

Introduction to MATLAB

Overview

- ▶ The purpose of this resource is to give a very basic introduction to MATLAB
- ▶ If you have no prior experience with MATLAB or computer programming, it's probably best to work through the slides in order
- ▶ Open MATLAB on your computer and wherever an example is given, try it for yourself in MATLAB
- ▶ If you have some prior MATLAB experience you may find it more useful to skip to specific sections of interest
- ▶ There are some [exercises](#) with [solutions](#) which accompany this course, to help you solidify your understanding

Course outline

Overview

Before you get started

The MATLAB desktop

- Sub-windows

- Search path

- The command window

- MATLAB documentation

Variables

- Creating variables

- Reassigning variables

- Variable names

Arithmetic operators

Importing data

Useful shortcuts and commands

Functions

Data types

Some number types

Numbers in MATLAB

Logical values

Scripts

Debugging scripts

Arrays

Accessing individual array elements

Accessing multiple array elements

Some useful array functions

Memory allocation

Array operations

Matrix operators

Plotting arrays

Sparse arrays

Finding help

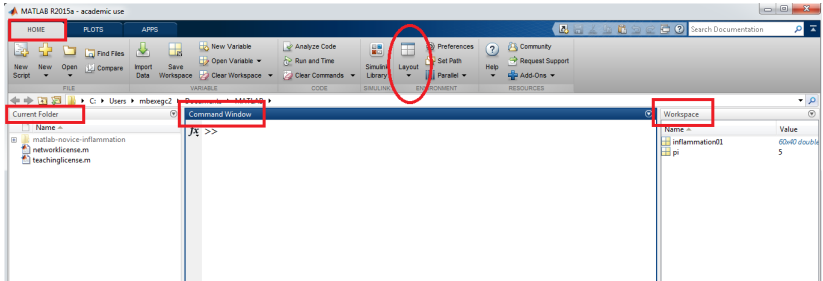
Before you get started

Licence restrictions, installation instructions, considerate behaviour

- ▶ Check the [IT Services website](#) for licence restrictions, and other useful information organised by tabs...
- ▶ ...for example, [installation instructions](#) (if you use a managed desktop, see [requesting a new application](#) instead)
- ▶ Ensure connection to University network licence i.e. connect to campus local-area network or [VPN](#) (staff only)
- ▶ The licence supports 616 concurrent research users (and generally fewer licences for MATLAB's toolboxes), so please:
 - ▶ Don't leave MATLAB running unnecessarily
 - ▶ Don't open MATLAB on multiple hardware

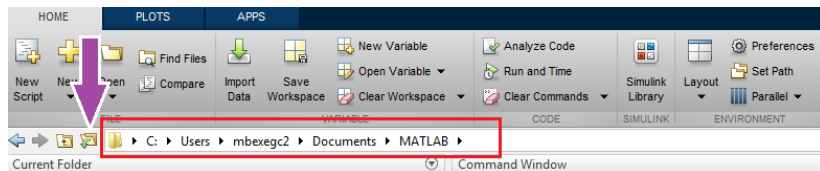
MATLAB desktop windows

- ▶ MATLAB has several movable sub-windows: current folder, command window (for entering commands), workspace (shows current variables).
- ▶ To get back to the default layout:
Home tab → Environment section → Layout button → Default



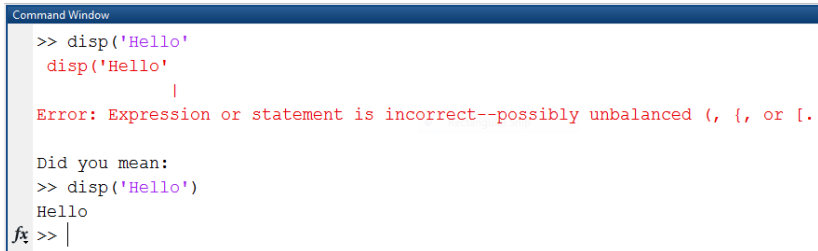
Setting the search path

- ▶ MATLAB uses an internal variable called the path, which it uses when searching for files. It already knows where the MATLAB libraries are, but in order to find *your* files (data files, MATLAB scripts that you have written etc), you will need to set the path.
- ▶ The current folder is shown in the red box below, and you can change the current folder by clicking on the button shown by the purple arrow. Files in this folder are now visible to MATLAB.



