Introduction to MATLAB

Alisdair Tullo
2014

Hello!

- Informal lab / lecture session
- Assumes **no** prior programming experience
- Two hours, with a 10 minute break
- One-third talking and two-thirds practical work
- Feel free to ask us questions, at any time
- Feel free to help each other and discuss the exercises

Purpose

- This is an **introduction** to MATLAB
- This will help you get familiar with MATLAB and some general computer programming concepts
- Exploration is encouraged – try the examples given, and anything else that occurs to you!

Failure

- Computer programming involves lots of failure
- Usually you have to fail several times to succeed once
- This is ok and happens to everyone
- Most of the time there will be an error message, which will give you a clue to the solution for the problem

Keyboard practicalities

- Go to System Preferences -> Language and Region
- Choose Keyboard Preferences
- Click on +, and add “British-PC”
- Close this window
- At the top-right of the screen, click on the flag and make sure “British-PC” is selected

Even so ....

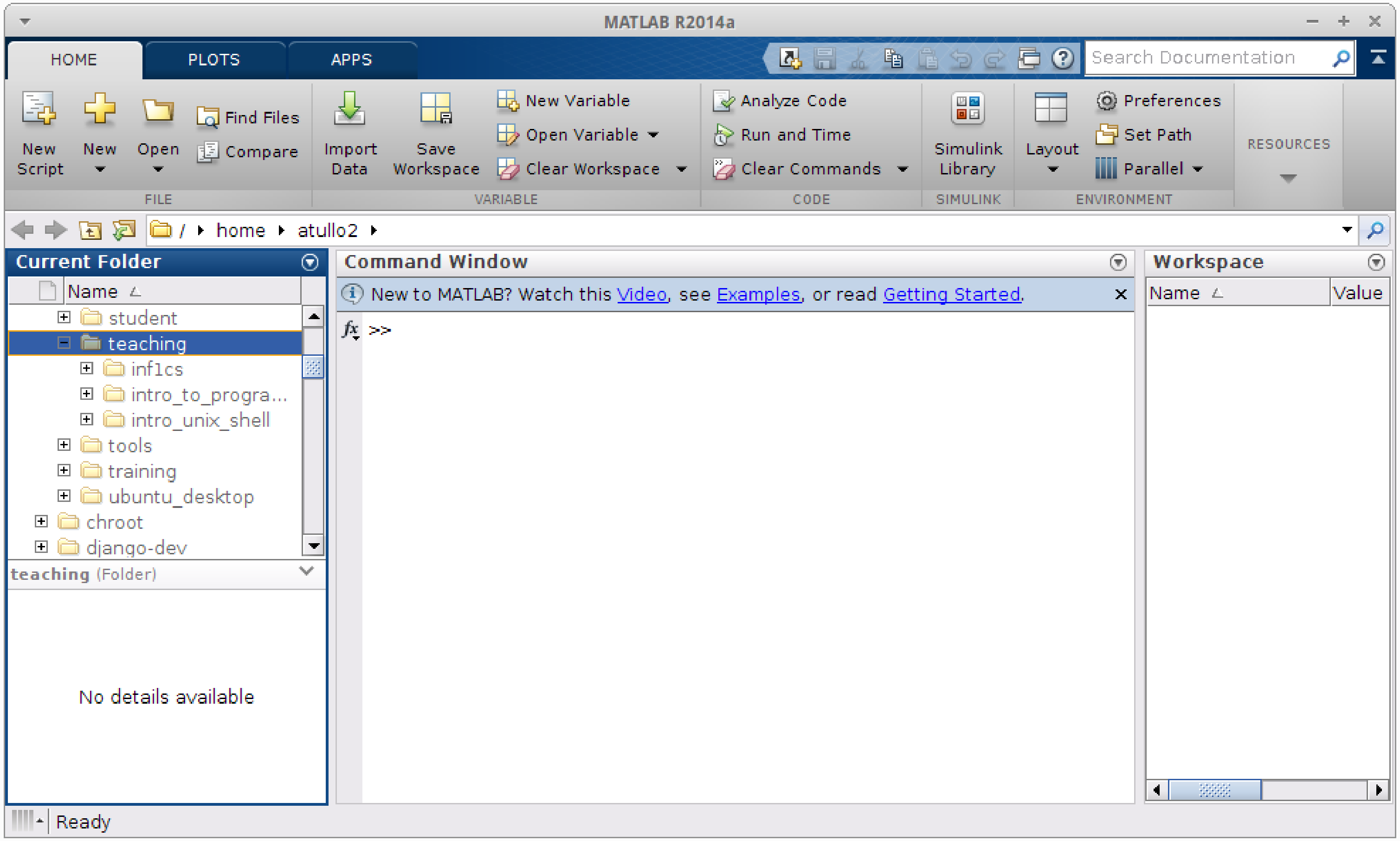
- One key is transposed on some machines!
- This is the key with:

```
|
\
```

- It's swapped with the very top-left key, which is next to 1

MATLAB

- Open MATLAB (in Applications -> Science)



The MATLAB prompt

- From now on, whenever you see this:

  >>

  it indicates something that you can type in to MATLAB
- Some of the things you type in will produce error messages
- Some of the things **I tell you** to type in will produce error messages

Saying hello to MATLAB

- Try this:

```
>> 'hello'
```

MATLAB as a calculator

```
>> 2+2
>> 2-20
>> 6*3
>> 1/10
>> 10^3
>> (2*3)+4
>> 2*(3+4)
>> 2*3+4
>> 2.5/1000000
```

