Introduction to MATLAB for Engineers

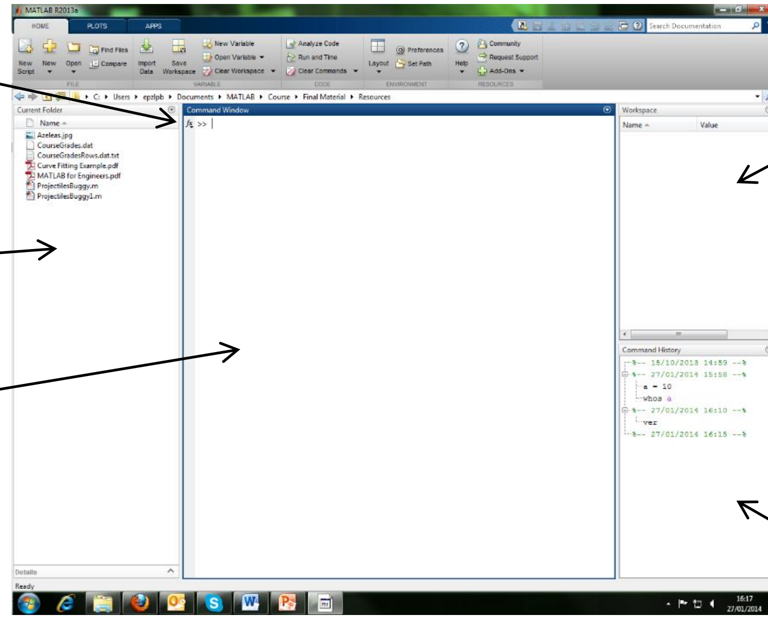
Louise Brown

http://moodle.nottingham.ac.uk/course/view.php?id=12439

The MATLAB Desktop

Function browser

Workspace window



Current folder window

Command window

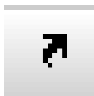
Command History window

To change path use dialog given by Environment->SetPath

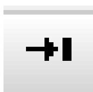
`path('Folder',path) or path(path,'Folder')`

Changing the desktop configuration

At top right hand corner of window gives drop down menu with options:



Dock and undock windows



Slide window to tabbed position at side of window

Restore maximised window to tiled position

Stack windows by dragging the title bars over each other

Save setup: Desktop -> