* version. Not only is it more concise to write, it will run far more quickly and efficiently using less memory and other computer resources.

Wherever possible vectorized forms should always be used. In larger programs it can make many hours of processing difference.

More Data Types - Cells and Structures

Cells I

Simple Matlab matrices must have the same no of columns in each row, and each row must be of the same data type

If you try and create a matrix with different no of columns an error is generated:

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

← Error

Similarly if you try and create a matrix which has different data types, an error will result:

```
b = [ [ 60  70 ]; 'cats' ];
```

← Error

Cells II

Matlab does allow these kind of constructs but only by using a more general data type known as a **cell**. Cells can contain rows with different numbers of columns and can contain different data types.

Cells are assigned with **curly brackets { }** like this:

A new sort of bracket. Take care not to confuse with round ( ) or square [ ] brackets. Curly brackets are unique to cells.

```
a ={  [1 2]  [3 4 5]  }
>> a =  [1x2 double]     [1x3 double]
```

Note each cell element is itself a vector

Or:

```
b = {  [60 70];  'cats'  }
>> b =  [1x2 double]
         'cats'
```

← Note this cell element is a string

Cells III

You can create a cell array one cell at a time. Matlab expands the size of the cell array with each assignment. For example:

```
g(1,1) = { [1 4 3; 0 5 8; 7 2 9] };
g(1,2) = { 'Anne Smith' };
g(2,1) = { 3+7i };
g(2,2) = { -pi:pi/4:pi };
```

If you assign data that is outside the dimensions of the current array, Matlab automatically expands the cell array to match. It fills any intervening cells with empty matrices. The following will automatically converts array g from a 2x2 to 3x3 matrix:

```
g(3,3) = {5};
```

This is general Matlab behaviour, not specific to cells. It works with other array types too:

```
a=zeros(2,2)
a(10,1) = 1
```

Cells IV

The values of cell elements can be displayed (recursively) using the **celldisp()** function:

```
c = { [1] [2  3  4]; [5;  9] [6  7  8; 10  11  12] }
celldisp(c)

>> c{1,1} =  1

>> c{2,1} =  5
             9

>> c{1,2} =  2     3     4

>> c{2,2} =  6     7     8
            10    11    12
```

Cells V

Cell elements can be accessed in the standard way with round brackets **( )** used to retrieve elements. Slightly confusingly however, curly brackets **{ }** can **also** be used to retrieve elements. The difference is in the return data type:

```
c = {  [1] [2  3  4]; [5;  9] [6  7  8; 10  11  12]  }
>> c =     [         1]    [1x3 double]
       [2x1 double]    [2x3 double]
```

```
d = c(1,2)
>> d = [1x3 double]
class(d)
>> ans = cell
```

Round brackets used here.
Returns **d** as 1x1 matrix of type 'cell'.
The data type of **d** will always be of type cell.

```
e = c{1,2}
>> e = 2     3     4
class(e)
>> ans = double
```

Curly brackets used here.
Returns **e** as 1x3 matrix of type 'double'.
The return type of **e** will be the type of the actual entry.

Cells VI

Once you have complicated cell arrays it may be slightly more tricky to extract data. For example:

```
c = {  [1] [2  3  4]; [5;  9] [6  7  8; 10  11  12]  }
```

Values within a cell element need to be retrieved by a 2nd level of addressing:

```
c{1,2}
>> ans =   2     3     4
```

1st level of addressing – selects the array in **c** that is in the 1st row, 2nd col position

```
c{1,2}(2)
>> ans =  3
```

2nd level of addressing – having got the array, then select from it the element in the 2nd position

If you find this confusing consider using an intermediate (temporary) variable to achieve what you want. e.g.:

```
tmp = c{1,2}
tmp(2)
```

Cells VII

Cells can themselves be nested within cells to make arbitrarily complicated data structures. For example:

```
c = {  [1] [2  3  4]; [5;  9] [6  7  8; 10  11  12]  }
d = { c c ; c c }
>> d =  {2x2 cell}    {2x2 cell}
        {2x2 cell}    {2x2 cell}
```

The more complex the cell array, the more levels of addressing may be needed to retrieve values. So here the actual numeric values need a 3rd level of addressing:

```
d{1,2}
d{1,2}{2,2}
d{1,2}{2,2}(2,2)
>> ans =  11
```

1st level of addressing
2nd level of addressing
3rd level of addressing

Converting TO cell format

- There are special Matlab functions to convert to cell format. The function **num2cell()** creates a cell array from the contents of a numeric array e.g.

```
n1 = [ 1 2 ; 3 4 ]
>> n1 = 1 2
        3 4
n2 = num2cell(n1)
>> n2 = [1] [2]
        [3] [4]
```

Standard 2x2 numeric array

2x2 cell array. Each cell element is individually a single number

- The function **cellstr()** creates a cell array from strings with each row of the input character array placed in a separate cell e.g.

```
s1 = strvcat('Hello', 'Psychology');
s2 = cellstr(s1)
class(s2)
>> ans =  cell
```

Note that this is the same as:
s2 = { 'Hello'; 'Psychology' };

Converting FROM cell format

- To do the reverse conversion (**from** cell format) there is a general function **cell2mat()**
- This converts the contents of a cell array (of the **same** data type) into a single matrix of that data type e.g.:

```
c = {  [1]   [2  3  4]; [5;  9]   [6  7  8; 10  11  12]  }
cell2mat(c)
>> ans = 1     2     3     4
         5     6     7     8
         9    10    11    12
```

Notice that in doing the conversion we have lost information about what data was originally in which sub-array. The conversion would also fail if there were missing data elements such that the resultant matrix **c** would be incomplete.

- Note however that if the cell array contains different data types or itself contains other cells there is no generic function to convert from this. **cell2mat()** will also fail in this case:

```
sn = { [ 1 2 ]; 'Hello'}
cell2mat(sn)
```

This will generate an error

Structures I

Data sets often have hierarchical structure. To represent such data, Matlab has one other data type, the **structure**.

To be clear about what we mean here, imagine a simple Reaction Time (RT) experiment carried out on several participants. We might want to record:

- Participant Name (string)
- RTs (array of type double)
- Error counts (array of type uint8)
- Whether debriefed or not (array of type Logical)
- Comments (string)

It is basically a simple database table.

Structures II

Such data can be entered as a structure as follows:

```
participant(1).name  = 'Steve Jones';
participant(1).RTs     =  [ 700 600 550 ];
participant(1).errors =  uint8([ 10 8 6 ]);
participant(1).debrief = true;
participant(1).comment = 'Fell asleep during expt';
participant(1)

participant(2).name  = 'Jane Smith';
participant(2).RTs     =  [ 600 500 450 ];
participant(2).errors =  uint8([ 3 2 1 ]);
participant(2).debrief = false;
participant(2).comment = 'Made very few errors';
participant(2)
```

Structures III

Structures can also be assigned using the **struct()** command. This has a general syntax:

s = struct('field1',VALUES1,'field2',VALUES2,...)

Where field1 is the *name* of the 1st field and VALUES1 are the *contents* of it

to create a structure array with the specified field names and values

As a specific example:

```
subjects  = struct('Name',{ 'Steve Jones' , 'Jane Smith' }, ...
'RTs', { [700 600 550] [600 500 450] })

subjects
subjects(1)
```

Structures IV

Note that in the previous example, multiple values were assigned to the struct by passing the values in a single **cell**.

For example the names were passed as:
{'Steve Jones' , 'Jane Smith'},

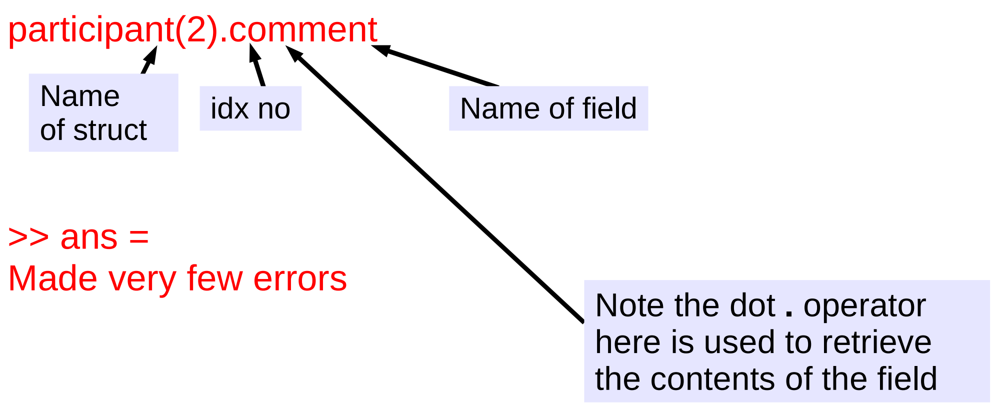
and the RTs were passed as:
{ [700 600 550] [600 500 450] }

Note also the use of the three dots … at the end of the 1st command line in the previous slide in order to allow the (long) command to span 2 lines of input.

This … is known as the Matlab **line continuation** operator.

Structures V

Values in structures are usually retrieved by using the **name** of the field after a **dot .** operator:



```
participant(2).comment
```

Name of struct
idx no
Name of field
Note the dot **.** operator here is used to retrieve the contents of the field

```
>> ans =
Made very few errors
```

Structures VI

Adding Fields to Structures

A new field can be added to every structure in an array by adding a field to a single structure. For example, to add a date-of-birth field, use an assignment like:

```
participant(2).dob = '06-05-1982';
```

Note the date is actually being stored as a string (array of type char)

Now participant(2).dob has the assigned value. Every other participant structure in the array also has the dob field, but these fields will contain an empty matrix until you explicitly assign a value to them.

Structures VII

Deleting Fields from Structures

You can remove a given field from every structure within a structure array using the **rmfield()** function. Its most basic form is:

newstruct = rmfield(array, 'field')

where array is a structure array and 'field' is the name of a field to remove from it. To remove the dob field from the participant array, for example, enter:

```
participant = rmfield(participant, 'dob');
```

Remember to assign the result of **rmfield()** back to something or nothing will seem to have changed. You can, as here, assign it back to the structure you are changing, effectively overwriting the old version.

Cells ↔ Structures

There are two functions in Matlab that can be used to convert cells to and from structures:

struct2cell()	Convert structure array to cell array. Note that if the structure contain p fields and is of size m rows by n cols, this function will return a cell of size p x m x n
cell2struct()	Converts cell array into structure array

For cell2struct(), if C = cell array, F=char or cell array of strings containing field names and D=dimension of the data, the syntax is:

```
C = { 'Vision' 1; 'Audition' 2 }
F = { 'modality' 'code' }
D = 2
cell2struct(C, F, D)
```

For next week

- Make sure you can do the Exercises here
- If using “MATLAB for Behavioural Scientists” (Rosenbaum), make sure you read and understand Chapter 5 and Chapter 7.
- Have a look at Workshop 5 handout on Canvas before next weeks workshop

Matlab Workshop 4

Exercises

Revision: 10 Oct 2017

Exercises 4.1

- Define
  V = randi(100,1,10000);
- Describe what variable **V** consists of. How many numbers? What sort of numbers? What range of numbers?
- Using the **for ... end** syntax write a program to step through each of the elements in the array **V** one value at a time and to print out each value
- Using the **if ... else ... end** syntax, now amend the above program so that it only prints out the value in the array if the value is greater than 99

Exercises 4.2

- Amend the program again to use a counter and then count how many times **V** exceeds 99 (Hint: this number should be about 100)
- Now try rewriting this program to use the **while … end** syntax rather than the **for … end** syntax

Exercises 4.3

Suppose we have this data from an expt:

```
data = [ 390      0.45;
         347      0.32;
         866      0.98;
         549      0.67;
         589      0.72;
         641      0.50;
         777      0.77;
         702      0.68 ];
```

Reaction Time (RT) in ms

Proportion Correct

- Using **for … end** and **if … end** syntax write a program to identify and save only those participants who had mean RTs greater than 500ms and proportions correct greater than 0.65.
- Save the matching data in 2 matrices:
  - **ID** (idx no of each participant that meets the criteria)
  - **scores** (2 col matrix, col 1 = RT and col 2 = proportion correct)
- How many participants fit the criteria?

Exercises 4.4

- Imagine you want to show some stimuli with particular values in an experiment. Suppose the stimuli to be shown need to have values drawn from a Normal distribution with a std dev of 1 (the default) and a mean equal to **A^B** where **A** has the values 1,2,3,4 and **B** also has the values 1,2,3,4 (i.e. 16 different values)

- Using a **for ... end** loop write a program to generate the 16 stimulus values above and store the values in a 4x4 matrix called **stim**

- Using **if... else... end** statements amend the above program to include only the cases where **A** is greater than 1 and **B** is greater than 2 (so that the matrix **stim** now has several zero entries)

Exercises 4.5

In a real experiment we would likely want to generate multiple trials and to show them repeatedly to participants

- Amend the initial version of the previous program from Exercise 4.4 (with the16 outputs) by adding a further **for … end** loop such that you generate 100 examples for each of the 16 conditions. Save these values in a 4x4x100 matrix called **stim100**

- Calculate the mean value of **stim100** for each of the 16 conditions by taking the mean across the 3rd dimension. Save this in a matrix **stimMean** and check this is of size 4x4

- Write a program to display the 1600 numbers in **stim100** in a randomised order. (This is tricky!)

Exercises 4.6

- Define
  s1 = 'I love Matlab'
  s2 = [true false]
  s3 = [10 13 NaN Inf]
- Combine arrays s1, s2, s3 into a 1x3 (row) cell array, **K**
- Combine arrays s1, s2, s3 into a 3x1 (column) cell array, **L**
- Expand the cell array **K** to be of size 1x10 by adding the numeric array [100 200 300; 7 8 9] as the 10th element of **K**
- Remove element 5 of the cell array **K**.
  Confirm **K** is now of size 1x9

Exercises 4.7

- Define:
  E = { zeros(2,2)+4; strvcat('abcdefg','1234'); 1:100 };
- Recover the 1st cell element of **E** as a cell, called **F**
- What is the size of **F** (how many rows and columns)?
- Convert **F** to its native data type, as variable **G**
- What is the size of **G** (how many rows and columns)?
- Recover the value of the 2nd cell element of **E**, 1st row, 3rd col in its native data type

Exercises 4.8

- Create a Matlab data structure **P** that has the following fields:
  - **Person name**
  - **Height**
  - **First language spoken**
  - **Right-handed?**
- Add 2 complete sample records and display them
- Retrieve the first language spoken of the 2nd participant
- Add a new field **Age**, fill in the entries for this field
- Remove the field **Height**

Exercises 4.9

- Convert struct array **P** (above) into a cell array called **X**
- What is the size of **X**?
- Use the **squeeze()** function on **X** to remove the extra dimension and create a cell array **Z** of size 4x2
- Now convert cell array **Z** into a struct array **S** with the same field names as **P**

Matlab Workshop 4

Suggestions for Answers to Exercises

Revision: 10 Oct 2017

Exercises 4.1

- Define
  V = randi(100,1,10000);

- Describe what variable **V** consists of. How many numbers? What sort of numbers? What range of numbers?
  The matrix V is of size 1 row by 10000 columns. It contains 10,000 integer numbers between 1 and 100, randomly sampled from a uniform distribution. There should be roughly an equal number of 1s, 2s, 3s, etc up to 100.