Numbers

- This last answer is in “floating point” notation

  $2.5e\text{-}4 = 2.5 \times 10^{-4}$

  $= 2.5 \times 0.0001$

  $= 0.00025$

- Try these:

```
>>  8e3
>>  4.5e2
>>  1e1
```

Numbers

- There are also some “special” values you might see. MATLAB still regards these as numbers. For example:

```
>> 1/0
```

- You may also see:
  - NaN (“Not a Number”)
  - i (or j) for the square root of -1

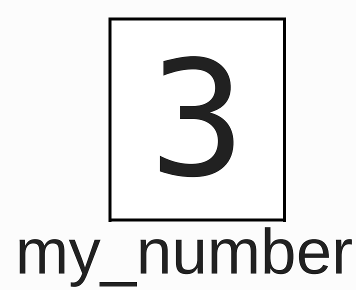
Variables

- We can create a variable using =

```
>> my_number = 3
```

- A variable is like a box for a value
- The variable name is the label on the box
- MATLAB will remember the value we give:

```
>> my_number
```



Variables

- You can use a variable with a number in it wherever you would use a number

```
>> my_number + 5
```

- You can put the result of such a calculation into another variable

```
>> another_number = my_number + 5
```

Variables

```
>> a=3
>> a
>> b=14
>> a+b
>> you_can_use_long_names = 5000
>> d = a + 20
>> i_dont_exist
```

Variables

- What can you use as a variable name?

```
>> a = 12
>> A = 0.7
>> 1number = 43
>> _things = 10
>> word count = 20
>> end = -40000
```

Variable names

- Variable names must start with a letter, and can contain letters, numbers and underscores
- Names are case sensitive
- They can't contain spaces, so what if you want to have multiple words in your variable name?

```
>> numberofthings = 12
>> numberOfThings = 12
>> number_of_things = 12
```

More variables

- There are other kinds of values in MATLAB, for example, text:

```
>> some_text = 'a line of text'
>> text2 = ' and some more text'
>> text3 = strcat(some_text, text2)
```

- or true/false values

```
>> is_ready = true
```

Types

- These different kinds of values are referred to as **types**
- Numbers – **floating point**

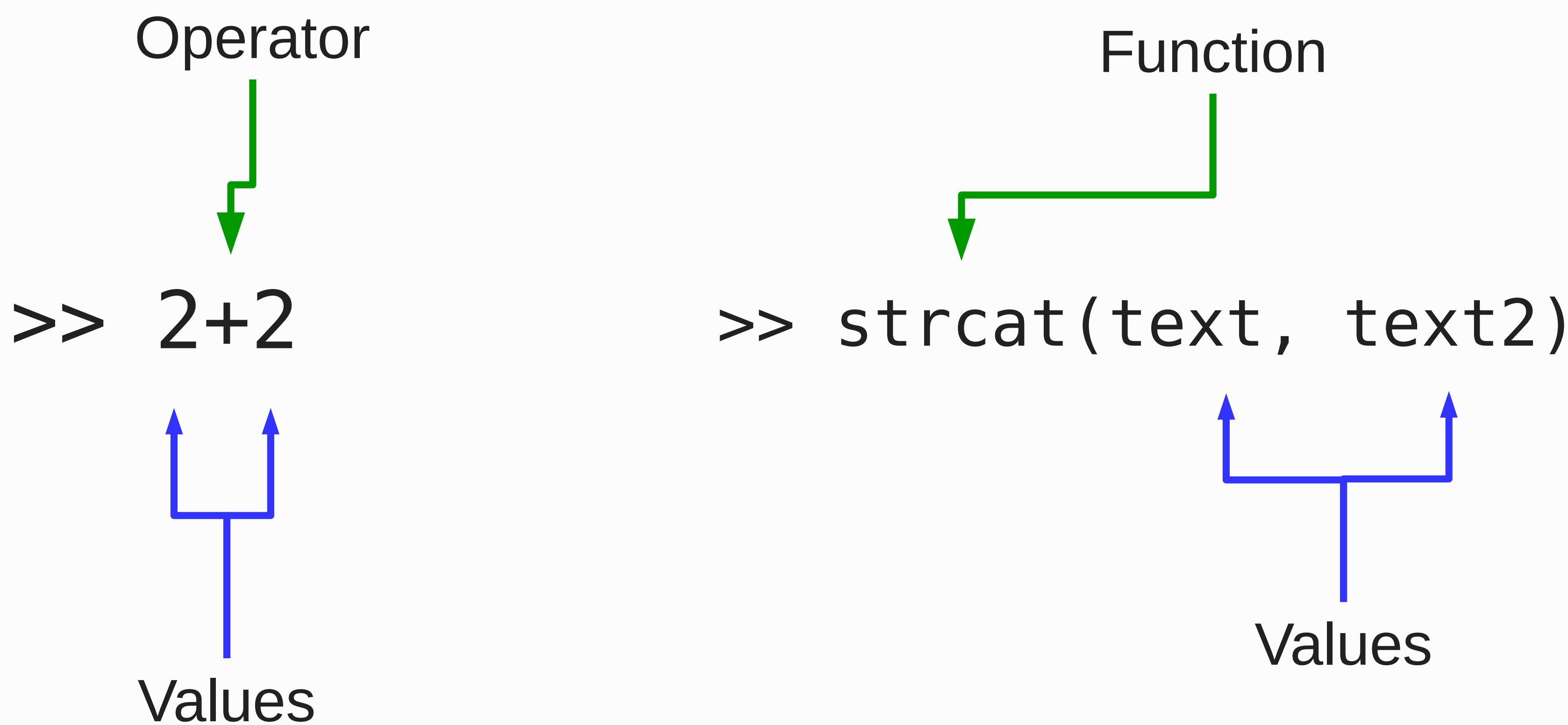
  0 -1200 5.0e20 0.0001 Inf NaN
- Text – **string**

  'hello' '1000' 'this is a text'
- True or False – **Boolean** or **logical**

  true (1), false (0)

Different ways to get results

- Functions and operators



Different ways to get results

- Despite being written differently, these do a very similar thing!
- In both cases, there are values going in, something is done with them, and there's one value going out.
- Names for the values going in: **arguments**, **parameters**, **operands**

This looks familiar!