The command window

- ▶ This allows you to use MATLAB interactively, by typing commands at the prompt (`>>`)
- ▶ After typing commands (alternatively, cut and paste), press **enter/return** key for MATLAB to run your command(s)
- ▶ This requires you to follow the rules of the MATLAB programming language
- ▶ If you enter an invalid command (e.g. typos) MATLAB will return an error
- ▶ The error may contain important information to help fix the problem (see example below)

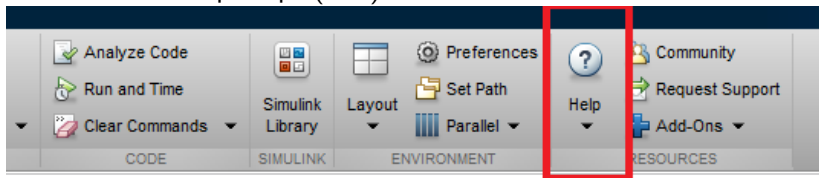


```
Command Window
>> disp('Hello'
disp('Hello'
|
Error: Expression or statement is incorrect--possibly unbalanced (, {, or [.

Did you mean:
>> disp('Hello')
Hello
fx >> |
```

MATLAB help and documentation

- ▶ MATLAB has very good help pages, which include example code
- ▶ To start help, click on the **help** button on the home tab or enter doc at the prompt (`>>`)



- ▶ For help with specific functions (e.g. plot), enter `doc plot` at the prompt to see the help pages or `help plot` to get information at the prompt
- ▶ Copy and paste MATLAB example code at the prompt to run

Examples

[collapse all](#)

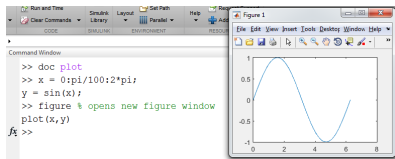
▼ Create Line Plot

Define x as a vector of linearly spaced values between 0 and 2π . Use an increment of $\pi/100$ between the values. Define y as sine values of x .

```
x = 0:pi/100:2*pi;  
y = sin(x);
```

Create a line plot of the data.

```
figure % opens new figure window  
plot(x,y)
```



Creating variables

- ▶ A variable allows us to assign data to a variable name
- ▶ To assign data to a variable, use the equals sign, =
- ▶ MATLAB first evaluates whatever is to the right of = , and then assigns this value to the variable
- ▶ To assign the result of $10 + 1$ to a variable var1

Command Window

```
>> var1 = 10 + 1  
var1 =  
    11
```

- ▶ To assign the result of $\text{var1} - 3$ to a new variable var2

```
>> var2 = var1 - 3  
var2 =  
     8
```

Reassigning (overwriting) variables

- ▶ To reassign our variable `var2` to be the cosine of zero

```
>> var2=cos(0)
var2 =
      1
```

- ▶ To add one to our variable `var2` (remembering that MATLAB first evaluates what is to the right of the `=`, then assigns this value to the variable `var2` on the left)

```
>> var2 = var2 + 1
var2 =
      2
```

Variable names

- ▶ These can contain letters, underscore and numbers
- ▶ They are case sensitive
- ▶ They must start with a letter
- ▶ `namelengthmax` returns maximum length (63 characters in MATLAB R2015a)
- ▶ Advice
 - ▶ Choose descriptive names that aid understanding how the code works
 - ▶ Dont be afraid of long names if it makes code more readable
e.g. `number_of_samples`, `BeamIntensity`

A warning about variable names

- ▶ MATLAB will let you create variables with the same name as built in MATLAB variables and functions
- ▶ This hides the default MATLAB behaviour and therefore should be avoided
- ▶ MATLAB will not warn you if you do this

Command Window

```
>> pi  
ans =  
    3.1416  
  
>> pi=5  
pi =  
    5
```

Working with variables

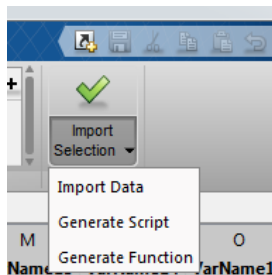
- ▶ To list all variables in current workspace enter `who` or `whos`
- ▶ Variables are shown in the Workspace window - double click on them to view
- ▶ `clear` removes all variables, `clear variable1` removes `variable1` only
- ▶ Use `save` and `load` to save variables to/ load from files (by default MATLAB saves binary data files, which guarantees no loss of accuracy and is quicker than human-readable text)
- ▶ See MATLAB Help pages for more information (`>> doc save; doc load`)

Arithmetic operators

- ▶ MATLAB has the usual arithmetic operators
 - ▶ + addition
 - ▶ - subtraction
 - ▶ * multiplication
 - ▶ / division
 - ▶ ^exponentiation (e.g. $2.3^5 = 2.3^5$)
- ▶ Caution: for arrays * / and ^ perform matrix operations
- ▶ For scalars, the above operators behave as expected
- ▶ See slide 62 for more information

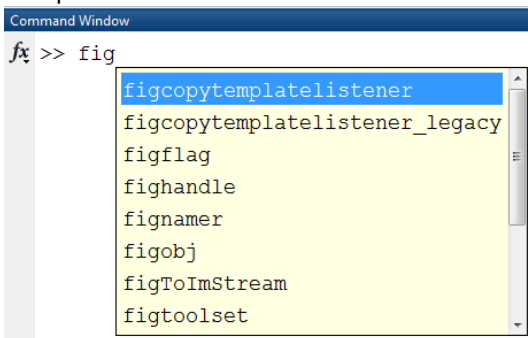
Importing data

- ▶ Often you will want to import data from a file e.g. comma separated values, Excel spreadsheets etc
- ▶ Note: some options for importing Excel spreadsheets require Excel to be installed read the MATLAB documentation
- ▶ A good starting point is the Import Data button (on the Home tab) this can generate scripts to automate the importing of data
- ▶ Browse to a data file, then click on Import Selection
- ▶ You can use these scripts as templates to import other data files