- Using the **for ... end** syntax write a program to step through each of the elements in the array **V** one value at a time and to print out each value

```
for n=1:10000              for n=1:length(V)              for n = V
  disp( V(n) )      OR       disp( V(n) )          OR       disp(n)
end                        end                            end
```

Exercises 4.1 (cont)

- Using the **if … else … end** syntax, now amend the above program so that it only prints out the value in the array if the value is greater than 99

```
for n = 1:length(V)
   if V(n) > 99
      disp( V(n) )
   end
end
```

OR

```
for n = V
   if n > 99
      disp(n)
   end
end
```

Exercises 4.2

- Amend the program again to use a counter and then count how many times **V** exceeds 99 (Hint: this number should be about 100)

```
count = 0;
for n = 1:length(V)
   if V(n) > 99
      count = count + 1;
   end
end

count
```

OR

```
count = 0;
for n = V
   if n > 99
      count = count + 1;
   end
end
```

Exercises 4.2 (cont)

- Now try rewriting this program to use the **while … end** syntax rather than the **for … end** syntax

```
count = 0;
n = 1;
while n <= length(V)
   if V(n) > 99
      count = count + 1;
   end
   n = n + 1;
end

count
```

Exercises 4.3

Suppose we have this data from an expt:

```
data = [ 390      0.45;
            347     0.32;
            866     0.98;
            549     0.67;
            589     0.72;
            641     0.50;
            777     0.77;
            702     0.68 ];
```

Reaction Time (RT) in ms

Proportion Correct

- Using **for … end** and **if … end** syntax write a program to identify and save only those participants who had mean RTs greater than 500ms and proportions correct greater than 0.65.
- Save the matching data in 2 matrices:
  - **ID** (idx no of each participant that meets the criteria)
  - **scores** (2 col matrix, col 1 = RT and col 2 = proportion correct)
- How many participants fit the criteria?

Exercises 4.3 (cont)

- Using **for ... end** and **if ... end** syntax write a program to identify and save only those participants who had mean RTs greater than 500ms and proportions correct greater than 0.65.

```
ID=[ ];
scores=[ ];
count = 0;
for n=1:length(data)
   if data(n,1)>500 & data(n,2)>0.65
      count = count + 1;
      ID(count) = n;
      scores(count,1)=data(n,1);
      scores(count,2)=data(n,2);
   end
end
```

Exercises 4.3 (cont)

- Save the matching data in 2 matrices:
  - **ID** (idx no of each participant that meets the criteria)
  - **scores** (2 col matrix, col 1 = RT and col 2 = proportion correct)

```
ID =      3     4     5     7     8
scores =  866.0000    0.9800
          549.0000    0.6700
          589.0000    0.7200
          777.0000    0.7700
          702.0000    0.6800
```

- How many participants fit the criteria?

```
count                  OR        length(ID)
>> count = 5                     >> ans = 5
```

Exercises 4.4

- Imagine you want to show some stimuli with particular values in an experiment. Suppose the stimuli to be shown need to have values drawn from a Normal distribution with a std dev of 1 (the default) and a mean equal to **A^B** where **A** has the values 1,2,3,4 and **B** also has the values 1,2,3,4 (i.e. 16 different mean values)

Exercises 4.4 (cont)

- Using a **for ... end** loop write a program to generate the 16 stimulus values above and store the values in a 4x4 matrix called **stim**

```
stim = [ ];
for A=1:4
   for B=1:4
       stim(A,B) = A^B + randn;
   end
end

stim
```

Exercises 4.4 (cont)

- Using **if... else... end** statements amend the above program to include only the cases where **A** is greater than 1 and **B** is greater than 2 (so that the matrix **stim** now has several zero entries)

```
stim=[ ];
for A=1:4
   for B=1:4
       if (A>1 & B>2)
         stim(A,B) = A^B + randn;
       end
   end
end

stim
```

Exercises 4.5

In a real experiment we would likely want to generate multiple trials and to show them repeatedly to participants

- Amend the initial version of the previous program (with the 16 outputs) by adding a further **for … end** loop such that you generate 100 examples for each of the 16 conditions. Save these values in a 4x4x100 matrix called **stim100**

- Calculate the mean value of **stim100** for each of the 16 conditions by taking the mean across the 3rd dimension. Save this in a matrix **stimMean** and check this of size 4x4

- Write a program to display the 1600 numbers in **stim100** in a randomised order. (This is tricky!)

Exercises 4.5 (cont)

- Amend the initial version of the previous program from Exercise 4.4 (with the 16 outputs) by adding a further **for … end** loop such that you generate 100 examples for each of the 16 conditions. Save these values in a 4x4x100 matrix called **stim100**

```
stim100 = [ ];          OR          stim100=zeros(4,4,100);
for A=1:4
    for B=1:4
       for sample=1:100
          stim100(A,B,sample) = A^B + randn;
       end
   end
end
```

Exercises 4.5 (cont)

- Calculate the mean value of **stim100** for each of the 16 conditions by taking the mean across the 3rd dimension. Save this in a matrix **stimMean** and check this is of size 4x4

```
stimMean = mean(stim100,3);
size(stimMean)

>> ans = 4  4
```

Exercises 4.5 (cont)

- Write a program to display the 1600 numbers in **stim100** in a randomised order. (This is tricky!)

  There are many ways to do this. The simplest is probably to use the **reshape()** function to change the 4x4x100 **stim100** matrix into a single row matrix **tmp** of size 1600x1 as in:

  ```
  tmp = reshape(stim100,1600,1,1);
  ```

  and then to step through **tmp** in a random way by creating an index of the 1600 numbers that is randomly permuted:

  ```
  idx = randperm(1600);
  for n = idx
    disp(tmp(n))
  end
  ```

Exercises 4.6

- Define

  ```
  s1 = 'I love Matlab'
  s2 = [ true false ]
  s3 = [ 10 13 NaN Inf ]
  ```

- Combine arrays s1, s2, s3 into a 1x3 (row) cell array, **K**

  ```
  K = { s1 s2 s3 }
  ```

- Combine arrays s1, s2, s3 into a 3x1 (column) cell array, **L**

  ```
  L = { s1; s2; s3 }
  ```

Exercises 4.6 (cont)

- Expand the cell array **K** to be of size 1x10 by adding the numeric array [100 200 300; 7 8 9] as the 10th element of **K**
  K(10) = { [100 200 300; 7 8 9] }
  or
  K{10} = [100 200 300; 7 8 9]

- Remove element 5 of the cell array **K**
  K(5) = [ ]

- Confirm **K** is now of size 1x9
  size(K)
  >> ans = 1 9

Exercises 4.7

- Define:
  E = { zeros(2,2)+4; strvcat('abcdefg','1234'); 1:100 };

- Recover the 1st cell element of **E** as a cell, called **F**
  F = E(1)

- What is the size of **F** (how many rows and columns)?
  size(F)
  >> ans = 1 1

Exercises 4.7 (cont)

- Convert **F** to its native data type, as variable **G**
  ```
  G = cell2mat(F)
  >> ans = 4 4
           4 4
  ```
- What is the size of **G** (how many rows and columns)?
  ```
  size(G)
  >> ans = 2 2
  ```

Exercises 4.7 (cont)

- Define:
  ```
  E = { zeros(2,2)+4; strvcat('abcdefg','1234'); 1:100 };
  ```
- Recover the value of the 2nd cell element of **E**, 1st row, 3rd col in its native data type
  ```
  E{2}(1,3)
  >> ans = c
  ```

Exercises 4.8

- Create a Matlab data structure **P** that has the following fields:
  - **Person name**
  - **Height**
  - **First language spoken**
  - **Right-handed?**
- Add 2 complete sample records and display them
- Retrieve the first language spoken of the 2nd participant
- Add a new field **Age**, fill in the entries for this field
- Remove the field **Height**

Exercises 4.8

Add 2 complete sample records and display them:

```
P(1).Name  = 'Steve Jones';
P(1).Height    =  180;
P(1).FirstLang =  'English';
P(1).RH = true;
P(1)

P(2).Name  = 'Jane Smith';
P(2).Height    =  170;
P(2).FirstLang =  'French';
P(2).RH = NaN;
P(2)
```

Exercises 4.8 (cont)

- Retrieve the first language spoken of the 2nd participant
  P(2).FirstLang
  >> ans = French
  (if same as example above)

- Add a new field **Age**, fill in the entries for this field
  P(1).Age=21
  P(2).Age=45

- Remove the field **Height**
  P = rmfield(P,'Height')

Exercises 4.9

- Convert struct array **P** (above) into a cell array called **X**
  X = struct2cell(P)

- What is the size of **X**?
  size(X)
  >> ans = 4 1 2

- Use the **squeeze()** function on **X** to remove the extra dimension and create a cell array **Z** of size 4x2
  Z = squeeze(X)
  size(Z)
  >> ans = 4 2

Exercises 4.9 (cont)

- Now convert cell array **Z** into a struct array **S** with the same field names as **P**

  ```
  fnames = {'Name'  'FirstLang'  'RH'  'Age'}
  S = cell2struct(Z, fnames, 1)
  ```

  Note that **S** is a 2x1 struct array with fields:

  Name
  FirstLang
  RH
  Age

Matlab Programming Module

Dr Peter Hansen
p.c.hansen@bham.ac.uk

School of Psychology / BUIC

Workshop 5
Data Input-Output

Revision: 12 Oct 2017

Matlab Paths

Your Matlab search path

- Matlab stores a list of a set of directories called the Matlab **path** which it searches when running commands. If the command can be found in your path then it will be run, otherwise an error will be generated, such as
  ??? Undefined function or variable
- Many of these directories are already present by default in Matlab
- Type
  path

  to see your current path list of search directories

Your own Matlab dir

- It is important that you have your own search directory in which to store Matlab files so that Matlab can find and run them
- Outside Matlab, from the Windows Start Menu, open **My Computer** or open Accessories → Windows Explorer
- Go to your School U: drive and check if there already exists a folder called:

  U:\matlab or U:\Matlab or U:\MATLAB

- If this folder does not exist then create it as U:\matlab
- We now need to tell Matlab about this new path

Adding Paths to Matlab

- From the Matlab Command Window run:
  pathtool
- In the Set Path window, click the **Add Folder** button and add the dir:
  U:\matlab (or the variant of this name if it already exists)
- Click the **Close** button to close the Set Path window (click the **No** button when the **Save Path** popup window appears)
- Run the path command again and make sure this new directory has been added (scroll to the top of the list to see it)

Adding other Paths

- What we have done above is to make a **temporary** change to the Matlab path – it will last only as long as your Matlab session
- Quit Matlab and then restart it again. Run the path command again. You should see that the directory you added above is no longer in the Matlab path
- Run pathtool again and once more add the U:\matlab directory
- This time in the Set Path window click the **Save** button
- Close the Set Path window by clicking the **Close** button
- You have now permanently added the new dir to your Matlab path

Data Input

Getting User Input I

The simplest way of getting input from the user is with the **input()** function. Try the following:

```
u1 = input('Type a number: ')
```

and then type a number (followed by the return key). The variable **u1** will then be set to the number you typed.

Try running this again and typing in a Matlab expression (e.g. 3*4). You should see that works too. Now try typing some chars (e.g. t or plop). Now you should see an error generated. Why? Finally, try typing the letter **i** or **j** – you should see these appear to work. Why is this?

There are several problems here that ideally need to be overcome to make **input()** more user friendly.

Getting User Input II

Firstly, even if the user types a number, there is no checking by **input()** on whether it is sensible or not. For example, if:

```
u2 = input('Type a number between 1-5: ')
```

the user could then type, say, 9 anyway. When dealing with user defined input it is important in the real world to validate the input. For example, we could amend the above input with a loop:

```
u2=0;
while (u2<1 | u2>5)
  u2 = input('Type a number between 1-5: ')
  if  (u2<1 | u2>5)
    disp('The number you have typed is invalid. Please try again!')
  end
end
```

Getting User Input III

Secondly, it is important to stop arbitrary Matlab expressions being entered as these could unwittingly wreak havoc. For example:

```
u3 = input('Type a number: ')
```

Let's show one further problem continuing on from slide 8 above. In response to the prompt, type `dir` or `ls` (lowercase letter L and S). Can you see what has happened? It is unlikely that this is what the Matlab programmer had in mind.

Thirdly, it is clearly useful to be able to capture character input from the user rather than just numbers (or Matlab expressions!)

It turns out that both of these problems can be solved at the same time.

Getting User Input IV

If we modify the command to be as follows:
u4 = input('Type your name: ', 's')

Then the variable **u4** will be constrained to be of type char.

This will also be true for:
u5 = input('Type your age: ', 's')

If you type in 23, then **u5** will be a string of chars containing the chars '2' and '3'. Note carefully therefore that variable **u5** does **not** contain the **number** 23. To be useful it will need to be converted to a number using the str2num() function.

This is good: it stops arbitrary Matlab expressions being entered and evaluated. But it now makes more work for the programmer as you need to deal with different data types.

Getting User Input V

For example, to cope with the case of wanting users to input a number between 1-5 we probably need some code like this:

Declare u6 to be an empty array

str2num() will return an empty array **[ ]** if the input is anything other than a number

```
u6=[ ];
while (isempty(u6) | u6<1 | u6>5)
  u6 = str2num(input('Type a number between 1-5: ','s'))
  if isempty(u6)
     disp('That is not a number!')
  elseif (u6<1 | u6>5)
     disp('The number you have entered is out of range!')
  end
end
```

isempty() tests if u6 is still empty.
help isempty for more details

Pausing

The Matlab function **pause** can be used to pause input. It works in one of two ways. By waiting until a key (any key) is hit:

```
disp('Hit any key to continue...'); pause
```

Or by waiting for a fixed number of seconds before continuing:

```
disp('Thinking...'); pause(3); disp('My thinking is over...')
```

Note that the commands are written on one line here as otherwise **pause()** would include the time taken for you to write the commands out over separate lines.

Recording Reaction Times

Times can be recorded in Matlab using a pair of functions called **tic** and **toc**. A timer is started when **tic** is run and the time relative to it is recorded when **toc** is run. This allows us to measure a basic reaction time, as follows:

```
tic; u2 = input('Enter a number: '); RT = toc
```

RT will be the time taken between the **tic** line and the **toc** line in Seconds. Note that there is nothing special about the variable name RT. I chose a name that is meaningful but it could be called anything.

Note also that the three commands are written on one line here as otherwise this would include the time taken for you to write the commands out over separate lines. Normally, when written in a script – rather than interactively as here – this would not be an issue and you would write the commands on three separate lines.

Caveats on Timings in Matlab

Caveat 1

Timing information using **tic/toc** is likely to be less accurate on the teaching PCs rather than on Windows machines specially prepared for running experiments. Multiple processes may be running on the PC and this can impact the timing information. Ideally the operating system needs to be miminalized so that only basic and necessary processes (and Matlab) are running.

Caveat 2

tic/toc are not the most sensitive of timing routines in any case, even under the best of circumstances. For time sensitive work, there are more accurate timing routines available through various other 3rd party toolboxes.

For our purposes here we will consider tic/toc to be adequate.

getkey()

- The **input()** command requires the user to press the return key to capture the input. This is no good for timing information.

- However, the command **getkey()** can be used instead. This is not part of standard Matlab. It is provided by a 3rd party developer.

- Using a web browser outside of Matlab, download the following file from here:

  www.buic.bham.ac.uk/teaching/matlab/getkey.m

  and save it in your own U:\matlab directory (the one you created earlier):

Using getkey()

- **getkey()** works in a similar way to **input()**

  However it does not take a user prompt as an argument. Rather, it is usually called with no arguments**.** It only accepts a single keypress and does not wait for the return key to be pressed. It then returns the ASCII code of the key pressed.

- Try this a few times:

  ```
  A = getkey()
  ```

- Note the return codes of the keys you press.
- Note that A is always numeric.

String Formatting

Formatting of data for on-screen display

You can change the style of output displayed using the **format** command. Try the following and see what happens:

```
t = [ 0 : 0.1 : 1]'

format bank
t
format rat
t
format short
t
format short g
t
format long
t
format long g
t
format                    (This is the default)
t
```

String Formatting I

The Matlab function for creating a formatted string is called **sprintf()**. Note that although it contains the word 'print', it is not really to do with printing, more with formatting.