- This is similar to running a command line program
- Program name, with parameters:
  `cp file1 file2`
- Operator, with parameters:
  `3 + 4`
- Function name, with parameters:
  `strcat('hello ',text2)`

This looks familiar! (part 2)

- The MATLAB prompt keeps track of your previous commands
- You can use the up arrow to go back through this history
- You can edit a line and run it again, or just run it again as-is
- This is exactly the same as the Unix shell

Comments

- Anything you write after a % is a comment
- This can be used to document the intent behind a piece of code
- For example, if you're doing something based on a paper, you could add a citation
```
% as per Mendel, 1865
```

Comments

- Try it!

```
>> % this will be ignored
>> a = 12   % here is my comment
```

… and you can check that this **fails** without %:

```
>> a = 12   here is my comment
```

Comments

- This can be used to temporarily disable code (more useful in code files, which we'll see later)

```
a = 16
%a = 16
```

- This is referred to as “commenting out” code

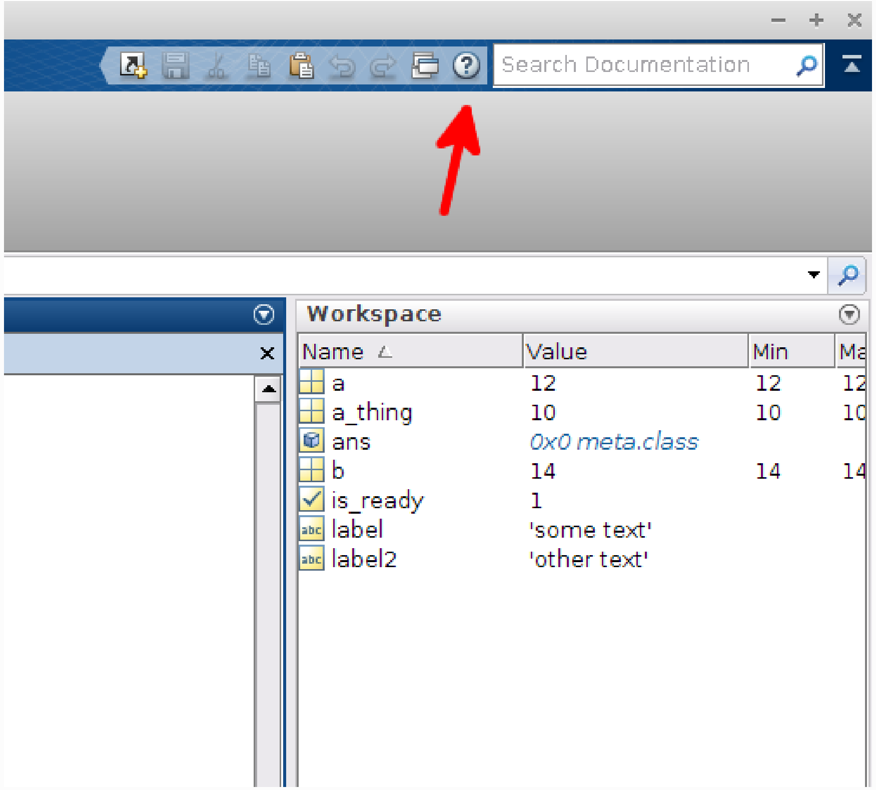
Getting help

- If you see a function you don't know, either ....
  - put the cursor on its name and press F1, or
  - right-click and choose “Help on Selection”, or

    ```
    >> doc function_name
    ```
- The only function we know so far is strcat
- Try one of these methods, to see the help for strcat
  (you may have to wait a moment!)

Getting help

- More generally, press F1, click on the help icon, or use the search box to see documentation



Getting help

- Plus, you can always ask the internet!
- The usual caveat applies: the person giving advice might have a different system to you

Showing results

- To show (print) a value in MATLAB we can just write it

```
>> result = 97
>> result
```

- This shows the variable name and the value

Not showing results

- To do something **without** showing a value, use a semicolon ';' at the end of the line

  ```
  >> result = 97;
  ```

- This still runs
- The variable 'result' will be set to 97

  ... but nothing is shown on the screen.