Useful shortcuts and commands

- ▶ Up arrow key scrolls through previous commands (useful for fixing typos)
- ▶ Can type the first few characters at the prompt, then use up arrow to find commands which match the characters typed
- ▶ Tab auto-complete: type the first few letters of a command (e.g. function or variable) then press tab key to show completions



The screenshot shows a 'Command Window' with a dark blue title bar. The prompt is 'fx >>'. The user has typed 'fig', and a list of completions is displayed in a yellow box. The first completion, 'figcopytemplatelistener', is highlighted in blue. The list of completions includes:

```
figcopytemplatelistener
figcopytemplatelistener_legacy
figflag
fighandle
fignamer
figobj
figToImStream
figtoolset
```

Working with folders

- ▶ `pwd` prints current directory
- ▶ `ls` lists files in current directory (Note: the parent directory is represented by `..`)
- ▶ `cd` changes directory
e.g. `cd ..` changes to the parent directory
- ▶ `mkdir` creates a new folder
e.g. `mkdir newFolder`
- ▶ Search MATLAB Help for “Manage Files and Folders”
- ▶ Alternatively use the [Current Folder](#) Window

Functions

- ▶ Most MATLAB functionality is accessed via functions
- ▶ To call (i.e. run) a function enter the function name
- ▶ Any data required by the function must be passed to the function as input arguments
- ▶ When the function finishes, values are passed back as output arguments
- ▶ Input and output arguments are sometimes optional
- ▶ In general MATLAB has very good documentation - read this to understand how to use MATLABs functions (hint: `>> function`)

Using functions

Brief overview - read the documentation for more detail

- ▶ Input arguments in parentheses () after the function name
- ▶ Output arguments in square brackets [] before the equals sign (brackets can be omitted for single output argument)
- ▶ Separate multiple arguments with commas
e.g. `[out1, out2, out3] = func(in1, in2)`
- ▶ Some arguments may be optional (this will be described in MATLAB Help pages)
- ▶ MATLAB functions provide useful error messages - e.g.

```
>> sin
```

```
Error using sin
```

```
Not enough input arguments.
```

MATLAB notation

The semicolon

- ▶ By default MATLAB displays output after a command is run (e.g. Assigning variables, Calculations, Function calls)
- ▶ The semicolon suppresses this output, e.g.

```
>> var1 = 10;  
>> sin(10);  
>> |
```

- ▶ This does not suppress text printed to the screen from the command or function (e.g. via `disp` or `fprintf` - see MATLAB Help on `disp` and `fprintf`)

Some functions to start with

- ▶ There are many built in MATLAB functions and operators
- ▶ Run the following commands to read about some key functions:
 - ▶ `help ops` (lists operators)
 - ▶ `doc operator precedence`
 - ▶ `help elfun` (lists elementary maths functions)
 - ▶ `help specfun` (lists specialised maths functions)

Toolboxes

- ▶ Some MATLAB functions come in 'add on' toolboxes
 - ▶ Additional cost
 - ▶ Possibly (very) limited number of licences
 - ▶ Enter ver to see list of installed toolboxes
- ▶ Just because a toolbox is installed doesn't mean there's a licence for it
- ▶ Each installed toolbox will have a top level entry in MATLAB Help
- ▶ Third party toolboxes can be added
 - ▶ Some are free
 - ▶ Advice: investigate quality before using (e.g. see <http://www.walkingrandomly.com/?p=2323>)

Some number types

- ▶ Real numbers

A rigorous definition is difficult, but basically not imaginary or complex. They include whole numbers and fractions.

e.g. 1, 2, 0, -1, $3/5$, π , 34.23515, -2341, -234243.243242

- ▶ Integers

Whole numbers (i.e. a subset of real numbers)

e.g. ...-6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6...

- ▶ Imaginary numbers

A number which when multiplied by itself gives a negative result

e.g. $\sqrt{-1}$ is an imaginary number as $\sqrt{-1} \times \sqrt{-1} = -1$

- ▶ Complex numbers

A number obtained by adding a real and imaginary number

e.g. $\sqrt{214.123} + 153$

Numbers in MATLAB

Double precision floating point

- ▶ By default numbers are double precision floating point
 - ▶ Double precision: 64 bits (binary digits) are used to store each value i.e. each bit represents either the value 1 or 0
 - ▶ Accurate to around 16 significant decimal digits
 - ▶ Most numbers represented by floating point are **approximations** to real numbers
 - ▶ Conform to IEEE standard
 - ▶ Includes Inf (infinity) and NaN (not a number)
- ▶ So the following assigns a double precision floating point value to the variable var1 `>> var1 = 10`
- ▶ Floating point values are used to represent real numbers and double precision floating point numbers are very likely what you will use (most) in MATLAB