Try the following:

```
Name='John'
Age=39
sprintf('My name is %s', Name)
sprintf('I am %d years old', Age)
sprintf('I am %d years old. Next year I will be %d', Age, Age+1)
sprintf('My name is %s and my age is %d', Name, Age)
tmp=sprintf('My name is %s and my age is %d', Name, Age);
disp(tmp)
```

The 1st argument of sprintf() is a format string

2nd and other arguments of sprintf() are the variables themselves

The point of this is that you can create a very complex string and save it in a variable to use later

String Formatting II

The first argument of the **sprintf()** function is a string containing placeholders, in a structured format, for the variables that follow:

These placeholders take the form:

%s	for a string (or single character)
%d	for an integer
%f	for a floating point number
%e	for a number in scientific notation (e.g.65,000,000 = 6.5e7)

Other things that can be put in the string are:

\n	for a new line
\t	for a tab space
%%	for an actual percentage sign
' '	(two apostrophes in a row) for an actual apostrophe

String Formatting III

Examples:

```
sprintf('This is pi in floating point: %f', pi)
sprintf('This is a very big no: %e', 65000000)
sprintf('Everyone should give 110%% effort')
sprintf('This\n is a\n very disjointed \n sentence!\n')
```

Finally try these:

Note here trying to use a **single** double-quote symbol

```
sprintf('It"s a very nice day')     ← Oops Wrong!
sprintf('It''s a very nice day')    ← Correct
```

Note there are **two** single apostrophes in a row

String Formatting IV

Field and width and precision can be specified like this:

sprintf('%f', pi) ← Default formatting

sprintf('%5.3f', pi)

The 3 here refers to the no of digits to show to the right of the decimal point

The 5 here refers to the minimum no of digits to show overall (including the decimal point)

"Padding" of the output with leading 0's can be done like this:

sprintf('%8.3f', pi)

sprintf('%08.3f', pi) ← A single 0 is used here to signify to Matlab to use padding Two or more 0's do not mean pad with 2 zeros!

There are other things that can be done with string formatting (e.g. justification). See doc sprintf for more details.

Data Output

Writing Data to Files I

Data can be written out to files using three Matlab functions:

- **fopen()** (file open: to open the file for writing to)
- **fprintf()** (file print format: to write data to the file)
- **fclose()** (file close: to close the file after writing)

For example, try the following:

```
cd U:\matlab
fid=fopen('test1.txt', 'w');
fprintf(fid,'This is line one.\n');
fprintf(fid,'The value of pi is: %f\n', pi);
fclose(fid);
```

Now examine the output file you have created. You can do this from inside the Matlab GUI (double click “test1.txt” in the Desktop → Current Folder window) or you can open the file outside of Matlab

Writing Data to Files II

Let's look at each of the three functions in more detail:

- **fopen()** (file open: to open the file for writing to)

The single job of this function is to create and return a **file identifier**. In this case I chose to call the variable **fid**:

```
fid = fopen('test1.txt', 'w');
```

File identifier   Name of file   File permissions

fid is just a Matlab variable - there is nothing special about the name. I chose **fid** as short for file identifier but I could have called it anything.

Later you will see that the **fid** variable will be passed as the first argument to other file input/output routines.

Writing Data to Files III

If **fopen()** cannot open the file, it returns **-1**. This could happen if for example you do not have read or write permission for the directory you are in, or if you are over disk quota.

Therefore it is a good idea to always check the return value of fid and only call **fprintf()** if the value of fid does not equal -1. For example:

```
fid=fopen('test1.txt', 'w');
if fid ~= -1
  fprintf(fid,'This is line one.\n');
  fprintf(fid,'The value of pi is: %f\n', pi);
  fclose(fid);
else
  disp('problem opening file!')
end
```

This is an example of a **sanity check** in your programming to make sure that the things you think are happening really are happening as you expect.

File Permissions

The following are the most common file permissions used with **fopen()**:

Permission	Meaning
'r'	Open file for reading (default).
'w'	Open file, or create new file, for writing; discard existing contents, if any.
'a'	Open file, or create new file, for writing; append data to the end of the file.
'r+'	Open file for reading and writing.
'w+'	Open file, or create new file, for reading and writing; discard existing contents, if any.
'a+'	Open file, or create new file, for reading and writing; append data to the end of the file.
't'	Opens the file in text mode instead of (the default) binary mode. (Note: on Linux, text and binary mode are the same).

Writing Data to Files IV

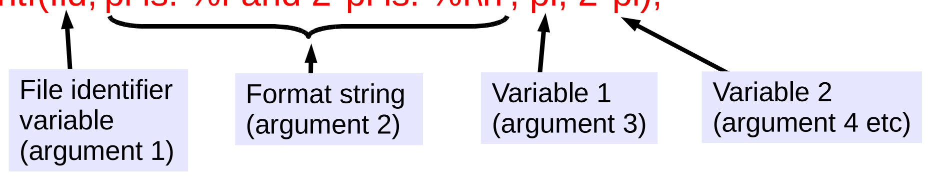
- **fprintf()** (file print format: to write data to the file)

fprintf() operates in exactly the same way as the previous **sprintf()** function we looked at above, except there is now a new first argument in the function call, which is the name of the **file identifier** variable.

Effectively, the formatted string created by the function is “printed” to the file linked with the name of the file identifier.

Normally we would call **fprintf()** with at least three arguments, as with:

```
fprintf(fid,'pi is: %f and 2*pi is: %f\n', pi, 2*pi);
```


File identifier variable (argument 1)
Format string (argument 2)
Variable 1 (argument 3)
Variable 2 (argument 4 etc)

Writing Data to Files V

- **fclose()** (file close: to close the file after writing)

The single job of this function is to **close** an already existing opened file. The function is called with a single argument, the **file identifier** (in this case called fid) that was used to open the file in the first place:

```
fclose(fid);
```

To avoid file data loss it's a good habit to always check that you have a matching **fclose()** to go with every opening **fopen()**.

Note also that an error will be generated if you try and close a file that is ***not*** open. The return status of **fclose()** can be checked with:

```
status = fclose(fid);
```

where status = 0 if successful and -1 if unsuccessful.

Saving Matlab Data with save() I

The Matlab **save** routine can be called as either a command:

```
save mydata
```

Note no quotation marks here

Or as a function:

```
save('mydata')
```

Note quotation marks *are* needed here

Both of the above routines will save ***all*** variables in the current workspace into a file called **mydata.mat**

The file is **not** a text file, but is Matlab-specific binary format that can only be understood by Matlab.

This is a very quick way of saving all of your session data into a single file for later use

Saving Matlab Data with save() II

The same routine can also be used to specify that only some of the variables in the current workspace should be saved:

```
A = 1:10;
B = 0:2:100;
C = 'Hello world!';
save mydata2 A C
save('mydata2','A','C')
```

Command format

Function format

Both of the above save only the variables A and C into the file mydata2.mat. Note that the .mat extension is added automatically.

Writing Text Data with save()

Data can be saved in text file format (rather than the default Matlab proprietary format) by using the extra argument '-ASCII' to the **save** routine:

```
D = [1:10]';
save mydata3.txt D -ASCII                OR
save('mydata3.txt','D','-ASCII')
```

Both of the above **save** lines save the variable **D** into the file mydata3.txt, but the file is now in text file format.

Have a look at the mydata3.txt file in a text editor outside Matlab to confirm this. You should see that the file is not formatted very elegantly but it is readable.

Writing Delimited Data with dlmwrite()

Another Matlab function that can be used is **dlmwrite()** which writes out data to a file delimited by a specified character. The default delimiter is a comma character (producing so-called CSV files). Another common delimiter is the tab ('\t') char.

For example:

```
data=rand(10,5);
dlmwrite('data1.csv',data);
dlmwrite('data2.tab',data,'\t');
```

Run these command and then have a look at the resulting output files (outside of Matlab) to see what the command has created.

Both formats can easily be read into SPSS or Excel

Reading Matlab Data with load()

Data that has been saved with the **save** routine can be loaded back into Matlab with the corresponding **load** routine. As with **save**, **load** can be run as either a command or as a function:

```
clear all
load mydata2      ← Command format
load('mydata2')   ← Function format
```

You can check this has worked by using the **who** command or by examining the workspace window

Either of these calls should restore the workspace so that it only contains variables A [1:10] and C ['Hello world!']

Note that, although the actual saved file is called 'mydata2.mat', only the first part of the name 'mydata2' is needed to load it.

Reading Text Data with load()

Structured data in a text file (this usually means a vector or matrix of numeric data with an equal number of columns per row) can also be read in using the **load()** function.

Using a browser outside of Matlab, download and save the file:
www.buic.bham.ac.uk/teaching/matlab/data2.txt
into your local U:\matlab drive

This file can then be read in as:

```
G = load('U:\matlab\data2.txt')
```

Note this would fail if, say, there were a number missing on line 2 such that the row only had 4 columns. In this case we might use the **textread()** function (covered below).

Reading Delimited Data with dlmread()

Character delimited text files, such as the ones we created earlier with the **dlmwrite()** function, can be read into Matlab using the **dlmread()** function.

For example:

```
D1=dlmread('data1.csv')

D2=dlmread('data2.tab')
```

Comma separated and tab delimited file formats are the two most common text file data interchange formats.

Note that when data1.csv is read into **D1** and data2.tab into **D2**, the data in **D1** and **D2** do not retain the commas or tabs – they contain only the numbers themselves

If the character separating the elements in the file (the delimiter character) is non-standard then this can be specified as in e.g.:

```
D3=dlmread('data2.tab', '\t')
```

37

Reading Unstructured Text Data

Finally, the **textread()** program can be used to read in data from text files where the data is of mixed type (not just numbers) and where the arrays are ragged (unequal no of columns). Using a browser outside of Matlab download and save the following file to your U:\matlab folder :
www.buic.bham.ac.uk/teaching/matlab/data3.txt

This file can be read into Matlab with the command:

```
filename = 'U:\matlab\data3.txt';
[forename, surname, experience, rating, age, faculty] = …
     textread(filename,'%s %s %s %f %d %s')
whos
```

This is known as a format string

You will see that many of the return arrays (like forename) are cell arrays. **textread()** is a complex function and there is much more that it can do. See doc textread for further details.

38

File Handling in General

Many things can go wrong when handling files so it is important when writing programs to cope with unexpected conditions that arise. For example, you do not want, unknowingly, to overwrite existing files, so putting a check in for this by using the **exist()** function is a smart thing to do.

Try running this code snippet twice with the same filename:

```
data=[1:10]'
filename = input('Name of save file: ', 's');
if exist(filename)
   disp('Errr... that file already exists!')
else
   save(filename, 'data', '-ASCII')
end
```

If you are not sure what the **exist()** function does, then look up the help on it

Evaluating Strings

Evaluating Strings I

In Matlab, it is possible to turn strings into executable commands using the **eval()** function. The function evaluates the contents of the string as if it were a standard line of Matlab. For example:

```
s1='3 + 4'
eval(s1)

s2='1:10'
eval(s2)

eval('t=0:0.1:1')

i=1
s3=['a' num2str(i) ' = 100' ]
eval(s3)
```

Note that this information is provided for completeness only. It is important that you see and understand what the **eval()** function does in case you encounter it. However, in practice it should be very rarely used.

If you find yourself wanting to use **eval()**, stop and think. Then think again. There is ALMOST CERTAINLY a better way to do things in Matlab than by using **eval()**

Evaluating Strings II

eval() could for example be used to generate multiple variables.

Suppose for example we want to create:

```
A1=[ ]
A2=[ ]
A3=[ ]
…
A100=[ ]
```

Note that in practice in this case you would probably just want to create a large matrix to begin with and not use **eval()** at all. For example, could use:

A=zeros(100,1) or A=cells(100,1)

That's a lot of work typing out the 100 lines by hand.
A less tedious method would be to use **eval()** like this:

```
for n=1:100
  eval(['A' num2str(n) '=[ ]' ]);
end

A74
```

For next week

- Make sure you can do the Exercises here
- If using “MATLAB for Behavioural Scientists” (Rosenbaum), make sure you have read and understand Chapter 6. Overall you should now have covered everything in the book up to the end of Chapter 7.

Matlab Workshop 5

Exercises

Revision: 12 Oct 2017

Exercises 5.1

- Using the **for ... end** syntax write a program to present the nine numbers from 1 to 9 one at a time on the screen
- Amend the above program using the **input()** command to prompt the user to type back the number that is presented. Save each of these replies separately in an array called **Response1**. **Response1** should contain all 9 responses.
- Amend the program again to use **tic** and **toc** such that it also records each reaction time for the user response to each of the numbers. Save these times in an array called **RT1**. **RT1** should contain all 9 RTs.

Exercises 5.2

- Using the **for ... end** syntax write a program to present the eight characters 'abcdefgh' one at a time on the screen
- Amend the above program using the **getkey()** command to prompt the user to type back the character that is presented. Save the returned values (ASCII codes) in an array called **Response2**
- Amend the program again to use **tic** and **toc** such that it also records the reaction time of the user response to each of the characters. Save these times in an array called **RT2**

Exercises 5.3

- Define
  A = 'Hello Peter'; B = 'The year is:'; C = 2010; D = 1.3333;
- Using the variables **A**-**D** above and the **sprintf()** function make the following text appear on the screen exactly as shown. Each command should include at least one of the variables.

```
Hello Peter
The year is: 2015
1.33%
01.3
```

Exercises 5.4

- Define:
  V = randi(100,10000,1); ← Generates 10,000 random integers in the range 1-100
- Using the **fprintf()** command syntax save the array **V** into a file called **v1.txt**. The file should be 10,000 lines long with one entry per line.
- Amend the above program to save **V** as two columns per line, with a comma separating each of the two entries, into a file called **v2.txt.** The file should be 5,000 lines long.

Exercises 5.5

- Use the **load()** function to load v1.txt into array **V1**
- Use the **load()** function to load v2.txt into array **V2**
- Use the **dlmread()** function to load v2.txt into array **V3**
- Use the **textread()** function to load v2.txt into arrays **V4a** and **V4b** (in the same command)

Exercises 5.6

- Create a string called **S1** that creates a 20x20 matrix full of random integers in the range 1-10 in array **J**
- Use the **eval()** function to convert the string **S1** to a command and run it
- Create a string called **S2** that contains the command to load the text file v1.txt into array **K**
- Use the **eval()** function to convert the string **S2** to a command and run it

Matlab Workshop 5

Suggestions for Answers to Exercises

Revision: 12 Oct 2017

Exercises 5.1

- Using the **for ... end** syntax write a program to present the nine numbers from 1 to 9 one at a time on the screen
- Amend the above program using the **input()** command to prompt the user to type back the number that is presented. Save each of these replies separately in an array called **Response1**. **Response1** should contain all 9 responses.
- Amend the program again to use **tic** and **toc** such that it also records each reaction time for the user response to each of the numbers. Save these times in an array called **RT1**. **RT1** should contain all 9 RTs.

Exercises 5.1

- Using the **for ... end** syntax write a program to present the nine numbers from 1 to 9 one at a time on the screen

```
for i = 1:9
   disp( i )
end
```

Exercises 5.1

- Amend the above program using the input() command to prompt the user to type back the number that is presented. Save each of these replies separately in an array called **Response1**. **Response1** should contain all 9 responses.

```
for i = 1:9;
  disp( i )
  Response1(i) = input('Type the number: ');
end

Response1
```

Exercises 5.1

- Amend the program again to use tic and toc such that it also records each reaction time for the user response to each of the numbers. Save these times in an array called **RT1**.
  RT1 should contain all 9 RTs.

```
for i = 1:9;
  disp( i )
  tic;
  Response1(i) = input('Type the number: ');
  RT1(i) = toc;
end

RT1
```

Exercises 5.2

- Using the **for ... end** syntax write a program to present the eight characters 'abcdefgh' one at a time on the screen
- Amend the above program using the **getkey()** command to prompt the user to type back the character that is presented. Save the returned values (ASCII codes) in an array called **Response2**
- Amend the program again to use **tic** and **toc** such that it also records the reaction time of the user response to each of the characters. Save these times in an array called **RT2**

Exercises 5.2

- Using the **for ... end** syntax write a program to present the eight characters 'abcdefgh' one at a time on the screen

```
letters='abcdefg';
for i = 1:length(letters)
    disp(letters(i))
end
```

Exercises 5.2

- Amend the above program using the **getkey()** command to prompt the user to type back the character that is presented. Save the returned values (ASCII codes) in an array called **Response2**

```
letters='abcdefg';
for i = 1:length(letters)
    disp(letters(i))
    disp('Type the letter')
    Response2(i)=getkey();
end

Response2
```

Exercises 5.2

- Amend the program again to use **tic** and **toc** such that it also records the reaction time of the user response to each of the characters. Save these times in an array called **RT2**

```
letters='abcdefg';
for i = 1:length(letters)
    disp(letters(i))
    disp('Type the letter')
    tic
    Response2(i)=getkey();
    RT2(i)=toc;
end

Response2
RT2
```

Exercises 5.3

- Define
  `A = 'Hello Peter'; B = 'The year is:'; C = 2010; D = 1.3333;`

- Using the variables **A**-**D** above and the **sprintf()** function make the following text appear on the screen exactly as shown. Each command should include at least one of the variables.

```
Hello Peter
The year is: 2015
1.33%
01.3
```

Exercises 5.3

- Using the variables **A**-**D** above and the **sprintf()** function make the following text appear on the screen exactly as shown. Each command should include at least one of the variables.

  ```
  disp(sprintf('%s',A))
  Hello Peter

  disp(sprintf('%s %d',B,C+5))
  The year is: 2015

  disp(sprintf('%4.2f%%',D))
  1.33%

  disp(sprintf('%04.1f',D))
  01.3
  ```

Exercises 5.4

- Define:
  `V = randi(100,10000,1);` ← Generates 10,000 random integers in the range 1-100

- Using the **fprintf()** command syntax save the array **V** into a file called **v1.txt**. The file should be 10,000 lines long with one entry per line.