Printing results, disp

- Note that this also prints the variable name (or if there is none, a default “ans =”)
- To print a value without this, use disp()
- Try these and compare:

```
>> 10004
>> disp(10004)
>> 'hello!'
>> disp('hello!')
```

fprintf

- If we want more control of how values are printed we can use the fprintf function
- fprintf can print one value:

```
>> a = 20;
>> fprintf('The value of a is %d.\n',a)
```

- or many:

```
>> b = 18.0015
>> fprintf('a is %d, b is %f.\n',a,b)
```

- or none:

```
>> fprintf('Good afternoon!\n')
```

fprintf

- The first argument to fprintf is a **format string**
- This can contain a number of special codes starting with **%**
- This code specifies how to print the value

Format strings

- Format strings print the values they are given
- Special codes starting with % in the string are replaced with these values, in order

```
>> things = 8
>> fprintf('number of things: %d\n',things)

>> a = 6
>> b = 14
>> fprintf('some numbers: %d and %d\n',a,b)
```

Format codes

- These codes starting with % are also called **format specifiers**
- There are many of these, and they correspond to the **type** of the value being shown:
  - %d means “a whole number”
  - %f means “a floating-point number” (i.e. a number with a decimal point)

Format codes

- Sometimes a value can be shown in more than one way
- E.g. if the value is 18, we can print this as a whole number or a floating-point number:

```
fprintf('%f or %d ....\n',18,18)
```

Newlines

- In the format string, '\n' means **newline**
- Try this:

```
fprintf('over\nseveral\nlines\n');
```

Exercise 1

- Create two variables, **room** and **seats** and give each a value (whole numbers only)

```
>> year = 2014;
>> students = 85;
```

- Now use the 'fprintf' function to print out these numbers in a sentence.
- Your output should look like this:

```
In 2014, 85 people studied maths
```

I just want some output!

- If this seems a little complicated you can always use disp()
- disp() doesn't need you to give format codes or a newline at the end
- You can only print one thing

```
>> disp(3.001)
>> disp('Good afternoon')
```

Matrices

- A matrix is a rectangular grid containing numbers
- These can come in all sizes

3 x 1
(or '3 by 1')

49.1
-18.6
-80.2

4 x 3

7	12	-3
4	6	8
-8	0	1
-21	2	19

1 x 4

4	3	1	2

1 x 1

7

Matrix sizes

- The size of an matrix is specified by the number of rows first, then the number of columns

e.g. 5 x 3 (or '5 by 3')

3 columns

5 rows

124	-893	540
6	45	-100
712	38	464
333	0	202
118	-71	42

Arrays

- You'll sometimes see these referred to as “arrays” as well
- The individual numbers in the array are referred to as “elements”

Matrices in MATLAB

- Create a matrix using square brackets:
```
>> B = [7 12; 4 6; -8 0; -21 2]
```

This is the matrix:

7	12
4	6
-8	0
-21	2

Matrices in MATLAB

- Some more examples:

```
>> example_matrix = [1 2 3; 4 5 6]
>> C = [8 7 6]
>> D = [0; 4; 18; 22]
```

- To make them simpler to type all these examples use whole numbers; but they don't have to!

```
>> E = [0.00003 14.8; 12.7 1.8e2]
```

Writing matrices

- A semicolon indicates a new row of the matrix
- Within a row, the elements can be separated by a comma or a space

```
>> B = [7 12; 4 6; -8 0; -21 2]
```

is equivalent to

```
>> B = [7,12; 4,6; -8,0; -21,2]
```

Matrices in MATLAB

- Each of these examples creates a new variable
- Our previous variables contained a single number, or some text
- These ones contain matrices

Matrix size

- Now ask MATLAB what size of an array is

```
>> size(B)
```

- This is specified as the number of rows, then the number of columns

Exercise 2

- Create a new variable called Z containing a matrix that looks like this:

24	-83	54
6	45	-10

- This is a **2 x 3** matrix
- Use size() to check this

Vectors

- In MATLAB a vector is represented by a matrix which has either:
  - only one row (a **row vector**) or
  - only one column (a **column vector**)

Column vector

49.1
-18.6
-80.2

Row vector

4	3	1	2

Scalars

- A scalar is a single value
- In MATLAB, a scalar is treated as a 1 x 1 matrix

```
>> n = 3
>> size(n)
```

3

- This is a 1 x 1 matrix, 1 row and 1 column

Getting values out of a matrix

- We **index** a matrix by giving the row number, then the column number, of the element we want

```
>> M = [7 12 -3; 4 6 8; -8 0 1; -21 2 19]
>> M(2,3)
```

7	12	-3	1
4	6	8	2
-8	0	1	3
-21	2	19	4
1	2	3	

Exercise 3

- Work out the correct indexes to find the following numbers in the matrix M
- For example: to find -8

```
>> M(3,1)
```

- Now repeat this for **12**, **19**, and **0**
- **Remember** if you want to see what M looks like, you can type:

```
>> disp(M)
```

or just

```
>> M
```

Setting individual elements

- We can set an individual element of a matrix in a similar way

```
>> disp(M)
>> M(2,3) = 1200
>> disp(M)
```

and set it back again:

```
>> M(2,3) = 8
>> disp(M)
```

Indexing

- We can get more than one value out of a matrix (this is still called **indexing**)
- When it's used as an index, ':' means 'select all'
- Try these:

```
>> M(2,:)
>> M(:,3)
```

- What's happening here?

Selecting rows and columns

- 2,: means “second row, all columns”, and
- :,3 means “all rows, third column”

7	12	-3	1
4	6	8	2
-8	0	1	3
-21	2	19	4
1	2	3	

Exercise 4

- For M, how would you select this row?:

```
7 12  -3
```

- Again for M, select this column:

```
12
6
0
2
```

Ranges in indexing

- You can also use : to select a range
- For example, try:

```
>> M(2,2:3)
```

Ranges in indexing

- M(2,2:3) selects row 2, and columns 2 to 3

7	12	-3	1
4	6	8	2
-8	0	1	3
-21	2	19	4
1	2	3	

- What we get is the part of row 2 in columns 2 and 3

Ranges in indexing

- We can do this for columns and rows
  `>> M(2:4,1:2)`

7	12	-3	1
4	6	8	2
-8	0	1	3
-21	2	19	4
1	2	3	

Exercise 5

- For M, use this kind of indexing to select

```
7   12
4    6
```

and then

```
 4  6  8
-8  0  1
```

Comparison, True and False

- You can use == to compare numbers

```
>> 2 == 2
>> 2 == 3
```

- This is like asking a question, for example, “is 2 equal to 2?”
- The answer is given as 1 (true) or 0 (false)

= vs. ==

- Setting a variable:

  `>> n = 2`

  is a command:

  “**make** n contain number 2”

- Comparison:

  `>> n == 2`

  asks a question:

  “**does** n contain the number 2?”