Floating point notation

- ▶ e means times ten raised to the power of
e.g. 1e2 means 1×10^2 i.e. 100
e.g. 1e-2 means 1×10^{-2} , i.e. $1/10^2$ or 0.01
- ▶ This allows us to represent very large and very small numbers

e.g.

```
>> realmax  
ans =  
    1.7977e+308  
  
>> realmin  
ans =  
    2.2251e-308
```

Output precision

- ▶ By default MATLAB displays numbers using up to 5 digits

```
>> pi  
ans =  
    3.1416
```

- ▶ However, π is a double precision value and has more precision than 5 digits
- ▶ We can change the format to show additional accuracy

```
>> format long  
>> pi  
ans =  
    3.141592653589793
```

- ▶ Look at the documentation for more information

Floating point versus real numbers

- ▶ Floating point numbers are not real numbers
Values are stored as [binary numbers](#) (a series of ones and zeros). General behaviour is close enough to reals for most purposes
- ▶ However it is very important to understand there are differences between real and floating point numbers. This is a topic for further reading e.g. The MATLAB Help page on “Floating-Point Numbers”, and [What Every Computer Scientist Should Know About Floating-Point Arithmetic](#)
- ▶ For example: sine of π is not zero in MATLAB

```
>> sin(pi)
ans =
    1.2246e-16
```

Floating point rounding

- ▶ In general not all real numbers can be represented exactly
- ▶ For example `eps` returns the distance between 1.0 and the next largest double-precision number. Real numbers between 1.0 and $1.0 + \text{eps}$ can't be represented using MATLAB doubles. Values between 1.0 and $1.0 + \text{eps}$ will be rounded.
- ▶ This introduces rounding errors

Single precision floating point values

- ▶ MATLAB also has single precision (32-bit) floating point numbers
 - ▶ They require half as much memory to store
 - ▶ They have less precision
 - ▶ Around 8 significant decimal digits
- ▶ They must be created explicitly e.g. using the `single` function
- ▶ You probably won't use these, but it may be important to know they exist

```
>> s1 = single(10)
```

```
s1 =  
    10
```

Complex numbers

- ▶ in MATLAB `i` represents the square root of -1

```
>> i*i  
ans =  
    -1
```

- ▶ Complex arithmetic is fully supported e.g.

```
>> sqrt(-1)  
ans =  
    0.0000 + 1.0000i
```

- ▶ Enter `help complex` for more information
- ▶ Warning: do not use `i` as a variable name as this will hide MATLAB default behaviour (see slide [13](#))

Integers

- ▶ MATLAB has various signed and unsigned integers
 - ▶ Signed integers can take positive, zero and negative integer values
 - ▶ Unsigned integers can only take positive and zero integer values
 - ▶ There are 8, 16, 32 and 64 bit integers of each type (signed and unsigned)
- ▶ Integers must be created explicitly
 - ▶ For signed integers, use `int8`, `int16`, `int32`, `int64`
 - ▶ For unsigned, use `uint8`, `uint16`, `uint32`, `uint64`
 - ▶ e.g. to create an unsigned 8 bit integer
`>> var3 = uint8(24);`
- ▶ Integers with more bits require more memory
- ▶ More bits gives a wider range of possible values

Maximum and minimum numerical values

- ▶ There are limits to the largest and smallest floating point values which can be represented
- ▶ There are limits to the most negative and most positive value for each integer type
- ▶ There are functions to give the value of these limits:
`intmin`, `intmax`, `realmin`, `realmax`
- ▶ These limits can introduce errors to your computation e.g.
 - ▶ Floating point results can differ significantly from equivalent real number calculations if very small values are involved
 - ▶ Floating point calculations may return `Inf` or `-Inf`
 - ▶ Integer results will be rounded to the maximum and minimum values for that type if out of range

Logical values

- ▶ These represent the logical values true and false
- ▶ Values can be either 1 (true) or 0 (false)
- ▶ Assign using true or false

```
>> a=true  
a =  
    1  
  
>> b = false  
b =  
    0
```

- ▶ Logical values are more often used when writing your own functions, but are also returned from some intrinsic MATLAB functions, so are useful to know about

More on data types

- ▶ To find the type of the data use the `class` function - this returns the class of any MATLAB object e.g.

```
>> class(s1)  
ans =  
single
```

- ▶ Reminder: we can explicitly convert data types e.g.

```
>> d1 = 10; % d1 is double precision  
>> i1 = int8(d1); % i1 is of class int8
```
- ▶ MATLAB automatically assigns the class for results involving multiple class types (e.g. multiplying a double and an integer)
- ▶ When undertaking calculations using multiple classes make sure you understand how MATLAB behaves (read the documentation or run tests)

Strings

- ▶ Strings (i.e. text) are arrays of characters (We'll come to arrays later)
- ▶ They are of class char
- ▶ They are created using single quotes e.g.

```
>> my_string='Hello'  
my_string =  
Hello  
>> class(my_string)  
ans =  
char
```

- ▶ Some MATLAB functions take strings as input arguments

Function handles

- ▶ These allow a function to be represented by a variable i.e. the variable can be used to execute the function
- ▶ The variable has the class `function_handle`
- ▶ These can be created using the `@` symbol e.g. create a function handle representing the `sin` function

```
>> mysine=@sin;  
>> mysine(pi/2)  
ans =  
      1
```