- Amend the above program to save **V** as two columns per line, with a comma separating each of the two entries, into a file called **v2.txt.** The file should be 5,000 lines long.

Exercises 5.4

- Using the **fprintf()** command syntax save the array **V** into a file called **v1.txt**. The file should be 10,000 lines long with one entry per line.

```
fid=fopen('v1.txt','wt');
for i=1:10000
   fprintf(fid,'%d\n',V(i));
end
fclose(fid);
```

Or more efficiently in a vectorized format:

```
fid=fopen('v1.txt','wt');
fprintf(fid,'%d\n',V);
fclose(fid);
```

Exercises 5.4

- Amend the above program to save **V** as two columns per line, with a comma separating each of the two entries, into a file called **v2.txt.** The file should be 5,000 lines long.

```
fid=fopen('v2.txt','wt');
for i=1:5000
   fprintf(fid,'%d, %d\n',V(i),V(5000+i));
end
fclose(fid);
```

Exercises 5.5

- Use the **load()** function to load v1.txt into array **V1**
- Use the **load()** function to load v2.txt into array **V2**
- Use the **dlmread()** function to load v2.txt into array **V3**
- Use the **textread()** function to load v2.txt into arrays **V4a** and **V4b**

Exercises 5.5

- Use the **load()** function to load v1.txt into array **V1**
  V1=load('v1.txt');
- Use the **load()** function to load v2.txt into array **V2**
  V2=load('v2.txt');
- Use the **dlmread()** function to load v2.txt into array **V3**
  V3=dlmread('v2.txt');
- Use the **textread()** function to load v2.txt into arrays **V4a** and **V4b** (in the same command)
  [V4a, V4b] = textread('v2.txt','%d,%d');

Exercises 5.6

- Create a string called **S1** that creates a 20x20 matrix full of random integers in the range 1-10 in array **J**
- Use the **eval()** function to convert the string **S1** to a command and run it
- Create a string called **S2** that contains the command to load the text file v1.txt into array **K**
- Use the **eval()** function to convert the string **S2** to a command and run it

Exercises 5.6

- Create a string called **S1** that creates a 20x20 matrix full of random integers in the range 1-10 in array **J**.

  S1 = 'J=randi(10,20,20)'

- Use the **eval()** function to convert the string **S1** to a command and run it

  eval(S1)

Exercises 5.6

- Create a string called **S2** that contains the command to load the text file v1.txt into array **K**

```
S2 = 'K=load(''v1.txt'');'
```

Note that each of these are two single ' quotation marks next to each other and **not** a double quotation mark “

- Use the **eval()** function to convert the string **S2** to a command and run it

```
eval(S2)
```

Matlab Programming Module

Dr Peter Hansen
p.c.hansen@bham.ac.uk

School of Psychology / BUIC

Workshop 6
Modules and Functions

Revision: 12 Oct 2017

Modules and Functions

M-files

So far all the code run in previous workshops has had to be entered on the keyboard line by line. When you want to rerun it, or correct a mistake you have had to re-type the code. This is a very inefficient process.

It is much more efficient to write the code in an external file and then to run the code in this file. In Matlab these files have a suffix of **.m** and are known as **M-files or .m files**. These M-files are the basic scripting / programming files of Matlab.

If the code needs to be run many times, the M-file can be re-run again and again. If it is wrong, or needs to be extended, we can edit the M-file as needed and then run it again.

M-files – Example

Let's suppose we want to write a program to work out and display the number of days between now and the end of the year. The code for this could be:

The date format used by **clock()** is a vector with 6 elements: [year month day hour min sec]

```
date_now = clock();
next_year = date_now(1) + 1;
date_eoy  = [ next_year 1 1 0 0 0 ];
diff_time   = etime(date_eoy,date_now);
days = floor(diff_time/(60*60*24));
disp(sprintf('Days between now and end of year: %d',days));
```

Calculated by dividing the diff_time by number of seconds in a day and rounding down

If we type this in, the command line will return something like:

```
Days between now and end of year: 60
```

etime() returns the time difference in seconds between two date vectors. To fully understand the code for this example look up the help for the **clock()** and the **etime()** functions.

M-files – Example

That's a lot of code to type in every time we want to calculate the no of days to the end of the year. So let's put it in an M-file instead:

In the Matlab GUI go the top menu bar and click either the "New Script" button (far left hand side) or "New –> Script" or type "Ctrl +N"

A new window should open and present you with a text editor.

Type the code from the previous slide (or copy and paste from the command window) into the text editor. Save the file as days2go.m in your U:\matlab directory i.e. save it as:

U:\matlab\days2go.m

M-files – Example

Now close the Matlab text editor.

If you look at the file days2go.m with a text editor outside of Matlab:
- Open **My Computer**
- Change directory to **U:\matlab**
- Right-mouse click on **days2go.m**
- **Open with** → **Wordpad** (or **Notepad**)

you will see that the file is saved as a standard text file (even though it ends in a **.m** extension). It can be edited in any text editor you prefer. Matlab just happens to have its own text editor built in. Close the editor window

In the Matlab Command Window, now type:
days2go

Note that you do <u>not</u> type the **.m** at the end

M-files – Example

If this has worked ok then you should see the contents of the file days2go.m have been run and Command Window should show something like:

```
Days between now and end of year: 60
```

If you have the Matlab Workspace tab open in the right hand side of the Matlab GUI (if not, select **Window** → **Workspace**), then you should also see that the intermediate variables calculated within the routine (such as next_year) are present in the workspace. You could also verify this by listing their values by typing in the command window:

```
next_year
date_now
```

The M-file has run almost exactly as if you had typed the commands one by one on the command line.

M-files – Modules

In Matlab, code such as this (in a separate file) is referred to as a **module** or **script**. Rather than typing it, putting all the code into one long and complex M-file, modules provide a way of separating out code into a set of smaller, logically distinct set of M-files.

Imagine we wanted to create a program to do 3 distinct things:

(1) Ask the user for their birthday,
(2) Work out no of days from beginning of year to the birthday,
(3) Work out no of days from the birthday to the end of the year.

This could be efficiently coded as **four** separate M-files. Why four? Because one M-file needs to be the master module. Effectively this master M-file is the “real” program. It in turn would then call each of the three M-files that contain code for (1), (2) & (3) above. In this way, the subroutines are then kept separate from the programs using them.

Using M-files

One benefit of using M-files is that they allow Matlab code to be neatly formatted and indented to show the start and end of loops. Such formatting helps prevent some simple errors. Make sure you do this.

Your Matlab code can also be commented. Comments are remarks and notes written to aid understanding of the code itself. They are not runnable Matlab code. Comments are **very** important and should not be regarded as an optional extra, but as a vital part of the program. Good programmers write lots of comments.

- As ***you*** are writing comments, they will focus ***your*** mind on what you think the Matlab code should be doing and why
- When other people look at your Matlab code it helps ***them*** understand what you are doing and why
- When ***you*** come to look back at your code after a few months it will help ***you*** understand what you have done and why

Using M-files

The symbol in Matlab for a comment statement is the percent character '%'. This can be either on a line of its own:

```
% Create a vector containing integers from 1 to 10
A=[1:10]
```

Or the comment can be at the end of a particular line:

```
for i=1:10           % Step through a loop from 1 to 10
   if i==5           % Look for the case where loop equals 5
      disp(i)        % Print out the value
   end
end
```

Note the use of indenting here to give clarity to the beginning and end of the **for ... end** loop and the **if … end** test.

Anything following the % sign is not interpreted by Matlab as code to be run. Instead, Matlab will ignore it.

Using M-files

Example comments for days2go.m:

```
% Get current date and time as 6 element vector
date_now = clock();

% Calc next year by extracting current year and adding 1
next_year = date_now(1) + 1;

% Generate date on Jan 1st, midnight (end of this year/beginning of next year)
date_eoy  = [ next_year 1 1 0 0 0];

% Calc difference in seconds between end of year and now
diff_time    = etime(date_eoy,date_now);

% Calc no of days by dividing diff_time by (no of secs in day) and rounding
days = floor(diff_time/(60*60*24));

% Print out final calc in pretty format
disp(sprintf('Days between now and end of year: %d',days));
```

Using M-files

For the results of an M-file module to be reproducible, the module should be self contained, unaffected by other variables that have been defined elsewhere in the Matlab session, and uncorrupted by leftover graphics. This means that it is often desirable at the **beginning** of a script to explicitly clear any old variables to ensure that the previous definitions do not affect the results and to close all graphics windows, thus starting with a blank slate:

```
% Remove old variable definitions
clear all

% Close old graphics windows
close all
```

Debugging M-files

A simple way of debugging an M-file is to use the echo command.

By default, when you run an M-file you do not see each command that has been run (in this respect, it is different to you typing it in the command window). However, you can turn on the display of Matlab commands run inside the M-file by including the command:

```
echo on
```

at the beginning of the M-file. If you do use this command then you should remember to turn the echoing **off** at the end of the module (otherwise it will be on all the time) by including the command:

```
echo off
```

at the end of the M-file.

Debugging M-files

More sophisticated debugging makes use of the built in Matlab debugger. It is good to get in the habit of using this as it can greatly speed up finding and fixing errors in Matlab code. Let's look at this in its simplest form. Edit the days2go.m module so that it contains a basic error. Change the line:

```
date_now = clock();
```

to:

```
date_now = cloc();
```

cloc() is a function that does not exist. We are deliberately making an error here (that we know about) so we can see how the debugging works.

Now run the days2go module. You should get an error like:

```
days2go

Undefined function or variable 'cloc'.
Error in days2go (line 1)
date_now = cloc();
```

Debugging M-files

Now let's turn the Matlab debugger on. In the Matlab command window type the command:
dbstop if error

Make sure all open Matlab editor windows are closed.
Now try running the days2go module again:
days2go

This time the behaviour should be different. Firstly, the Matlab editor should open in the days2go.m file and put you on the line with the error. Secondly the command window will show a prompt that looks like:
K>>
rather than
>>
The “K” prompt indicates Matlab is in debug mode.

Debugging M-files

Matlab should also put a helpful green arrow next to the line in the M-File where it thinks the error is. Fix the error in days2go.m (change the command back to **clock()**) and resave the module. This should get you out of debug mode and return you to the normal Matlab prompt. Check that it runs correctly again.

Now lets use another debug command. In days2go.m between the line (next_year=..) and (date_eoy=..) *insert* the command:
keyboard

and resave the module. Now run days2go again. This time the K>> prompt will appear but without an error being generated. The program has stopped on the line with keyboard and is now awaiting input. At this point you can examine and/or change variables. But note you are actually inside the module – so the values are what the module sees at this point.

Debugging M-files

For example:

```
K>> next_year
next_year = 2018

K>> next_year = 2019
next_year = 2019
```

Here we are overwriting the value of next_year from 2018 to 2019 (in the middle of the module).

This is in order to see what effect it has.

Note this is only a temporary change and does not change the actual .m file.

There are 2 commands you could type at this point: dbcont will carry on processing the module:

```
K>> dbcont
Days between now and end of year: 420
```

← using 2019 year

Or dbquit will immediately stop all processing and return to the Matlab command prompt.

See doc keyboard or help keyboard for more information

The Problem with Modules

Modules rely on variables being set and retrieved in a shared 'global' workspace. For example, we run the getbday.m module in order to get a birthdate and then save it in the variable bday.

However in doing this we've also defined (in the same global space) the variables year, month and day. For a bigger module there could be many more variables. This could easily get very messy with hundreds of names defined in the common name space.

Also, the next module called that makes use of bday may have defined its own variables. Perhaps bday is a variable inside the next module. Variable name clashes will cause subtle errors and bugs.

Or maybe, in trying to keep things clean, clear all has been run at the beginning of the next module. So the value of bday will be lost.

Functions

The solution to these problems is to define our own **functions** rather than using **modules**. Functions differ from modules in several ways:

- They have a defined entry and exit syntax. This means each function needs to be told explicitly what input is coming into it and what output is going out of it.
- Variables inside the function are “private” to the function and will be unknown in the global workspace or to the calling module.
- You can have several subfunctions within one M-file. Only the part of the file that contains the relevant function is run. With a module, the program is executed sequentially through the file. So one file is needed to define each module.

Structure of a Function

- The first line of the M-file must start with the Matlab word **function**
- On the same line, after **function**, are several things, in order:
  (1) A list of variables returned by the function (optional)
  (2) If variables are returned then an **=** sign must follow
  (3) The **name** of the function (this must be present)
  (4) A list of variables passed into the function (optional)
- If variables are returned from a function then this is done with either a named single variable or with an array of named variables using square brackets **[ ]** (followed by an = sign)
- If variables are passed into the function then this is done with round brackets **( )** after the name of the function in a comma separated list

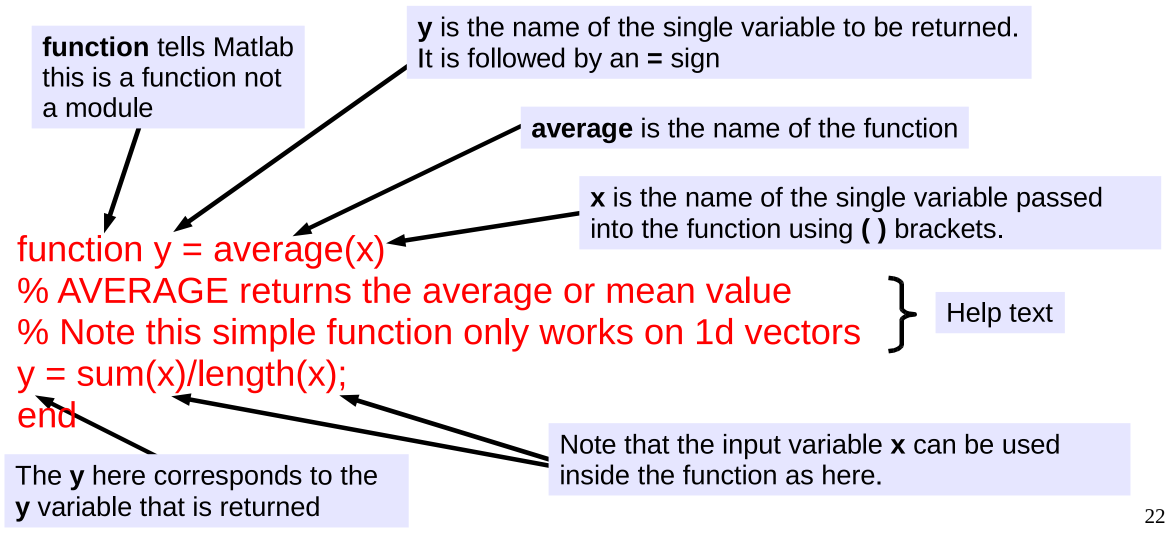
Structure of a Function

- Lines after the first function definition line that begin with a % (comment symbol) are treated as the help text for the function
- When you type help fxxx (where fxxx is the name of your function) the % lines from line 2 of the M-file onwards will be printed out
- Lines after the % help text lines are standard Matlab code. These will execute the code required for the function.
- To avoid any confusion it makes sense to save the M-file with the same name as the function that it contains. So if the function is called fxxx, then save it in a file called fxxx.m

Note that the names of the variables passed into, and returned from the function, as well as the name of the function itself, are all chosen by the programmer

Example Function

Here is a simple function that takes 1 input and returns 1 value. Create a new function using File → New → Function. Save it as average.m in your U:\matlab directory with the following contents:



function tells Matlab this is a function not a module

y is the name of the single variable to be returned. It is followed by an **=** sign

average is the name of the function

x is the name of the single variable passed into the function using **( )** brackets.

```
function y = average(x)
% AVERAGE returns the average or mean value
% Note this simple function only works on 1d vectors
y = sum(x)/length(x);
end
```

Help text

The **y** here corresponds to the **y** variable that is returned

Note that the input variable **x** can be used inside the function as here.

Running the Example Function

In the Matlab command window, define:

```
Z=1:11
```

And then run:

```
average(Z)
```

You should see:

```
ans = 6
```

Also, in the Matlab command window try typing:

```
help average
```

You should see the average() function help text as from the last slide. You have defined a function called average() that behaves just like the Matlab built-in function mean().

Functions returning 2 values

Let us define a new function called stat that calculates and returns both the mean and the standard deviation of a vector. Save this in a file called stat.m in your U:\matlab directory. Note how this function returns the **two** values (mn and sd):

```
function [mn, sd] = stat(x)
n    = length(x);
mn = sum(x)/n;
sd  = sqrt(sum((x-mn).^2/n));
end
```

Return values **mn** and **sd** passed back as an array in **[ ]** brackets

Single variable **x** passed into the function

In the Matlab command window you could use it like this:

```
Z = 1:11
[m,s] = stat(Z)
m =  6
s  =  3.1623
```

Functions returning 2 values

In fact we could further break this up with two internal subfunctions.
Create a file called stat2.m with the following contents:

```
function [mn, sd] = stat2(x)
mn = my_avg(x);
sd  = my_std(x);
end

function m = my_avg(x)
n  = length(x);
m = sum(x)/n;
end

function y = my_std(x)
n = length(x);
y = sqrt(sum((x-my_avg(x)).^2)/n);
end
```

These 2 subfunctions are private to the stat2.m module

Verify that stat2(Z) gives the same result as stat(Z).

Functions with no input values

A function could take no input values but return something.
Here is a simple function that returns the current year:

A single variable called **year** is returned from the function

```
function year = getyear()
date = clock();
year = date(1);
end
```

Note that the brackets after the function name are empty – no variables are passed into the function

Try creating this, saving it as getyear.m, and running it as:

```
getyear()
```

Functions with no return values

We could have a function that does something and does not need to return a value. Here is a simple pretty formatted function to print out the square root of its input to 3 decimal places:

Note there are no return variables defined before the name of the function

Single variable **x** passed into the function

```
function pretty_sqrt(x)
disp( sprintf('The square root of %f is %.3f', x, sqrt(x)) )
end
```

Try creating this in pretty_sqrt.m and running it with:
pretty_sqrt(10)

You might think this **is** returning a value here by printing it out (e.g. the sqrt of 10 in this example). But it isn't so don't be fooled. Try:
A = pretty_sqrt(10)

Functions with 2 input values

We could modify the previous function to take a second input variable that specifies the no of decimal places we want the printing to be accurate to. This might look something like:

Note still no return variables defined here

Two variables **x** and **p** passed into the function as a comma separated list

```
function pretty_sqrt(x,p)
fmt_txt = sprintf('The square root of %%f is %%.%df',p);
disp( sprintf(fmt_txt, x, sqrt(x)) )
end
```

Try modifying this in pretty_sqrt.m and running it with:
pretty_sqrt(10,6)

Private variables

To show global and private variables in action, try creating and saving an M-file called private.m containing:

```
function private(x)
x = 1;
y = 2;
disp(sprintf('x = %d', x))
disp(sprintf('y = %d', y))
end
```

Now, in the Matlab command window, clear all variables:

```
clear all
```

and set x to be 0:

```
x = 0
```

Private variables

Now call the function private() on x:

```
private(x)
```

At this point, what do you think the value of x is in the global space?
Type:

```
x
```

Also, what is the value of y in the global space?
Type:

```
y
```

If these results confuse you, then remember that the value of x=1 and the variable y existing at all, are only defined within the function private(). Once we are 'outside' the function then x reverts back to its original value (0) and y is not defined at all.

For next week

- Make sure you can do the Exercises here
- If using “MATLAB for Behavioural Scientists” (Rosenbaum), you should now have covered roughly all the material up to the end of Chapter 8

Matlab Workshop 6

Exercises

Revision: 12 Oct 2017

Exercises 6.1

Modify days2go.m such that it prints out the no of days from the **beginning** of the year. Save the new M-file as daysgone.m
Be careful not to overwrite days2go.m

Hint. The main change you need to make is to replace

```
date_eoy  = [ next_year 1 1 0 0 0 ];
```

Matlab 6 element clock vector for Jan 1st at the stroke of midnight next year. Note this is also the very end of this year (EOY).

with something like:

```
date_boy  = [ this_year 1 1 0 0 0 ];
```

Matlab 6 element clock vector for Jan 1st at the stroke of midnight this year. Note this is the very beginning of this year (BOY).

and then modify the calculation that you need do with beginning of year.
Run your new M-file and check that is has an output something like:

```
daysgone
Days since beginning of year: 304
```

Exercises 6.2

Comment your code in daysgone.m to a level that you think makes sense. If you are working with a partner, make sure they also understand and agree with your comments.

Make sure to resave your module with the same name - daysgone.m in U:\matlab

Run the command again in the command window:
daysgone

The command should still run in the same way, give the same output and none of the comments you made should be visible. If this is **not** the case, fix the problems and try again.

Exercises 6.3

Amend your code in daysgone.m again:

- Add lines to clear all previous variables and close any graphics windows. Comment these lines.
- Turn the echoing on at the beginning of the module, and off at the end
- Run your program and observe the effect of the echo statements.
- Remove the echo lines and run the module again so that it runs cleanly and only outputs something like:
  Days since beginning of year: 304

Exercises 6.4

Create a new module called getbday.m with the following code:

```
clear all
close all
year = input('What is your year of birth:');
month = input('What is your month of birth (1-12):');
day = input('What is your day of birth (1-31):');
bday = [ year mon day 0 0 0 ];
disp(sprintf('\nYou were born on: %s', datestr(bday)))
```

There is an error in this code. Run the module and use the debugger to help identify and fix the problem.

Exercises 6.5

Amend the M-file getbday.m to be a function that returns birthday as a 6 element date vector. Make sure it runs as intended by calling it as:
`birthday = getbday()`

Now amend the M-file days2go.m to be a function that takes the birthday vector (from above) as its input and calculates the number of days from birthday to the end of the year.

Now amend the M-file daysgone.m to be a function that takes the birthday vector (from above) as its input and calculates the days from the beginning of the the year to birthday.

Finally create a wrapper module called do_birthday.m that calls each of the above three functions in turn. Make sure it runs as:
`do_birthday`

Matlab Workshop 6

Suggestions for Answers to Exercises

Revision: 12 Oct 2017

Exercises 6.1

Modify days2go.m such that it prints out the no of days from the **beginning** of the year. Save the new M-file as daysgone.m
Be careful not to overwrite days2go.m

Hint. The main change you need to make is to replace

```
date_eoy  = [ next_year 1 1 0 0 0 ];
```

Matlab 6 element clock vector for Jan 1st at the stroke of midnight next year. Note this is also the very end of this year (EOY).

with something like:

```
date_boy  = [ this_year 1 1 0 0 0 ];
```

Matlab 6 element clock vector for Jan 1st at the stroke of midnight this year. Note this is the very beginning of this year (BOY).

and then modify the calculation that you need do with beginning of year.
Run your new M-file and check that is has an output like:

```
daysgone
Days since beginning of year: 304
```

Exercises 6.1

Modify days2go.m such that it prints out the no of days from the **beginning** of the year. Save the new M-file as daysgone.m

Should be something like:

```
date_now = clock();
this_year = date_now(1);
date_boy  = [ this_year 1 1 0 0 0];
diff_time   = etime(date_now,date_boy);
days = floor(diff_time/(60*60*24));
disp(sprintf('Days since beginning of year: %d',days));
```

Exercises 6.2

Comment your code in daysgone.m to a level that you think makes sense. If you are working with a partner, make sure they also understand and agree with your comments.

Make sure to resave your module with the same name - daysgone.m in U:\matlab

Run the command again in the command window:
daysgone

The command should still run in the same way, give the same output and none of the comments you made should be visible. If this is **not** the case, fix the problems and try again.

Exercises 6.2

```
% Get current date and time as 6 element vector
date_now = clock();

% Get current year from element 1 of  date_now
this_year = date_now(1);

% Generate a 6 element time vector for Jan 1, beginning of year
date_boy  = [ this_year 1 1 0 0 0];

% Generate the difference in time (in sec) between now and beg. of year
diff_time   = etime(date_now,date_boy);

% Divide time by (no. of sec in day) and round down
days = floor(diff_time/(60*60*24));

% Pretty print the answer
disp(sprintf('Days since beginning of year: %d',days));
```

Exercises 6.3

Amend your code in daysgone.m again:

- Add lines to clear all previous variables and close any graphics windows. Comment these lines.
- Turn the echoing on at the beginning of the module, and off at the end
- Run your program and observe the effect of the echo statements.
- Remove the echo lines and run the module again so that it runs cleanly and only outputs something like:

```
Days since beginning of year: 304
```

Exercises 6.3

```
% Clean out the workspace
clear all
% Close all open graphics windows
close all
% Turn the echoing on
echo on
% Get current date and time as 6 element vector
date_now = clock();
% Get current year from element 1 of date_now
this_year = date_now(1);
% Generate a 6 element time vector for Jan 1, beginning of year
date_boy = [ this_year 1 1 0 0 0];
% Generate the difference in time (in sec) between now and beginning of year
diff_time = etime(date_now,date_boy);
% Divide time by (no. of sec in day) and round down
days = floor(diff_time/(60*60*24));
% Pretty print the answer
disp(sprintf('Days since beginning of year: %d',days));
% Turn the echoing off
echo off
```

Exercises 6.4

Create a new module called getbday.m with the following code:

```
clear all
close all
year = input('What is your year of birth:');
month = input('What is your month of birth (1-12):');
day = input('What is your day of birth (1-31):');
bday = [ year mon day 0 0 0 ];
disp(sprintf('\nYou were born on: %s', datestr(bday)))
```

There is an error in this code. Run the module and use the debugger to help identify and fix the problem.

Exercises 6.4

There is an error in this code. Run the module and use the debugger to help identify and fix the problem.

This line:
```
bday = [ year mon day 0 0 0];
```

References a variable 'mon' that is not defined anywhere. This is shown in the Matlab command window:
```
Undefined function or variable 'mon'.

Error in getbday (line 6)
bday = [ year mon day 0 0 0 ];
```

We can fix this by using the previously defined variable 'month':
```
bday = [ year month day 0 0 0];
```

Exercises 6.5

Amend the M-file getbday.m to be a function that returns birthday as a 6 element date vector. Make sure it runs as intended by calling it as:
```
birthday = getbday()
```

Now amend the M-file days2go.m to be a function that takes the birthday vector (from above) as its input and calculates the number of days from birthday to the end of the year.

Now amend the M-file daysgone.m to be a function that takes the birthday vector (from above) as its input and calculates the days from the beginning of the the year to birthday.

Finally create a wrapper module called do_birthday.m that calls each of the above three functions in turn. Make sure it runs as:
```
do_birthday
```

Exercises 6.5

Amend the M-file getbday.m to be a function that returns birthday as a 6 element date vector. Make sure it runs as intended by calling it as:
birthday = getbday()

File getbday.m:

```
function bday = getbday()
year = input('What is your year of birth:');
month = input('What is your month of birth (1-12):');
day = input('What is your day of birth (1-31):');
bday = [ year month day 0 0 0];
disp(sprintf('\nYou were born on: %s', datestr(bday)))
end
```

Exercises 6.5

Now amend the M-file days2go.m to be a function that takes the birthday vector (from above) as its input and calculates the number of days from birthday to the end of the year.

```
function days2go(day)
   date_now  = clock();
   this_year  = date_now(1);
   next_year  = date_now(1) + 1;
   date_eoy   = [ next_year 1 1 0 0 0 ];
   date_bday = [ this_year day(2) day(3) 0 0 0];
   diff_time   = etime(date_eoy,date_bday);
   days          = floor(diff_time/(60*60*24));
   disp(sprintf('Days from birthday to end of year: %d',days));
end
```

Exercises 6.5

Now amend the M-file daysgone.m to be a function that takes the birthday vector (from above) as its input and calculates the days from the beginning of the the year to birthday.

```
function daysgone(day)
   date_now = clock();
   this_year  = date_now(1);
   date_boy  = [ this_year 1 1 0 0 0];
   date_day  = [ this_year day(2) day(3) 0 0 0];
   diff_time   = etime(date_day,date_boy);
   days         = floor(diff_time/(60*60*24));
   disp(sprintf('Days since beginning of year to birthday: %d',days));
end
```

Exercises 6.5

Finally create a wrapper module called do_birthday.m that calls each of the above three functions in turn. Make sure it runs as:
do_birthday

File do_birthday.m:

```
clear all
close all
birthday = getbday();
daysgone(birthday)
days2go(birthday)
```

Matlab Programming Module

Dr Peter Hansen
p.c.hansen@bham.ac.uk

School of Psychology / BUIC

Workshop 7
Plotting and Graphics

Revision: 13 Nov 2017

Plotting and Graphics

Figures

Matlab shows graphical information in **figure** windows. The first figure created is figure number 1, the next one is figure number 2 and so on. This number 1 or 2 etc is one of the properties of the figure object (the 'Number' property). At any one time, Matlab always has one figure window selected. By default, it will only write new data to this window. Some of the basic Matlab commands for manipulating graphics are:

figure	Creates a new figure window and brings it to the foreground
clf	Clears the contents of the currently selected figure
shg	Show graphics window - bring the current figure window to the foreground
close	Closes the currently selected figure window
close all	Close **all** figure windows

Figures

Each Matlab figure is associated with a figure **handle**. This is a Matlab structure. Each element of the handle structure represents a property of the figure that controls its appearance and behaviour.

H = gcf()	Get the figure handle **H** for the current figure
clf(H)	Clears the contents of the figure window with handle **H**
H = figure()	Create a new figure with handle **H**
figure(H)	If figure **H** exists (where **H** is a valid figure handle) make it the current figure and bring it to the foreground
close(H)	Close the figure window with handle **H**

Figures - Shortcuts

For those functions that accept a graphics **handle** as an argument, it is possible instead to use the figure **Number** (one of the properties of the figure handle) as a shortcut. This is in order to maintain backwards compatibility with Matlab versions prior to R2014b.

clf(N)	Clears the contents of the figure window with Number **N**
H = figure(N)	Create a new figure Number **N** with handle **H**
figure(N)	If figure **N** exists (where **N** is an integer number) make it the current figure and bring it to the foreground
close(N)	Close the figure window with Number **N**

Plots

There are many plotting functions in Matlab. The simplest is **plot()** which shows a 2d plot of some sort. We will use this as an example and discuss its options in more detail later. Firstly, let's show a basic example:

```
clear all
close all
H = figure(1)
x = -pi:pi/10:pi;
y = tan(sin(x)) - sin(tan(x));
plot(x,y)
shg
```

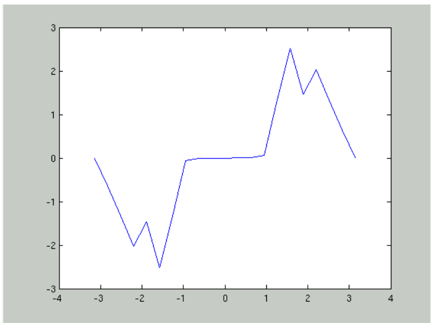


Figure Properties

The figure handle, since it is a Matlab structure, can be accessed and manipulated directly. This is the preferred approach since R2014b.

```
H
>> H =
>>
>>  Figure (1) with properties:
>>
>>      Number: 1
>>        Name: ''
>>       Color: [0.9400 0.9400 0.9400]
>>    Position: [670 575 554 420]
>>       Units: 'pixels'
>>
>>  Show all properties
>>
>>         Alphamap: [1x64 double]
>>         BeingDeleted: 'off'
>>         BusyAction: 'queue'
>>                  …
```

Display the contents of the structure that is handle **H** (the properties of the figure)

Figure Properties

For example let us change the background colour of the figure **H** and also remove the graphics toolbar in that particular window:

```
H.Color = [0 0.5 0.5];
H.ToolBar = 'none';
```

Remember elements of a Matlab structure are accessed using **.** notation as here

This should work instantly.