Comparison

- Conversely, ~= means “not equal to”

```
>> 61 ~= 61
>> 48 ~= -99
```

- Wherever we use a number, we could also use a variable containing a number

```
>> n = 8       % set n to 8
>> n == 40     % does n equal 40?
>> n ~= 12     % does n not equal 12?
>> n == 8      % does n equal 8?
```

Comparison

- Similarly:

```
>> n = 8          % set n to 8
>> n > 3
>> n < 12
>> n >= 10
>> n <= 8
```

Comparison

- The resulting True or False value can be stored, in the same way that we store the result of any other calculation

```
>> count = 15
>> limit = 12
>> limit_passed = (count > limit)
```

Exercise 6

- Use a comparison to check whether 299 times 134 is greater than 40000
- Now do the same, but first put the numbers into variables

```
>> a = 134;
>> b = 299;
>> limit = 40000;
```

Condition

- In general, a piece of code that tests something to see if it's true or false is referred to as a **condition**
- What can we use this for?

if

- Assuming that 'a', 'b', and 'limit' are still defined from Exercise 6
- Type this:

```
>> if a*b > limit
    disp('over the limit!')
end
```

- **Note:** you can press Return after the first line, MATLAB will realise that you have more to say

if

```
if a*b > limit
    disp('over the limit!')
end
```

- The value of the expression after the 'if' is computed

```
a*b > limit
```

- If this condition is true, then the code between 'if' and 'end' is run
- Otherwise, we jump straight to the code after 'end'

if

- Let's check what happens in the other case, when the condition is false
- Change something so that
  `a*b > limit`
  is false
  (for example, you could set 'limit' to 50000)
- Now run the 'if' code again:

```
if a*b > limit
    disp('over the limit!')
end
```

Code files

- So far we've only typed code in to the MATLAB prompt
- You can also type a sequence of commands into a file to make a MATLAB program
- This stores the instructions so you can edit them, and run them more than once
- Let's do that now ....

.m file – Create

- Click on “New Script” in the top left of your MATLAB window
- In the window that appears, click “Save” (again, top left) and give the file a name ending in .m **(as always, no spaces in the filename!)**
- Notice that this window **doesn't** have the '>>' prompt
- In contrast to code typed on the prompt, the code you type in here won't run immediately

.m files – Edit

- Now you're ready to edit your first MATLAB script
- When a script is run, all its lines of code are run in order
- This is the same as if you'd typed them all in again

.m files – Edit

- Copy some simple code we've already run. **Note:** You don't need the prompt >> characters in the .m file!

```
a = 134;
b = 299;
limit = 40000;
if a*b > limit
    disp('over the limit!')
end
```

- Click on “Save” to save the code

.m files – Run

- To run the .m file, click “Run” (the green triangle icon at the top)
- The output from running the .m file will appear in the prompt window

Running .m files

- Click on 'Run', and the name of the .m file (without .m extension) appears at the prompt
- The .m file can be run by typing this name at the prompt
- E.g. if your file was called limit_exercise.m, you could type:

```
>> limit_exercise
```

- Try it!

Flow of control

- To be clear about what's happening, we can always add code to output some text
- Add a new line just before the 'if', like this:

```
disp('starting')
```

and another new line just after the 'end':

```
disp('finishing')
```

Testing

- Now run the program, and look at the output
- Change one of the inputs so that the condition will be false – for example, you could change the value of 'b' to zero
- Run it again and note how the output changes
- It's always worth testing every 'path' (possibility) in your code like this

if-else

- Optionally, we can use 'else' to give some code to be run if the condition is false.

```
if a*b > limit
    disp('over the limit!')
else
    disp('under the limit!')
end
```

- Either
  - 'a' times 'b' is greater than 'limit', and the first bit of code is run, or
  - it is not, and the second is run.

Another 'if' example

- Open a new code file, and save it under a different name to the first
- Type the following:

```
balance = 120;
if balance > 0
   fprintf('I owe you %d.\n',balance)
else
   fprintf('You owe me %d.\n',balance)
end
```

- Run the code
- Try changing the value of 'balance' and running it again (include some negative values for 'balance')

Exercise 7

- What's wrong with the code, in particular when 'balance' is a negative number?
- What do you think could be done to correct it?
- Are there any values of 'balance' that the code doesn't deal with correctly?

elseif

- What if you want to test more than one condition?

```
if balance > 0
   fprintf('I owe you %d.\n',balance)
else
   fprintf('You owe me %d.\n',balance)
end
```

- This doesn't behave correctly when balance is exactly zero ....

elseif

- Change your code to read:

```
if balance == 0
   fprintf('I owe you nothing\n')
elseif balance > 0
   fprintf('I owe you %d.\n',balance)
else
   fprintf('You owe me %d.\n',-balance)
end
```

elseif

- In general:
- Each of the conditions after the 'if' and subsequent 'elseif's are tested **in order**
- If one of them is true, then the corresponding code is run
- If **none of these** are true, the code after 'else' is run

Combining conditions, and

- We can apply more than one condition at once with **logical operators**
- **&** means '**and**'

```
>> n = 20
>> m = -14
>> n > 10 & m < 0
>> n > 40 & m < 0
>> n > 40 & m == 8
```

- Both sides must be true for the result to be true

Combining conditions, or

- **|** means '**or**'
- This is the 'pipe' character
- This is either on the key next to 'z' or the key next to '1'

```
>> n > 10 | m < 0
>> n > 40 | m < 0
>> n > 40 | m == 8
```

- If either side is true (or both) the result is true

Combining conditions, not

- **~** means '**not**'
- The tilde character is on the key next to Return
- Unlike the others, this only works on one value

```
>> n < 0
>> ~(n < 0)
```

- It inverts the value (true to false, false to true)
- Why doesn't this do what we expect?

```
>> ~n < 0
```

Exercise 8

- In a new code file, write:

```
current_time = 14
```

- This is the hour (24h clock, from 0 to 23)
- Write something which displays:
  - 'Good morning' if the time is between 3 and 12
  - 'Good afternoon' if the time is between 13 and 18
  - 'Good evening' if the time is between 19 and 22
  - 'Good night' if the time is 0, 1, 2, or 23
- Run this with a few different values of current_time

Exercise 8, continued

- At the top of your file, add

  weekday = 4
- This is the day of the week as a number
- 1 is Monday, 7 is Sunday
- Now change your program so that:
  - On Saturday, we're informal and just say 'Hi' all day
  - On Sunday, we don't say 'Good afternoon' until 14

for

- 'if' allows us to choose between different code blocks
- What if we want to run the same code many times?
- We can use 'for' – try this at the prompt:

```
>> for n = 1:10
disp(n)
end
```

for

- So, we know we can run for on a range of integers
- What else can we do?
- Try this again, but using different first lines:

```
for n = 10:20

for n = [-20 49 62 1000]

for n = 0:0.05:1
```

'for' loops

- This is a kind of **loop**
- The code inside the loop – in this case, just

  `disp(n)`

  is run repeatedly, for each value given

'for' loops

- We could have any amount of code in the loop, though, e.g. (no need to type this):

```
for n = 1:10
    m = n + 2
    fprintf('n is: %d, m is: %d\n',n,m)
end
```

A digression about ranges

- This part of the 'for' code specifies a range:

  `1:10`

- This isn't special code just used in 'for'
- It means 'an array with the numbers 1 to 10'

A digression about ranges

- In general, a range looks like this:

  start:stop **OR** start:spacing:stop

- So for example, you could type:

  `>> 0:2:10`

  which gives you an array, or

  `>> my_range = 0:2:10`

  to put this array into a variable

Ranges and indexing

- This connects up with what we saw in indexing

  `D(1,5:10)`

  means 'the first row, columns five to ten' of D

Exercise 9

- Show a nine times table (i.e. the first 12 multiples of 9)
- Use a 'for' loop, the * operator, and the disp() function

MATLAB hates loops

- Most things we can do in a loop, we can just do to a whole array
- How do you think you might multiply the numbers 1 to 10 by 9, if not using a loop?

MATLAB hates loops

- Try this:

```
>> X = 1:10
>> X*9
```

- What happens when you try this:

```
>> 1:10 * 9
```

- Why do you think this gives a different result?
- What can we do to correct it?

MATLAB doesn't really hate loops

but ....

- Avoiding loops can make code more concise
- Your intent is usually clearer, too

Matrix operations

- As this shows, we can do maths on whole matrices
- Let's try this out with a small example:

```
>> B = [-2 -8; 9 4]
>> B + 1
>> B + 10
>> B + 100
```

Matrix operations

- As you can see, this adds the number you give to each of the elements of Y
- This works for other mathematical operations:

```
>> B – 20
>> B .* 5
>> B ./ 10
```

- We use .* and ./ here because * and / are reserved for other uses!

Transpose

- A matrix can be transposed using an apostrophe ' after its name:

  ```
  >> B'
  ```

- Transposing flips the matrix so that the rows become columns (and the columns become rows)
- Note that the ' must be just after the matrix, without any spaces

Transpose

- This is more obvious in non-square matrices:

```
>> G = [1 2 3 4; 5 6 7 8]
>> G'
```

- Even without looking at the actual matrix, you can also see the effect on the size.
- The counts of rows and columns are swapped:

```
>> size(G)
>> size(G')
```

Working with matrices

- We can also perform calculations with two matrices

```
>> A = [1 2; 3 4]
>> C = A + B
```

- The equivalent mathematical notation:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} -2 & -8 \\ 9 & 4 \end{pmatrix} = \begin{pmatrix} -1 & -6 \\ 12 & 8 \end{pmatrix}$$

Maths with two matrices

- Similarly, for subtraction:

```
>> D = A - B
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} - \begin{pmatrix} -2 & -8 \\ 9 & 4 \end{pmatrix} = \begin{pmatrix} 3 & 10 \\ -6 & 0 \end{pmatrix}$$

Matrix multiplication

- **Careful with * ....**
- .* means “multiply individual elements” (referred to as **elementwise** multiplication)

```
>> E = A .* B
```

- * means “matrix multiplication”

```
>> F = A * B
```

Matrix “division”

- Similarly with /
- ./ means “divide individual elements” (**elementwise** division)
- There's no such thing as matrix “division”, but:

```
>> A / B
```

  matrix-multiplies A by the inverse of B

- In MATLAB we would write this:

```
>> A * inv(B)
```

Joining matrices together

- We can concatenate matrices the same way we make them from numbers
- Recall that, for example

```
[1, 2, 3]
```

puts numbers in a row, and

```
[1; 2; 3]
```

puts them in a column

Joining matrices together

- Try these:

```
>> [A, B]
>> [A; B]
```

- As you might expect,

```
>> [A, B]
```

joins (or **concatenates**) horizontally, and

```
>> [A; B]
```

joins vertically

Joining matrices together

- For this to work the matrices must be the right shape
- To join horizontally, they must have the same number of rows
- To join vertically, they must have the same number of columns
- Thinking of matrices as tables, the side on which they are joined must be the same size

Joining matrices together

- Joining horizontally:

24	-83	54
6	45	-10

-3	42
16	0

→

24	-83	54	-3	42
6	45	-10	16	0

but joining these vertically does not work:

24	-83	54
6	45	-10

-3	42
16	0

Exercise 10

- Here are four matrices in MATLAB notation
- Find all the combinations in which they can be joined together vertically or horizontally

```
>> W = [1 2 3; 4 5 6; 7 8 9; 0 0 0]
>> X = [24 -83 54; 6 45 -10]
>> Y = [-3 42; 16 0]
>> Z = [NaN; -Inf]
```

Matrix functions: diag

- Taking an example 3 x 3 matrix:

```
>> D = [-1 43 37; -10 52 32; 30 -44 -67]
```

- We can take the **diagonal** of D using diag():

```
>> diag(D)
```

- This gives the numbers from the top-left of D in a diagonal line
- To put it another way, it's the same as:

```
>> [D(1,1) D(2,2) D(3,3)]
```

Matrix functions: triu, tril

- The triu and tril functions return the upper and lower triangular parts of a matrix

```
>> triu(D)
>> tril(D)
```

- These give the diagonal plus everything above it (upper) or below it (lower)

Matrix functions, sum

- The sum() function gives the sum along rows or columns
- Sum of columns:

```
>> sum(D,1)
```

- Sum of rows:

```
>> sum(D,2)
```

Vector dot product

- The dot() function gives the vector dot product

```
>> U = [-1 43 37]
>> V = [-10 52 32]
>> dot(U,V)
```

Vector dot product

- We can check this using some of what we've already learned:

```
>> products = U.*V
>> sum(products, 2)
```

- Also note that this is the same as:

```
>> U*V'
```

repmat

- The repmat() function replicates a matrix to create a larger one
- It's used like this:

```
repmat(M,[row col])
```

where:

  - M is the matrix to be replicated
  - *row* is the number of “rows of M”
  - *col* is the number of “columns of M”

Example

- Try this:

```
>> M = [1 4; 9 16]
>> repmat(M, [2 3])
```

- You'll see that M is repeated:

  - twice vertically (two “rows of M”)
  - three times horizontally (three “columns of M”)

Exercise 11

- Define the following vectors:

```
>> X = [1; 2; 3]
>> Y = [-4 -5 -6]
```

- Using repmat() on X, make a 3 by 3 matrix
- Using repmat() on Y, make a 5 by 6 matrix

(**Remember:** if you want can use the size() function to check the size of a matrix, rather than counting rows and columns on the screen)

Creating matrices

- We already know how to create a matrix with specific, different values in it, e.g.:

```
>> D = [-1 43 37; -10 52 32; 30 -44 -67]
```

- There are some other special functions that allow us to create matrices

ones

- The “ones” function creates a matrix consisting only of ones
- Following the usual convention, it takes a number of rows, and a number of columns
- e.g. to create a 4 x 3 matrix:

```
>> ones(4,3)
```

zeros

- Similar to ones() this creates a matrix consisting only of zeros
- e.g. to create a 2 x 8 matrix of zeros:

```
>> zeros(2,8)
```

Identity matrix, eye()

- The “eye” function creates an identity matrix
- Identity matrixes are always square, so it only takes one number (which is both the number of rows and of columns)
- e.g. to create a 5 by 5 identity matrix:

```
>> eye(5)
```

Exercise 12

- Using the ones() function, and what we've already learned about arithmetic with matrices, how would you quickly create:
  - a 6 by 3 matrix
  - where every element of the matrix is 12?

Plotting

- We can plot a line simply with the “plot” function
- This takes an array of X values and an array of Y values, e.g.

```
>> X = [1 2 4 5 8];
>> Y = [-6 8 7 3 -1];
>> figure
>> plot(X, Y)
```

Changing plot colours and style

- We can specify the colour and style of the line (or points)
- e.g.

```
>> plot(X, Y, 'r')
>> plot(X, Y, 'gx')
>> plot(X, Y, 'b--')
```

- There are **lots** of possibilities
- See the documentation for “plot” for more details

Adding some annotations

- We can add a title, and labels for the x and y axes

```
>> title('My example graph')
>> xlabel('time (days)')
>> ylabel('temperature (deg C)')
```

Changing the x and y range

- We can use the axis() function to give the limits for the plot
- These are given in a list:

  x start, x end, y start, y end

- e.g.