- ▶ We can use these to pass functions to other functions - e.g. try `ezplot(@sin)` or `ezplot(mysine)`

Cells

Also known as Cell Arrays

- ▶ These can contain multiple values of different data type
- ▶ Values are accessed using indices
- ▶ You may not need these, but they're useful to know about (search for *Cell Arrays* in MATLAB documentation)
- ▶ They can be created using curly braces { }
- ▶ These have the `cell` class

Structures

Also known as Structure Arrays

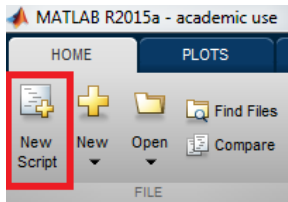
- ▶ These allow you to group related data of different types e.g.

```
>> data.name='sample1';data.x=1234;data.y=1234.4321;  
>> class(data)  
ans =  
struct  
>> data  
data =  
    name: 'sample1'  
      x: 1234  
      y: 1.2344e+03
```

- ▶ It may be useful to know about these - see MATLAB Help for further reading

Scripts

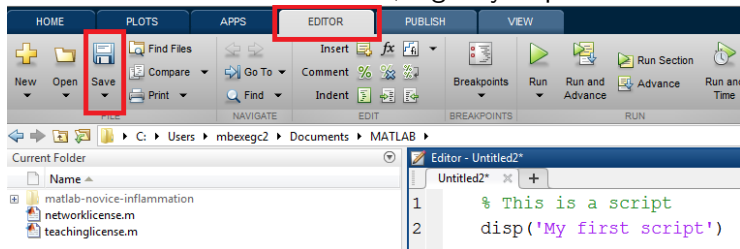
- ▶ Are text files containing a list of commands which are executed sequentially (as if they had been entered at the prompt)
- ▶ Enter edit at the prompt to start the Editor
- ▶ Or click on Home → New Script



- ▶ We can use the editor to write scripts (and also functions)
- ▶ Advantages of scripts over typing commands at the prompt:
 - ▶ Easier for complex calculations
 - ▶ Scripts can be edited and reused
 - ▶ Calculations can be repeated

Script files

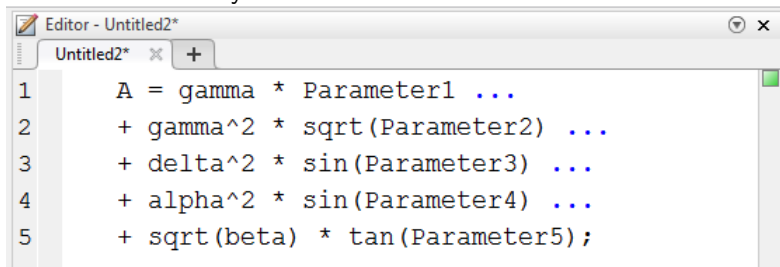
- ▶ Commands can be typed in a script and run as a program: type the list of commands into the editor instead of into the command window
- ▶ Typically each command is on a separate line
- ▶ Save to a file with .m extension, e.g. myscript.m



- ▶ Ensure script is in a directory on the MATLAB path (see slide 6 and `help path` for more info on the MATLAB path)
- ▶ Run the script by entering the name of the file (without the .m file extension) e.g. myscript

Continuation of long lines

- Use ... (three dots) to continue a command onto the next line so that it fits nicely into the editor



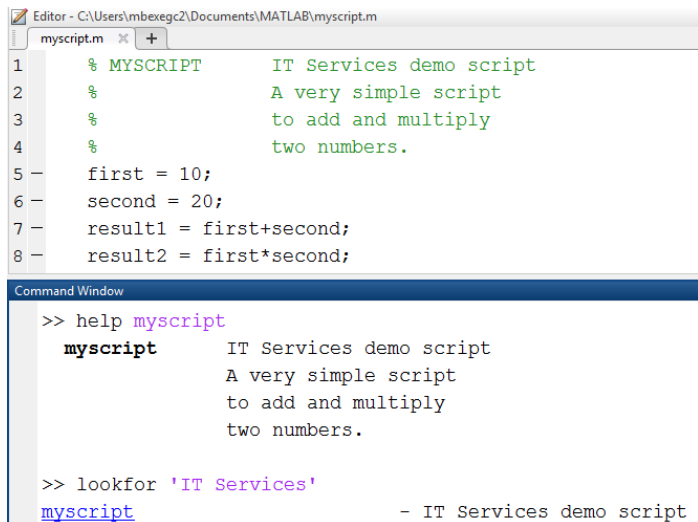
```
1      A = gamma * Parameter1 ...
2      + gamma^2 * sqrt(Parameter2) ...
3      + delta^2 * sin(Parameter3) ...
4      + alpha^2 * sin(Parameter4) ...
5      + sqrt(beta) * tan(Parameter5);
```

Comments

- ▶ Any text following % is a comment
 - ▶ Comments are ignored when the script is executed
 - ▶ Comments can be used to explain how your code works
 - ▶ Advice: add verbose comments so you understand everything your script does when you come to look at it later
- ▶ The first comment line (referred to as the H1 line) is used by lookfor (lookfor searches for strings in the H1 line of .m files - see `help lookfor`)
- ▶ The text in the first block of comments is returned by `help` (see `doc help` or `help help` for more info)
- ▶ It is good practice to write scripts that work with MATLABs `help`

Example script

Used with `help` and `lookfor`



The image shows a MATLAB environment with two windows. The top window is the Editor, showing a script named `myscript.m` with the following content:

```
1 % MYSCRIPT      IT Services demo script
2 %              A very simple script
3 %              to add and multiply
4 %              two numbers.
5 - first = 10;
6 - second = 20;
7 - result1 = first+second;
8 - result2 = first*second;
```

The bottom window is the Command Window, showing the output of the `help` and `lookfor` commands:

```
>> help myscript
myscript      IT Services demo script
                A very simple script
                to add and multiply
                two numbers.

>> lookfor 'IT Services'
myscript      - IT Services demo script
```

Using scripts

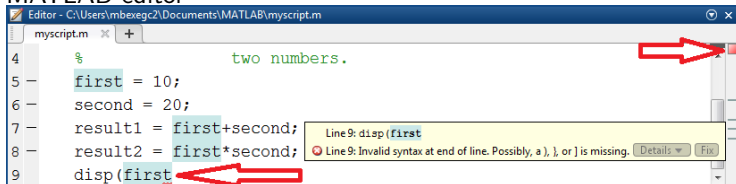
- ▶ Scripts can call other scripts - return can be used to pass control back to the calling script or the keyboard
- ▶ Scope of variables
 - ▶ A script has full access to all variables in the MATLAB workspace and in any calling script
 - ▶ Variables created in a script persist after the script completes
 - ▶ Therefore if using multiple scripts care must be taken when choosing variable names
 - ▶ **For this reason, use of scripts is best reserved for simple tasks**
 - ▶ Write your own functions for more complex programming

Good programming practice

- ▶ Document your scripts and functions, so that you (or anyone else using your code) can understand how everything works
- ▶ When writing comments, assume you'll come back to the code in many years' time and won't remember anything about how it works
- ▶ Use variable names that describe what they are for
- ▶ Divide longer programs into logical units (e.g. using functions)
Functions not covered in this course (look for examples in the MATLAB documentation, and consider attending the [Programming in MATLAB](#) course)

Tools for checking code

- ▶ MATLAB has tools to report potential errors and improvements to your scripts e.g.
 - ▶ Syntax errors e.g. missing brackets (these can be hard to spot in long m-files)
 - ▶ Possible efficiency improvements
- ▶ There are various interfaces to these tools
 - ▶ MATLAB editor



- ▶ Underlines suspect code and gives helpful messages
- ▶ Provides coloured box to top right of window (red means potential problems)
- ▶ *Home* tab (code section) → *Analyze Code*
- ▶ Check documentation for additional methods

Debugging

- ▶ MATLAB has very good errors and warnings
 - ▶ These can help identify bugs
 - ▶ We can also add error and warning messages to our code
- ▶ We can also debug by outputting variable names
e.g. use `fprintf`, `disp` or remove semicolons (see examples on next slide)
- ▶ MATLAB also has a very nice debugger

Printing text to screen or files

- ▶ disp displays single values or strings

```
>> a=1.234;
```

```
>> disp(a)
```

```
1.2340
```

```
>> disp('Hello')
```

```
Hello
```

- ▶ fprintf can print formatted text to screen (see doc fprintf)

```
>> fprintf('The value a is %0.5g. Pi=%0.7e\n',a, pi)
```

```
The value a is 1.234. Pi=3.1415927e+00
```

- ▶ fprintf can also print formatted text to files

```
>> mydata=[12,3,44,35,99];
```

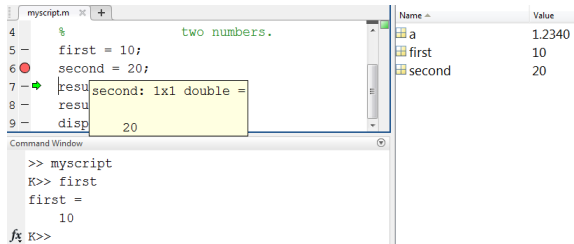
```
>> myfileID = fopen('myDataFile', 'w');
```

```
>> fprintf(myfileID,'My formatted data: %4.1f\n', mydata);
```

```
>> fclose(myfileID);
```

The debugger

- ▶ You can run the debugger from the MATLAB Editor (or from the prompt e.g. use `dbstop` function)
- ▶ Using the debugger we can
 - ▶ Set stop points (including conditional stop points)
 - ▶ Execute a program line by line, either stepping into or over functions
 - ▶ View variables (three methods, all shown in screenshot below)
 - ▶ Hold the mouse cursor over a variable in the Editor
 - ▶ Use the prompt (note the different prompt `K>>`)
 - ▶ Use the Workspace window



Arrays

- ▶ An array is a systematic collection of data items
 - ▶ In general arrays can have 1 or more dimensions
 - ▶ Data in an array can be selected by indices (e.g. one index per dimension)
- ▶ MATLAB arrays
 - ▶ 2 or more dimensions
 - ▶ Indexing starts at 1
 - ▶ Arrays with zero size are allowed
 - ▶ Arrays are dense by default (all values are stored in memory, even if they are zero)
 - ▶ Sparse arrays are also available (only non-zero values are stored)
- ▶ Note: arrays are different from the `cell` and `struct` classes
- ▶ Example of a 4×6 array:

```
>> A=rand(4,6)
```

```
A =
```

0.8147	0.6324	0.9575	0.9572	0.4218	0.6557
0.9058	0.0975	0.9649	0.4854	0.9157	0.0357
0.1270	0.2785	0.1576	0.8003	0.7922	0.8491
0.9134	0.5469	0.9706	0.1419	0.9595	0.9340

Types of array

- ▶ ALL MATLAB data is held in arrays
 - ▶ Scalars: a special case of 2D array (size = 1×1)
 - ▶ Vectors: special case of 2D array (size $1 \times n$ or $n \times 1$, where $n > 1$)
 - ▶ Matrices: 2D arrays (size $m \times n$, where $m > 1$ and $n > 1$)
- ▶ Advised further reading for the mathematically inclined:
Scalar, vector and matrix arithmetic

Creating arrays

- ▶ Use square brackets with semicolon to separate rows
e.g. `>> A=[1 2; 3 4]`
- ▶ Space (or comma) separates values in the same row
- ▶ Can create arrays from scalars or arrays (since scalars are considered 1×1 arrays)
e.g. `>> [A,A;A,A]`
- ▶ Alternatively use MATLAB array generation functions, e.g. `zeros`, `ones`, `rand`, `randn`, `eye`
e.g. `>> B=[A; rand(2,2); ones(2,2)];`
- ▶ Zero length arrays are allowed, e.g. `a=[]`
- ▶ Try the examples given above

Accessing individual array elements

One index per dimension

- ▶ Use parentheses and indices
e.g. `Y(3,4)` for element at row 3, column 4 in 2D array `Y`
- ▶ Can access array elements on left or right of assignment operator

```
>> Y=magic(4)
```

```
Y =
```

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

```
>> val1=Y(2,3)
```

```
val1 =
```

```
10
```

```
>> Y(1,4)=-17
```

```
Y =
```

16	2	3	-17
5	11	10	8
9	7	6	12
4	14	15	1

Accessing individual array elements

Single index notation

- ▶ MATLAB stores elements contiguously in memory using column-wise ordering, i.e. array values are stored one after the other, a column at a time
- ▶ A single index can be used to access elements based on the column-wise ordering e.g.

Y =

16	2	3	-17
5	11	10	8
9	7	6	12
4	14	15	1

>> Y(9)

ans =

3

- ▶ Enter `help paren` for more information

Accessing multiple array elements

- ▶ Arrays can be used to index arrays e.g.
X([4,2,5]) accesses elements with indices 4, 2 and 5
(Reminder: create arrays using square brackets [])

```
>> X=(5:10) % Create an array
```

```
X =
```

```
     5     6     7     8     9    10
```

```
>> ind=[4,2,5] % Create another array
```

```
ind =
```

```
     4     2     5
```

```
>> X(ind)% Index array X using array ind
```

```
ans =
```

```
     8     6     9
```

```
>> X([1,3,4])% Index X using array [1,3,4]
```

```
ans =
```

```
     5     7     8
```

- ▶ We can use this to assign multiple values simultaneously e.g.

```
>> X([1,2,4])=[12,13,14] % Assign values using arrays
```

```
X =
```

```
    12    13     7    14     9    10
```


Accessing elements using colon notation

Syntax

- ▶ This creates vectors using start, step and end values
- ▶ The notation is `start:step:end`
step is optional and defaults to 1
3:8 is equivalent to 3 4 5 6 7 8
25:-2:20 is equivalent to 25 23 21
- ▶ Enter `help colon` for more information
- ▶ We can use this notation to access multiple array elements

Accessing elements using colon notation

Some examples

- ▶ Colon notation allows us to access several elements at once
- ▶ Try these examples for yourself, removing the semicolons to show the results

```
>> Y(1:2:10)=1:4:20; % assign multiple values  
>> Y(5:end); % for elements 5th to the last  
>> Y=magic(7); % create matrix for next examples  
>> Y(3:4,5:7); % elements in rows 3 & 4, columns 5 to 7  
>> Y(:,7:-2:3); % elements in all rows, columns 7,5,3  
>> Y(:)=2; % assign all elements the value 2
```

Some useful array functions

- To find array size

```
>> A = rand(4,10);
```

```
>> size(A)
```

```
ans =
```

```
     4     10
```

```
>> nrow = size(A,1); % Number of rows
```

```
>> ncol = size(A,2); % Number of columns
```

- To find number of dimensions

```
>> ndims(A)
```

```
ans =
```

```
     2
```

Memory allocation - Preallocate arrays whenever possible!

- ▶ Array memory allocation is automatic
 - ▶ Arrays are extended as and when required
 - ▶ Undefined values are filled with zeros
 - ▶ Both are shown in the example below

```
>> clear; A(4)=5
```

```
A =
```

```
    0    0    0    5
```

- ▶ Extending arrays may be relatively slow
 - ▶ e.g. growing a vector by 1 element each time a value is inserted.
 - ▶ The entire array must be stored contiguously, so it may be necessary to copy all values to a new memory location each time the vector is extended.
- ▶ Advice: **preallocate arrays whenever possible**
 - ▶ i.e. create the array before using it
 - ▶ e.g. use zeros function to create the array, then insert the values

Array operations

- ▶ MATLAB (**MAT**rix **LAB**oratory) is designed to operate on matrices and arrays
- ▶ MATLAB functions will work with arrays, e.g.

```
>> A=[1,2,3,4]; B = sqrt(A)
```

```
B =
```

```
    1.0000    1.4142    1.7321    2.0000
```

- ▶ Operators also work on arrays

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

```
A =
```

```
    2    3    4    5
```

```
>> A(:)=2
```

```
A =
```

```
    2    2    2    2
```

Matrix operators

Syntax

- ▶ Some operators have two versions
 1. Array operator
Operates on individual elements (of same size arrays)
 2. Matrix operator
Used to perform matrix operations
- ▶ . (dot) is used to differentiate the array operator if **(and only if)** there are 2 versions
- ▶ e.g. matrix multiply * versus array multiply .* (i.e. dot *)
 $A*B$ multiplies matrix A by matrix B
 $A.*B$ multiplies individual elements of arrays A and B
- ▶ e.g. matrix power ^ versus array power .^
 A^2 is equivalent to $A*A$ i.e. matrix power
 $A.^2$ calculates the square of each element in array A

Matrix operators

Examples

```
>> a=magic(3)
```

```
a =
```

8	1	6
3	5	7
4	9	2

```
>> % matrix multiplication
```

```
>> a*a
```

```
ans =
```

91	67	67
67	91	67
67	67	91

```
>> % array multiplication
```

```
>> a.*a
```

```
ans =
```

64	1	36
9	25	49
16	81	4

```
>> % matrix power
```

```
>> a^2
```

```
ans =
```

91	67	67
67	91	67
67	67	91

```
>> % array power
```

```
>> a.^2
```

```
ans =
```

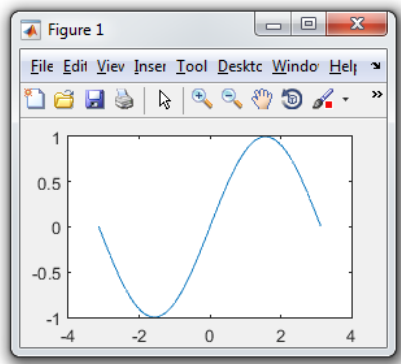
64	1	36
9	25	49
16	81	4

Plotting arrays

- ▶ MATLAB's plotting functions operate on arrays

For example to plot sine between $-\pi$ and π

```
>> x = linspace(-pi,pi,100);  
>> y = sin(x);  
>> plot(x,y)  
>>
```



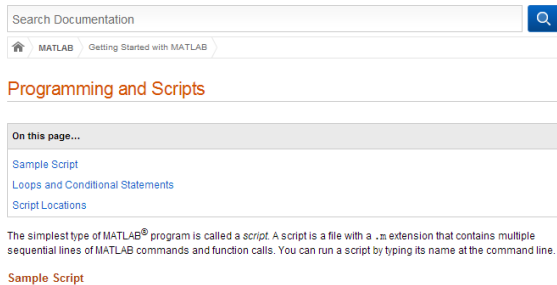
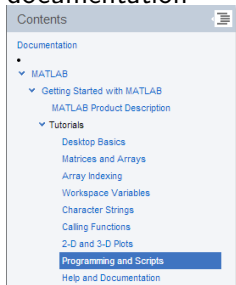
- ▶ `linspace(x,y,n)` creates n equally spaced values from x to y
- ▶ For more on plotting
 - ▶ Search MATLAB Help for Plotting Basics
 - ▶ <http://uk.mathworks.com/discovery/gallery.html>

Sparse arrays

- ▶ These only store non-zero values in memory
- ▶ Some calculation generate large matrices containing few non-zero values
e.g. finite element modelling
- ▶ It would be very inefficient to store and use all the zero values
- ▶ It would also limit the largest problem size which could be stored in memory
- ▶ Hence it may be useful to know about sparse arrays

Finding help

- ▶ MATLAB has very good documentation (see slide 8), which contains example code snippets. Copying and pasting example code at the prompt is a good way to learn about new features.
- ▶ There are tutorials contained within the MATLAB documentation



- ▶ **MATLAB central** is a user forum where you can find the answers to many MATLAB questions
- ▶ Search on google