Figure Properties – Backwards Compatibility

An alternative way of viewing and changing the figure properties, that will work in all versions of Matlab including those prior to 2014b, is to use the Matlab **get()** and **set()** functions:

```
H2 = get(1)
>> H2 =
>>
>>
>>          Position: [1304 561 554 420]
>>             Units: 'pixels'
>>          Renderer: 'opengl'
>>      RendererMode: 'auto'
>>                  …
```

Get the figure handle structure **H** containing the properties of figure 1

Figure Properties – Backwards Compatibility

One property of the the figure is its position and size. This is shown in the list of the elements of **H** above. Or it can be shown individually with:

```
H.Position
>> ans =   680   678   560   420
```

The four values [left bottom width height] are the x and y positions of the bottom left corner of the figure and the figure width and height.

We can update these values using the **set()** command:

```
set(H, 'Position', [ 0 0 400 800 ] )
or
set(1, 'Position', [ 0 0 400 800 ] )
```

Note here that you can use **get()** and **set()** with either the figure handle **H** or the figure number **N**.

You should have noticed the figure has now moved and changed size.

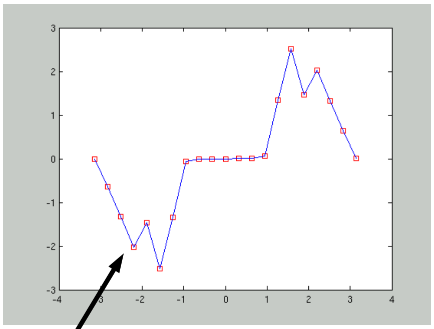
Plots

Now let's improve the look and feel of this plot. Firstly, let's overplot the points with red squares. Be careful here not to close the figure window before running this – we want to add more things to the currently open figure.

```
hold on
plot(x,y,'rs')
hold off
```

hold on holds the current plot and all axis properties so that subsequent graphing commands are *added* to the existing graph.

hold off returns to the default mode whereby new plot commands *erase* the previous plots.

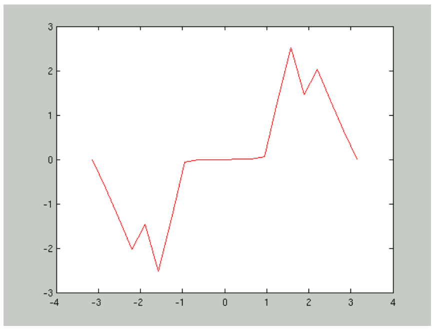


Red square symbols now present

Plots

If you do not specify a colour when plotting more than one line, **plot()** automatically cycles through colours. When we did the first plot, it defaulted to the colour blue. But we could have specified red instead:

```
clear all
close all
figure(1)
x = -pi:pi/10:pi;
y = tan(sin(x)) - sin(tan(x));
plot(x,y,'r')
```



This 'r' is an optional modifier to the plot function signifying the colour of the plot

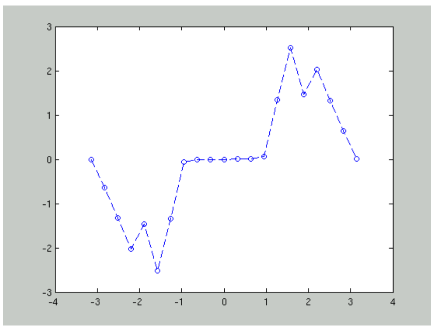
Plot Colour Modifiers

Symbol	Colour
b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black

Plots

Similarly, Matlab has defaulted the line style to be a continuous line joining up the points. Again, we could override the default and specify a dashed line '--' together with an 'o' symbol:

```
figure(1)
x = -pi:pi/10:pi;
y = tan(sin(x)) - sin(tan(x));
plot(x,y,'--o')
```



This '--o' is an optional modifier to the plot function signifying the style of the plot

Plot Style Modifiers

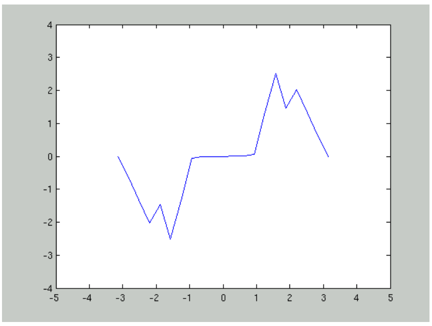
Symbol	Point style
.	point
o	circle
x	x-mark
+	plus
*	star
s	square
d	diamond
v	triangle (down)
^	triangle (up)
<	triangle (left)
>	triangle (right)
p	pentagram
h	hexagram

Symbol	Line style
-	solid
:	dotted
-.	dashdot
--	dashed
(none)	no line

Plot Axes

We could also change the range of the plotted axes. Instead of the default axes, suppose we wanted x to be plotted on a range from -5 to 5, and y from -4 to 4:

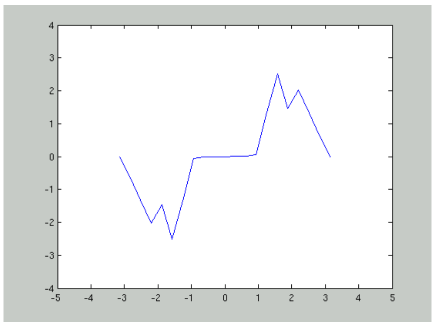
```
figure(1)
x = -pi:pi/10:pi;
y = tan(sin(x)) - sin(tan(x));
plot(x,y)
axis([ -5 5 -4 4])
```



Plot Axes

This could also have been done using the **xlim()** and **ylim()** functions. These are more convenient to use if changing only one axis. Suppose as above we want x to be plotted on a range from -5 to 5, and y from -4 to 4:

```
figure(1)
x = -pi:pi/10:pi;
y = tan(sin(x)) - sin(tan(x));
plot(x,y)
xlim([-5 5])
ylim([-4 4])
```

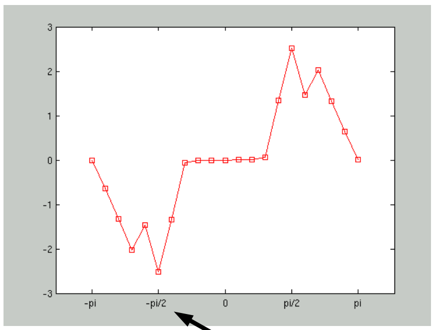


17

Plot Axes

You can adjust the axis tick-mark locations and the labels appearing at each tick. For example, we can relabel the x-axis with more meaningful values:

```
figure(1)
x = -pi:pi/10:pi;
y = tan(sin(x)) - sin(tan(x));
plot(x,y,'-rs')
set(gca,'XTick',-pi:pi/2:pi)
set(gca,'XTickLabel',{'-pi','-pi/2','0','pi/2','pi'})
```



Note axis change

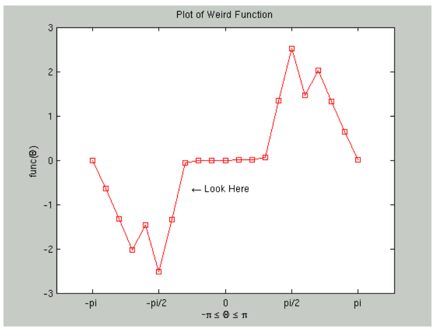
The **gca()** function returns a handle to axis information in the current figure. We can see the properties that can be changed using:

```
get( gca() )
```

18

Adding Titles, Axis Labels, and Annotations

Matlab enables you to add axis labels and titles. For example, using the graph from the previous example, we can add an x- and y-axis label, a title and an annotation:



```
xlabel('-\pi \leq \Theta \leq \pi')

ylabel('func(\Theta)')

title('Plot of Weird Function')

text(-pi/4,sin(-pi/4), '\leftarrow Look Here')
```

See doc plot for much more information about plot()

Plotting in Matlab

As mentioned above, plotting is a large topic in Matlab, too big to cover in depth here. Besides basic plotting functions such as **plot()** there are families of plotting functions covering:

- Area, Bar, and Pie Plots
- Contour Plots
- Direction and Velocity Plots
- Discrete Data Plots
- Function Plots
- Histograms
- Polygons and Surfaces
- Scatter/Bubble Plots

More specialised image plots can be accessed from the Matlab Image Processing Toolbox

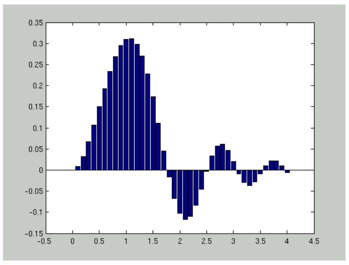
(This is pathed in by default for us).

For more details consult the Matlab online documentation

Plotting Demos – Bar Charts

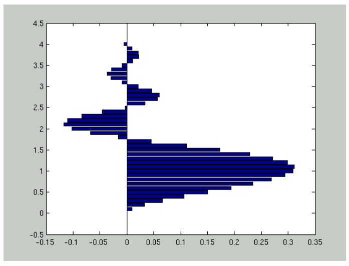
Standard (grouped) vertical bar chart:

```
x = 0:0.1:4;
y = sin(x.^2).*exp(-x);
bar(x,y);
```



Standard (grouped) horizontal bar chart:

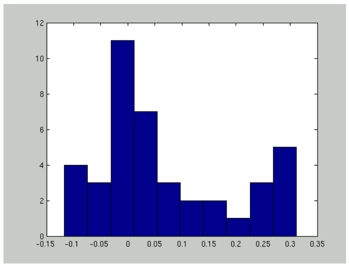
```
barh(x,y);
```



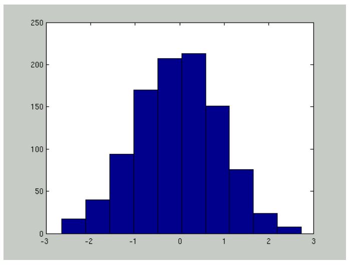
Plotting Demos – Histogram

Standard histogram using the same data set from previous slide:

```
x = 0:0.1:4;
y = sin(x.^2).*exp(-x);
hist(y)
```



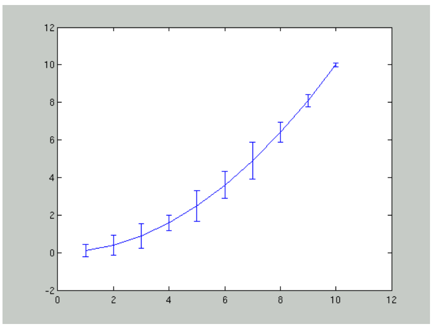
Or from a Normal distribution:
`hist(randn(1000,1))`



Plotting Demos - Errorbars

Define:
x=1:10;
y=x.*x/10;
e = rand(size(x));

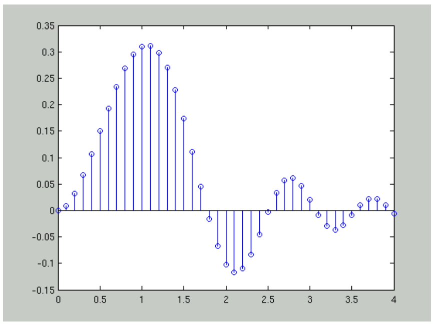
Can now plot x vs y
with errors in vector e:
errorbar(x,y,e)



Plotting Demos – Stem Plot

Define:
x = 0:0.1:4;
y = sin(x.^2).*exp(-x);

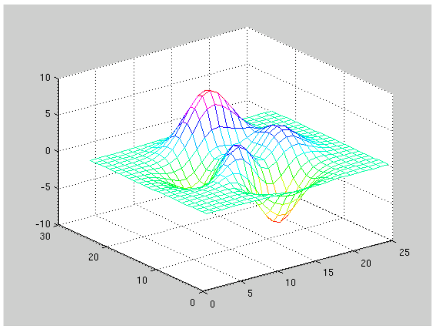
Can do stem plot as:
stem(x,y)



Plotting Demos – 3D Mesh Plot

Can do 3D mesh plot as:

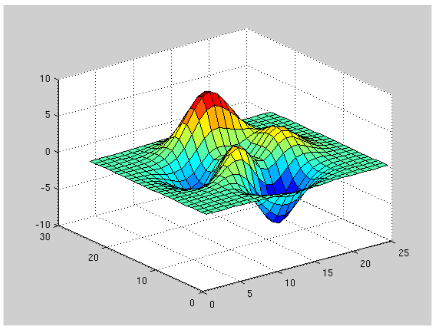
```
z=peaks(25);
mesh(z);
colormap(hsv)
```



Plotting Demos – 3D Surface Plot

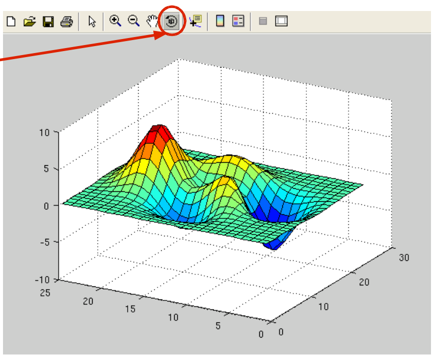
Can do 3D surface plot as:

```
z=peaks(25);
surf(z);
colormap(jet);
```



Plotting Demos – 3D Surface Plot

Can interactively rotate the 3D surface with this button



MRI data

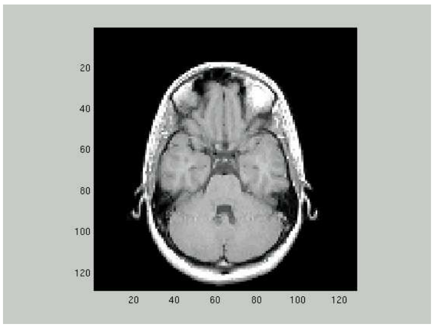
Matlab contains a sample demo T1 weighted MRI image. This particular MRI data file, D, is stored as a 128-by-128-by-1-by-27 array. In Matlab the 3rd dimension for an image is typically used for the image colour data. However, in this case there is no actual information in the 3rd dimension, which can therefore be removed using the squeeze() function. This results in a more standard 3d array for the T1 image with size 128-by-128-by-27. It can be accessed as:

```
load mri
D = squeeze(D);
whos
>>  Name      Size          Bytes    Class    Attributes
>>  D        128x128x27    442368   uint8
>>  map       89x3          2136     double
>>  siz      1x3            24       double
```

MRI data

To display one of the slices in the MR images, say the 8th, you can use the image command, indexing into the 3d volume to obtain the eighth slice. Then adjust axis scaling, and install the MRI colormap, which was loaded along with the data.

```
slice=D(:, : ,8);
image(slice)
axis image
colormap(map)
```



For more information:
`doc image`

MRI data in NIFTI format

To use typical MRI data such as that from BUIC we need to use a toolbox that can read and write data in NIFTI format. There are several such toolboxes available. A simple one is the “Tools for NIfTI and ANALYZE image” Toolbox by Jimmy Shen which can be downloaded here:

https://www.buic.bham.ac.uk/buic/teaching/matlab/NIfTI_20140122.zip/at_download/file

Open a web browser outside of Matlab, go to this web page and download this file. The toolbox is a zip file. Open the zip file and extract the files inside it. Save them into a new subdirectory you have created called:
U:\MATLAB\nii

Add U:\MATLAB\nii to your matlab path using the pathtool command.