```
axis([0 10 -9 9])
```

Plotting multiple lines

- You can plot multiple lines on the same chart
- Try this:

```
>> Y2 = [2 4 -7 6 4];
>> plot(X,Y,'r--',X,Y2,'bv')
```

Functions

- We can write our own functions, which we can then use just like built-in functions:
- our own functions will take a number of parameters (or none!)
- and can also **return** (i.e. give back) a value

Functions

- Functions allow you to store a set of instructions and run them on different values, for example:

```
function [result] = add_two(n)
    result = n+2;
end
```

Function in a .m file

- Functions must be defined in .m files
- The file has the same name as the function
- Open a new .m file, this time by clicking New -> Function
- This opens a template for a function

Function in a .m file

- Edit the template to look like this:

```
function [result] = add_two(n)
    result = n+2;
end
```

- Save the file
- You'll find that MATLAB suggests the correct filename, which is the function name (and .m)

Running your function

- Now run your function:

```
>> add_two(3)
>> add_two(-42)
>> add_two(0)
```

Function naming

- The rules for creating names are the same as for variables
- **Reminder:**
- Names must start with a letter, and can contain letters, numbers and underscores
- Names are case sensitive
- They can't contain spaces

How a function is defined

- A function definition:

Name of return value

Name of function

```
function [result] = add_two(n)
    result = n+2;
end
```

Function body

How a function is called

- **Reminder:**
- A function is called with its name, followed by the parameters in brackets:

```
>> add_two(8)
```

What happens when a function is called

- The instructions in the function are followed from the top to the end
- Values are then given back (**returned**) to the point where it was called
- The values returned are the ones specified at the top of the function (in our example above this is “result”)

Function with no return value

- A function doesn't have to return anything
- If it doesn't, then the first line changes
- For example, a function like this:

```
function [result] = my_function(n)
```

without a return value would be:

```
function my_function(n)
```

Another example function

- Open another .m file with New -> Function
- Edit the template to look like this:

```
function [string_out] = greeting(to_greet)
    string_out = ['Hello ', to_greet];
end
```

- Save this (as greeting.m)

An aside on joining strings

- MATLAB treats strings like arrays
- The same way we can do this:

```
>> V = [1 2 3 4];
>> W = [5 6 7 8];
>> [V W]
```

we can also do this:

```
>> g = 'Hello '
>> n = 'Edinburgh'
>> [g n]
```

Another example function

- Now run the function

```
>> x = greeting('you');
```

- What is in x now? Have a look ....
- Try this out with a few values

```
>> disp(greeting('Edinburgh'));
```

- Quick exercise: can you give the function a value that causes an error?

Exercise 13

- Open another .m file, to write another function
- This will take a string as a parameter, like the “greeting” function
- We want our new function to do this:

```
>> greet_and_count('Anna')
Hello Anna
Your name has 4 letters
>> greet_and_count('Andrew')
Hello Andrew
Your name has 6 letters
```

Exercise 13, clues

- You can call the greeting() function from your new function
- You'll need fprintf()
- You'll also need the length() function which can measure the length of a string

Exercise 13, questions

- Are there any input values that cause an error?
- Are there any input values that cause “wrong” (or wrong-looking) output?

Exercise 14

- Change your function to do something sensible:
  - when the input is only one character long
  - when the input is empty
- These inputs would look like this:

```
>> greet_and_count('B')
>> greet_and_count('')
```

- You'll need to use if .... elseif .... end

Exercise 15

- Start a new function called bars()
- It should take a row vector as input
- For each number in the vector, in order, it should show that number of '*' on a line
- For example:

```
 >> bars([1 6 2 4])
*
******
**
****
```

Exercise 15, clues

- You'll need to use a for ... end loop to work through the input vector
- You can use repmat to copy strings, e.g.:
  `>> repmat('moo ', [1 20])`

  (if this seems confusing try it for a few different values)

Function example

- Download example_function.m from the usual place (short link: http://edin.ac/1y1Pd7K)

```
function [result] = example_function(M)
    tl = M(1,1);
    tr = M(1,end);
    bl = M(end,1);
    br = M(end,end);
    if tl == tr & tl == bl & tl == br
        result = 1;
    else
        result = 0;
    end
end
```

Indexing note

- As an index, “end” always means the last thing
- So, M(3,end) means the element in the third row, and last column of M
- This can be used in ranges e.g.

```
>> M(4:6, 2:end)
```

Exercise 16

- This is more of a thought exercise!
- It's important to come up with good further questions to ask, as well as answers ....
- What do you think this function does?
(feel free to try it out)
- Can you think of a good name for it?

The “return” keyword

- This stops a function from running any further
- No more instructions in the function are carried out
- The values specified at the top of the function are returned

Early return

- Let's say we want our function to complain if it's given a matrix with less than 2 rows, or less than 2 columns
- In this case we'll return NaN (this is a common way to say “didn't work” in MATLAB)
- Let's break this problem down a little

Exercise 17

- Open a code file
- Write this:

```
function [result] = less_than_2_by_2(M)

end
```

- Add code between these lines, so that:
  - if M has less than 2 rows, or less than 2 columns, the function returns 1
  - otherwise, it returns 0

Exercise 18

- Go back to the function in the last exercise
- At the start of the function, add some code to return early (returning NaN) if the input matrix is smaller than 2 by 2
- Use “if”, and the less_than_2_by_2 function

Even or odd?

- Save this code as even.m

```
function [is_odd] = odd(n)
    return mod(n, 2)
end
```

- This returns 0 if a number is even and 1 if it is odd
- Try it out

(**Note:** the mod() function calculates remainders – see documentation for details)

Exercise 19

- Open a new .m file
- Write a function called odd_count() that
  - takes a row vector as its only parameter
  - returns the number of odd numbers in the vector
- Try it on some examples

```
>> odd_count([1 2 3 4])
>> odd_count([-12 -9 -7 -5 1 0 6])
```