MRI data in NIFTI format

We now need to copy over some MRI data in NIFTI format to test with.

Outside of Matlab, create a directory called U:\data

Outside of Matlab, open a web browser and save the following two MRI files in your U:\data directory:

www.buic.bham.ac.uk/teaching/matlab/highres.nii.gz
www.buic.bham.ac.uk/teaching/matlab/filtered_func_data.nii.gz

MRI data from BUIC

To display typical MRI data such as that acquired locally in BUIC, first read in the data using the NII toolbox load_untouch_nii( ) function as follows:

```
T1 = load_untouch_nii('U:\data\highres.nii.gz');
```

Have a look at what is created in the T1 data.
You should be able to recognize this as a Matlab structure.

The actual image data is present in the T1 structure element .img as can be seen with:

```
size(T1.img)
```

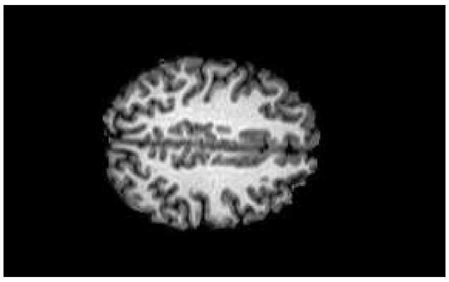
MRI data

To display one of the axial slices in the image, say the 200th slice, you can use the imshow() command, indexing into the data array to obtain the 200th slice in the z-dimension.

```
slice=T1.img(:, :, 200);

imshow(slice,[ ])
```

The 2nd parameter passed to imshow() - empty square brackets – tells it to scale the display between the minimum and the maximum values of intensity in the image



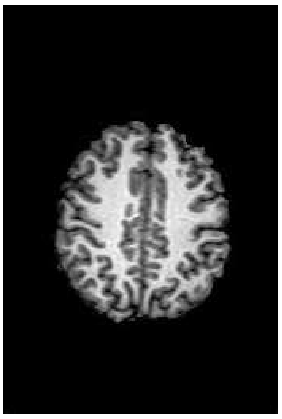
For more information:
doc imshow

MRI data

You may have noticed that the displayed image is not quite in the orientation you might have expected. The 3rd dimension of the loaded image is definitely the z-axis but the x and y dimensions have been flipped between dimensions 1 and 2. This can be corrected using the rot90() function:

```
slice=T1.img(:,:,200);
slice=rot90(slice);
imshow(slice,[ ])
```

Counterclockwise 90º rotation of slice



For more information:
doc rot90

Generic Images

Matlab has a pair of functions called **imread()** and **imwrite()** that are of general use when reading in or writing out bitmap images. These functions support a variety of image formats such as jpeg, bmp, gif, tif, png etc.

However, before using these functions it is important to realise that Matlab internally can represent images in a number of different ways:

- Indexed images
- RGB images
- Intensity images
- Binary images

Indexed Images

Indexed images consist of 2 **separate** arrays:

- A 2d image matrix, and
- A m-by-3 colormap matrix

Each pixel in the image matrix contains a single value that is an **index** into the colormap.

The colormap is an ordered set of values that represent the colors in the image. It is an m-by-3 matrix where m=the number of defined colors. Each individual row of the colormap matrix specifies 3 values for the red, green and blue (RGB) components of a single color. R,G,B are numbers that range from 0.0 (nothing) to 1.0 (full intensity).

colormap = [R G B];

Indexed Images

For example:

```
[x1,m]=imread('canoe.tif');
whos
>>  Name  Size       Bytes   Class  Attributes
>>  m        256x3     6144   double
>>  x1        207x346 71622  uint8

m(100,:)
>> ans  =  0.6471    0.5176    0.3216
```

Load an index image returning image matrix (x1) and colormap (m)

Show the size of the variables retrieved

Show a typical entry in the colormap

Amount of red present (0-1)

Amount of green present (0-1)

Amount of blue present (0-1)

```
imshow(x1,m)
```

Show the image and apply the colormap

RGB Images

Like an indexed image, an RGB image represents each pixel color as a set of three values representing the red, green and blue intensities that make up the color. Unlike an indexed image, however, these intensity values are stored directly in the same array and not indirectly in a secondary colormap. RGB images are therefore represented by a **single** data matrix.

In Matlab, RGB images are always matrices of size (r x c x 3) where r and c = the number of rows and columns of pixels in the 2d image.

The 3rd dimension can be thought of consisting of three planes, containing the red, green and blue intensity values. For each pixel in the 2d image, these red, green and blue elements are combined to produce the pixel's actual color.

RGB Images

For example:

```
x2=imread('autumn.tif');
whos
>>  Name  Size          Bytes   Class  Attributes
>>  x2      206x345x3  13210  uint8
```

Load an RGB image

Show the size of the variables retrieved

```
x2(100,100,:)
>> ans(:,:,1) =   56
>> ans(:,:,2) =   51
>> ans(:,:,3) =   35
```

Show a typical entry in the image

```
imshow(x2)
```

Show the image

Intensity Images

Intensity images are like RGB images except they are in greyscale rather than color and are therefore matrices of size (m x n). Each pixel value directly represents the grey intensity (either in the range 0-1 if the data is of type double or in the range 0-255 if of type uint8). The intensity 0 represents black and 1 (or 255) white.

For example:

```
x3=imread('cameraman.tif');
whos
>>  Name       Size           Bytes  Class    Attributes
>>  X3       256x256          65536  uint8

x3(128,128)
>> ans = 160

imshow(x3)
```

Binary Images

In a binary image each pixel assumes only one of two discrete values – either on or off. A binary image is stored as a 2d matrix of 0's (off) and 1's (on).

For example:

```
x4=imread('circbw.tif');
whos
>>  Name        Size          Bytes  Class     Attributes
>>  x4        280x272         76160  logical

x4(140,140)
>> ans = 1

imshow(x4)
```

Saving Images

imwrite() can be used to save an image file in a format of your choosing. Note though that you generally need to specify both the type of output file (bmp, jpg etc) and the output file name. Because the image file can be in several formats (indexed, BW etc) the **imwrite()** function can either be:

`imwrite(img, filename, format)` or as

`imwrite(img, map, filename, format)`

Where **img**=image file data, **map**=colormap, **filename**=name of output file and **format**=the type of file (in quotes) such as 'bmp'. Some of the file formats may take further options.

See `doc imwrite` for full details.

Saving Images - Examples

```
[x1,map]=imread('canoe.tif');
imwrite(x1, map, 'image1.bmp', 'bmp')
```

Load indexed TIF image, save as BMP file

```
x2=imread('autumn.tif');
imwrite(x2, 'image2.jpg', 'jpg')
```

Load RGB TIF image, save as JPG file

or

```
imwrite(x2, 'image3.jpg', 'jpg', 'Quality', 10)
```

Save as compressed (10/100) JPG file

```
x3=imread('cameraman.tif');
imwrite(x3, 'image4.png', 'png');
```

Load intensity TIF image, save as PNG file

Saving Images

One way of using **imwrite()** is to create an image in Matlab as a matrix, then perform operations on the matrix to create a complex image, before saving it as a bitmap image for use elsewhere.

For example:

```
clear all
for x=1:256
  for y=1:256
     img(x,y)=exp(-((x-128)^2+(y-128)^2)/5e3);
   end
end
imshow(img)
imwrite(img,'gaussian.bmp','bmp');
```

This produces a 2d Gaussian blob as an intensity image

This is a simple example of how you can use Matlab to create complex visual stimuli which you can then save as image to be used later in an experiment

For next week

- Make sure you can do the Exercises here
- If using “MATLAB for Behavioural Scientists” (Rosenbaum), you should now have covered roughly all the material up to the end of Chapter 10. Note that there's more in the book than we have covered in the workshops.
- **Self Test Quiz should now be available on Canvas. Have a go before next week.**

Matlab Workshop 7

Exercises

Revision: 13 Oct 2017

Exercises 7.1

Clear the workspace (clear all) and close all open figure windows (close all). Create:
x = [-2*pi:0.1:2*pi];

- Plot x against cos(x) in figure 1 using the default settings (simple line plot, solid blue line)
- Now replot x against cos(x) in figure 1 but with a solid black line
- Open figure 10 and plot x against sin(x) as a set of green points only (with no line joining the points up)
- Close figure 10, keeping figure 1 open, and reselect figure 1
- Change the name of figure 1, adding 'TEST' to the figure name

Exercises 7.2

- In figure 1, whilst keeping the black line, overplot the figure with red circles where the points are
- Add to the existing plot sin(x) as a dashed blue line
- Change the x-axis to show only the range 0 to pi.
- Change the x-axis tic marks and labels to go from 0 to pi in steps of pi/4
- Add a plot title called 'My favourite plot', an x-axis label called 'x-axis' and a y-axis label called 'y-axis'

Exercises 7.3

- Clear the workspace and close all open figure windows
- Load in the sample fMRI data file from U:\data\filtered_func_data.nii.gz
- Select the 25th axial slice (z-dimension) of the 10th volume and display this in the correct orientation
- Select the 40th coronal slice (y-dimension) of the 100th volume and display this in the correct orientation
- Select the 30th sagittal slice (x-dimension) of the 200th volume and display this in the correct orientation

Exercises 7.4

- Load the image file called street1.jpg
- In Matlab terms, what sort of an image is it:
  Indexed, RGB, Intensity or BW image?
- Save it as a PNG file called street1.png
- Make sure you can view the PNG file outside Matlab with a non-Matlab viewing program

Exercises 7.5

- Clear the workspace and close all open graphics windows
- Load the RGB image autumn.tif into the variable x
- Crop the image to be size 206x206 keeping the left hand side of the image and keeping it in color. Save this image to variable y. Display image y.
- Convert the image to be greyscale. Save this image to variable z. Display image z.

Matlab Workshop 7

Suggestions for Answers to Exercises

Revision: 13 Oct 2017

Exercises 7.1

Clear the workspace (clear all) and close all open figure windows (close all). Create:
x = [-2*pi:0.1:2*pi];

- Plot x against cos(x) in figure 1 using the default settings (simple line plot, solid blue line)
- Now replot x against cos(x) in figure 1 but with a solid black line
- Open figure 10 and plot x against sin(x) as a set of green points only (with no line joining the points up)
- Close figure 10, keeping figure 1 open, and reselect figure 1
- Change the name of figure 1, adding 'TEST' to the figure name

Exercises 7.1

Plot x against the cos(x) in figure 1 as a simple line plot
figure(1); plot(x,cos(x))

Now replot x against cos(x) in figure 1 but with a solid black line
figure(1);plot(x,cos(x),'k')

Open figure 10 and plot x against sin(x) just as a set of green points (with no line joining them up)
figure(10); plot(x,sin(x),'g.')

Close figure 10, keeping figure 1 open, and reselect figure 1
close(10)
figure(1)

Exercises 7.1

Change the name of figure 1, adding 'TEST' to the figure name

Either:
H = gcf();
H.Name = 'TEST'

or:
H = gcf();
set(H,'Name','TEST')

or:
set(1,'Name','TEST')

Exercises 7.2

- In figure 1, whilst keeping the black line, overplot the figure with red circles where the points are
- Add to the existing plot sin(x) as a dashed blue line
- Change the x-axis to show only the range 0 to pi.
- Change the x-axis tic marks and labels to go from 0 to pi in steps of pi/4
- Add a plot title called 'My favourite plot', an x-axis label called 'x-axis' and a y-axis label called 'y-axis'

Exercises 7.2

In figure 1, whilst keeping the black line, overplot the figure with red circles where the points are

```
figure(1)
hold on
plot(x,cos(x),'ro')
hold off
```

Add to the plot sin(x) as a dashed blue line

```
hold on;
plot(x,sin(x),'b--')
hold off;
```

Change the x-axis to go from 0 to pi.

```
xlim([ 0 pi ] )
```

Exercises 7.2

Change the x-axis tic marks and labels to go from 0 to pi in steps of pi/4

```
set(gca,'XTick',0:pi/4:pi)
set(gca,'XTickLabel',{'0','pi/4','pi/2', '3*pi/4', 'pi'})
```

Add a plot title called 'My favourite plot', an x-axis label called 'x-axis' and a y-axis label called 'y-axis'

```
title('My favourite plot')
xlabel('x-axis')
ylabel('y-axis')
```

Exercises 7.3

- Clear the workspace and close all open figure windows
- Load in the sample fMRI data file in U:\data\filtered_func_data.nii.gz
- Select the 25th axial slice (z-dimension) of the 10th volume and display this in the correct orientation
- Select the 40th coronal slice (y-dimension) of the 100th volume and display this in the correct orientation
- Select the 30th sagittal slice (x-dimension) of the 200th volume and display this in the correct orientation

Exercises 7.3

- Load in the sample fMRI data file

```
fMRI = load_untouch_nii('U:\data\filtered_func_data.nii.gz');
```

- Select the 25th axial slice (z-dimension) of the 10th volume and display this in the correct orientation

```
slice = fMRI.img(:, :, 25, 10);
slice = rot90(slice);
imshow(slice, [ ])
```

Exercises 7.3

- Select the 40th coronal slice (y-dimension) of the 100th volume and display this in the correct orientation

```
slice = fMRI.img(:, 40, :, 100);
slice = squeeze(slice);
slice = rot90(slice);
imshow(slice, [ ])
```

Exercises 7.3

- Select the 30th sagittal slice (x-dimension) of the 200th volume and display this in the correct orientation

```
slice = fMRI.img(30, :, :, 200);
slice = squeeze(slice);
slice = rot90(slice);
imshow(slice, [ ])
```

Exercises 7.4

- Load the image file called street1.jpg
- In Matlab terms, what sort of an image is it:
  Indexed, RGB, Intensity or BW image?
- Save it as a PNG file called street1.png
- Make sure you can view the PNG file outside Matlab with a non-Matlab viewing program

Exercises 7.4

Load the image file called street1.jpg
[x,m] = imread('street1.jpg');

In Matlab terms, what sort of an image is it?
Indexed, RGB, Intensity or BW?

```
whos
  Name        Size              Bytes  Class     Attributes
  m           0x0                   0  double
  x         480x640x3          921600  uint8
```

The structure of x (with the third dimension being of size 3), and that fact that m is returned as an empty array makes it fairly clear street1.jpg is an RGB file.

Exercises 7.4

- Save it as a PNG file called street1.png
  imwrite(x, 'street1.png', 'png')

- Make sure you can view the PNG file outside Matlab with a non-Matlab viewing program.
  For example right click on it and select **Preview**.

Exercises 7.5

- Clear the workspace and close all open graphics windows
- Load the RGB image autumn.tif into the variable x
- Crop the image to be size 206x206 keeping the left hand side of the image and keeping it in color. Save this image to variable y. Display image y.
- Convert the image to be greyscale. Save this image to variable z. Display image z.

Exercises 7.5

- Clear the workspace and close all open graphics windows
  clear all
  close all
- Load the RGB image autumn.tif into the variable x
  x = imread('autumn.tif');

Exercises 7.5

- Crop the image to be size 206x206 keeping the left hand side of the image and keeping it in color. Save this image to variable y. Display image y.

```
y = x(:,1:206,:);
imshow(y)
```

- Convert the image to be greyscale. Save this image to variable z. Display image z.

  Any of these should work:

```
z = mean(y,3)/256;
z = uint8(round(mean(y,3)));
z = rgb2gray(y);
imshow(z)
```

Matlab Programming Module

Dr Peter Hansen
p.c.hansen@bham.ac.uk

School of Psychology / BUIC

Workshop 8
Introductory Programming Exercises

Revision: 13 Oct 2017

Programming Exercise

Overview Expt 1

The objective for today is to program two small experiments using the methods you have learned in earlier workshops:

For the first experiment you will need to:

- Load a series of images from files on disk
- Present each of these on screen
- Prompt the user with a question for each image

The image files consist of a series of pictures of people's faces, where the image has been cropped so that only the region around the eyes is visible. The question that will be asked for each image is: “What is the age of the person?” The user will need to estimate the age (to the nearest year) and enter it.

Overview

There are 18 image files. These can be found in:

www.buic.bham.ac.uk/teaching/matlab/eyes.zip

Download this file, unzip the contents, and save the files in a new directory called U:\matlab\eyes. The files are labelled as:

eyes_01.bmp
eyes_02.bmp
eyes_03.bmp
eyes_04.bmp
…
etc
...
eyes_18.bmp

Overview

Once you have successfully programmed the basic structure of the program we may want to extend its functionality in order to have it do more things later.

It is important to take this need for flexibility into account when programming the basic version of the program. Otherwise the structure you adopt may mean you have to substantially rewrite it to accommodate later additions.

To extend the basic experiment we will later want to do things such as randomising the order in which the images are presented and saving the user responses in a text file.

Step 1

The first step is to **plan** how to do this.

This part of the exercise does not need a computer nor should it require you to write any actual Matlab code. Your best tools will be a blank sheet of paper and a pen.

You will need to conceptually map out the basic set of operations that are needed to complete this task and the order in which they should happen. This is your program outline. To begin with, focus on just the basic task (not the extensions mentioned above).

The first thing you should think about is how you are going to break the task into modules and functions. How many do you need? What does each one of these need to do?

Step 1

Other things you need to consider before starting:

- What kind of image file types are the pictures?
- How will you load them? With what Matlab function?
- Note that the images are **not** all the same size.
  How will you deal with that?
- In what Matlab form will you store them prior to displaying them?
- How will you display them? With what Matlab function?
- How will you get user input? With what Matlab function?

Step 1

At the end of this stage you should have a very good idea in your program outline of what needs to be done in Matlab and the structure of how it should be implemented.

It is up to you to decide what is an appropriate level of detail here. Some people prefer to write out in detail using meta-code very precisely what they want the particular parts of the program to do. If you are new to Matlab than the more thought and detail you can provide here the easier you will find the next stage.

For example you need to be clear about what Matlab structures and functions you are going to need. For this, you may want to refer back to the previous workshops to identify the most relevant code fragments that you may want to use. All previous workshops can be found on Canvas.

Step 2

The next step is to start implementing your project outline.

To do this go back to working with Matlab on the computer and start trying to convert your pen and paper outline into actual Matlab code.

If you are stuck and do not know what to do here, then this is a sign that you have not specified enough detail in the program outline. Go back to the Step 1 and provide more details for the program outline until you are happy you know how to proceed.

You may find unexpected problems as you start to implement your program outline. There are likely to be issues that you have not thought of. This is normal. See if you can overcome them by using the online help and by referring to previous Workshops.

Step 2

Make sure that **ALL** of your program code (procedures and functions) are saved in your individual U:\MATLAB directory

Remember that you must save your .m files in this directory if Matlab is to be able to find and run them.

It is a good idea also to adopt a very clear naming convention for your files. For example your files could all begin with **eyes_** (since this is an experiment involving images of eyes). The top level program for the first experiment might then be called

eyes_expt1.m

Make sure the final program runs through as expected and you can display all 18 images and get a user response for each one.

Step 3

Once you are satisfied that the basic task requirements have been successfully met, let's extend the program:

Firstly, we want to randomise the order in which the images are presented. So now we do not want eyes_01.bmp to always be the first image presented. We want a random one of the 18 to be shown first, followed by a random one of the other 17 and so on.

How are you going to do this? What functions have you seen in previous workshops that allow you to create randomised sets of numbers? Once you have randomised set of numbers, how are you going to apply this to randomly select the image?

Once you have an idea about the above, amend your Matlab program and test it works.

Step 3

Let's extend the program again:

Secondly, we want to save the user responses in a file. So now rather than just displaying the user input on the screen we also want to save the guessed ages in an external text file. Since we've now randomised the order in which the images are displayed we also had better save in the text file the identity of the image file (either its number or its filename).

How are you going to do this? What functions have you seen in previous workshops that allow you to write to text files and to save numbers and char strings in these files?

Once you have an idea about the above, amend your Matlab program and test it works.

Overview Expt 2

For the second experiment you will need to:

- Load the same series of images from files on disk
- Combine pairs of these images together to form a new image
- Present this new image on the screen
- Prompt the user with a question for each image

The input image files are the same series of pictures of people's eyes as before. However, you will now initially need to combine these so that one pair of eyes appears above the other in a single new image. This requires you to build on what you have learned in earlier workshops.

The question that will be then asked for each new image is "Which image shows the younger person?" The user will need to choose whether the lower or the upper image is the younger one.

Step 1

If you were starting from scratch then you would need to proceed exactly as with Expt 1.

However, provided you have constructed a reasonably modular program for Expt 1, it should be easy to make a copy of the code you created for Expt 1 and to add and amend it to do what is needed for Expt 2.

For each of the programs and files you have created for Expt 1, such as eyes_expt1.m copy the file and rename it as eyes_exp2.m

Step 1

Here you need to consider the main changes between Expt 1 and Expt 2 and what you will need to do differently before starting:

- How to combine pairs of images? How will they be stored and accessed prior to displaying them?
- Remember that the images are **not** all the same size. Programmatically, how will you best deal with this?
- There are potentially 18*17 = 306 unique pairs of images (excluding the identical comparisons). Will you generate all image pairs in advance or generate each pair on-the-fly?
- How will you get user input? What sort of input? What keys?

Step 2

As with Expt 1 the next step is to start implementing your code changes. Let's assume for now we will only show 20 typical image pairs rather than all 306 possible image pairs.

Make sure that **ALL** of your program code (procedures and functions) are saved in your individual U:\MATLAB directory

Remember that you must save your .m files in this directory if Matlab is to be able to find and run them.

The top level program for the second experiment might best be called **eyes_expt2.m**

Make sure the final program runs through as expected and you can display 20 different image pairs and get a user response for each one.

Step 3

Once you are satisfied that the basic task requirements have been successfully met, let's extend the program:

Firstly, we want to randomise the order in which the image pairs are presented. Now we do not always want, say, image1 + image2 to be the first image presented. Each time we run the program we want a different random one of the 306 possible image pairs to be shown first, followed by a random one of the next 305 and so on until 20 random pairs have been shown.

How are you going to do this? What functions have you seen in previous workshops that allow you to create randomised sets of numbers? Once you have randomised set of numbers, how are you going to apply this to randomly select the image?

When you are clear about the above, amend your Matlab program and test it works.

Step 3

Let's extend the program again:

Secondly, we want to save the user responses in a file. So now rather than just displaying the user input on the screen we also want to save the chosen responses in an external text file. Since we've now randomised the order in which the images are displayed we also had better save in the text file the identity of the image file (the id of the two images that make up the composite image).

How are you going to do this? What functions have you seen in previous workshops that allow you to write to text files and to save numbers and char strings in these files?

Once you have an idea about the above, amend your Matlab program and test it works.

Both Programs – Optional Steps

For both programs, at this stage you should have a working program that shows images in a random order and that saves the user responses to a text file.

But there are several things we could do to enhance each program. Try and add the following improvements to your programs:

- Before the images are displayed, show some introductory help text when the program starts up to explain to the user what the task is and what they have to do
- Before the images are displayed, prompt the user for the name of the text file in which they are going to save the response data

Both Programs – Optional Steps

- Ensure that the user input (choice of image) is verified:
  - For Expt 1 check that it is actually a number, not text or empty. Check that the number lies between 1-100.
  - For Expt 2 check that the response is as expected and valid (if using keyboard entry check for case sensitivity)
- Deal with the case where the saved text file cannot be written to (for example by trying to write to C:\response.txt). Warn the user and then prompt the user for a new filename.
- Deal with the case where the saved text file already exists. Prompt the user as to whether they want to overwrite it or to write a new file (and then prompt the user for a new filename).
- Extend the program so that it can be called with a parameter N: e.g. eyes_main(N) so that the program only shows N random images.

For next week

- Make sure you can complete the Introductory Programming Exercise here
- If using “MATLAB for Behavioural Scientists” (Rosenbaum), you should now have covered roughly all the material up to the end of Chapter 10. Note that there is more in the book than we have covered in the workshops.
- Answers to Matlab Self Test Quiz should now be on Canvas

Matlab Workshop 8

Suggestions for Answers to
Introductory Programming Exercises

Revision: 13 Oct 2017

Expt 1: main program: eyes_expt1.m

```
function eyes_expt1()
% EYES_EXPT1()
%
% Program to load in and show images of eyes,
% get a user response and save to a file

% Number of images to process
N=18;

% Load in random N of 18 possible eye images as a cell array
eyes=eyes_load(N);

% Display text
clc
disp('You are going to see a series of pictures of partial faces');
disp('You will need to estimate the age of each person shown');
disp('Press any key to start');
pause();

% Open file to save responses in
fid=fopen('U:\data\eyesdata.txt','w+');

% Main loop
f=figure(1);
for i=1:N
    imshow( eyes{i} );
    shg;
    txt=sprintf('%d/%d: What is the age of the person?: ', i,N);
    age=input(txt);
    fprintf(fid,'Image: %d\tAge: %d\n', i, age);
end

% Close save file
fclose(fid);
```

Note this assumes you have created an output **U:\data** directory in which output saved data like this can be stored

Note also that we have **not** dealt with the problem of the file already existing or not having permission to write into the directory

Note that we have also **not** dealt with the problem of the user input being incorrect or invalid

Helper function to load in images

```
function img=eyes_load(N, randord)
% IMG = EYES_LOAD(N, randord)
% Load up N eyes images (where N=1 to 18, defaults to 18)
% randord=Boolean flag. true=randomize order, false=no randomization. Default = false
%
% IMG is a cell array containing N random images from 18

% If N has not been set, default it to 18
if ~exist('N','var')
    N=18;
end

% Check that N is plausible
if N<1 | N>18
  error('Number of images selected out of range')
end

% If randorg has not been set, default it to false
if ~exist('randord','var')
    randord=false;
end

% Populate the all_images cell array with all 18 images
for i=1:18
  file = sprintf('U:\\matlab\\eyes\\eyes_%02d.bmp', i);
  all_images{i} = imread(file);
end

% Randomize order and select only the first N images
if randord
   idx = randperm(18);
   img = all_images( idx(1:N) );
else
  img = all_images;
end
```

Note that this bit of code assumes you know both the directory you are getting the files from, that there are 18 files (at most) and that they have the particular naming convention:
eyes_01.bmp
eyes_02.bmp
…
etc
...
eyes_18.bmp

This is not ideal, it is merely adequate for the purpose here.

Alternate version of helper function to load in images

```
function img=eyes_load(N, randord)
% IMG = EYES_LOAD(N, randord)
% Load up N eyes images (where N=1 to 18, defaults to 18)
% randord=Boolean flag. true=randomize order, false=no randomization. Default = false
%
% IMG is a cell array containing N random images from 18

% Directory name where image files are
dirname='U:\matlab\eyes\';

% If N has not been set, default it to 18
if ~exist('N','var')
    N=18;
end

% Check that N is plausible
if N<1 | N>18
  error('Number of images selected out of range')
end

% If randorg has not been set, default it to false
if ~exist('randord','var')
    randord=false;
end

% Get names of image files and populate all_images cell array
d=dir( [ dirname '*.bmp' ] );
for i=1:length(d)
  file = [ dirname d(i).name ];
  all_images{i} = imread(file);
end

% Randomize order and select only the first N images
if randord
   idx = randperm(18);
   img = all_images( idx(1:N) );
else
  img = all_images;
end
```

Note that the **dir()** function will return **d** as an 18x1 Matlab struct array with fields:
- name
- date
- bytes
- isdir
- datenum

Expt 2 - Outline

We can think of Expt 2 as being a slightly trickier version of Expt 1. Essentially we can repeat most of the code that we have written for Expt 1 in Expt 2. The part that is different is that initially we need to preprocess the 18 images that have been loaded in the cell array.

We need to combine them into new images drawn from the 18x17 possible pairs. The simplest way to achieve this is to construct 306 new images using pairs of images. The problem with doing this is that the original images are all different sizes. Therefore, for this program we need to get around that problem in some way. You may have resized or rescaled images to achieve Expt 1. Or like me you may have handled them using a cell array and kept them in the native sizes. Now we need to manipulate the images.

Firstly, let us check the image sizes using one of the versions of the helper eyes_load() functions from Expt 1 :

```
img=eyes_load();
for i=1:18
   rows(i)  = size(img{i},1);
   cols(i)   = size(img{i},2);
   depth(i) = size(img{i},3);
end
```

Examination of the **rows** and **cols** arrays show they come in range of sizes. **depth** is always the same.
The key numbers for us are:

```
max(rows)
>> ans = 508
```
This means the maximum height of any single image is 508 pixels.

```
max(cols)
>> ans = 873
```
This means the maximum width of any single image is 873 pixels.

Now if we imagine creating a blank new image that is of height 1050 pixels and of width 900 pixels then it should be clear that we can fit any 2 of the possible 18 original images into this space, with one image above the other.

So now our immediate problem is to write a function to do this – to take two images and put one centred in the middle of this image at the top and one centred in the middle at the bottom.

Expt 2: helper function: eyes_combine.m

```
function combined_img=eyes_combine(img1, img2)
% COMBINED_IMG = EYES_COMBINE(IMG1, IMG2)
% Take the two input images and put them together into one new composite image
%
% COMBINED_IMG is the new resultant image

MaxRows=1050;
MaxCols=900;

% Create empty image of fixed size, filled with white pixels (255,255,255)
combined_img = uint8(255*ones(MaxRows, MaxCols, 3));

% Extract sizes of these 2 images
[img1_r, img1_c, img1_d]=size(img1);
[img2_r, img2_c, img2_d]=size(img2);

% Generate coords of where to place images
row1 = round((MaxRows/2 - img1_r)/2);
col1 = round((MaxCols - img1_c)/2);
row2 = round(MaxRows/2 + (MaxRows/2 - img2_r)/2);
col2 = round((MaxCols - img2_c)/2);

% Insert the 2 images into the new empty image
combined_img(row1:row1+img1_r-1,col1:col1+img1_c-1,:) = img1;
combined_img(row2:row2+img2_r-1,col2:col2+img2_c-1,:) = img2;

end
```

Expt 2: helper function: eye_gen_pairs.m

```
function eye_pairs=eye_gen_pairs()
% EYE_GEN_PAIRS()
% Generate a single cell array containing all possible pairs of images
%
% EYE_PAIRS is a cell array containing all 18*17 = 306 possible pairs of images

% Load in all eye images as a cell array
eyes=eyes_load();

% Empty cell array
eye_pairs = {};

%Initialize loop counter
cnt=1;

% Loop over pairs of images, excluding the case where the image is the same.
for I1=1:18
  for I2=1:18
    if (I1~=I2)
      eye_pairs{cnt} = eyes_combine( eyes{I1}, eyes{I2} );
      cnt = cnt + 1;
    end
  end
end
```

Expt 2: main program: eyes_expt2.m

```
function eyes_expt2()
% EYES_EXPT2()
%
% Program to load in and show images of eyes, get a user response and save to a file

% Generate all pairs of eye images
eyes2 = eye_gen_pairs();

% Display text
clc
disp('You are going to see a series of pairs of eyes');
disp('You will need to choose which image shows the younger person');
disp('Press the T key for the Top image or the B key for the bottom image');
disp('Press any key to start');
pause();

% Open file to save responses in
fid=fopen('P:\data\eyesdata2.txt','w+');

% Main loop
f=figure(1);
for i=1:20
    imshow( eyes2{i} );
    shg;
    txt=sprintf('%d: Which image shows the younger person?:', i);
    response=input(txt, 's');
    fprintf(fid,'Image: %d\tResponse: %d\n', i, response);
end

% Close save file
fclose(fid);
```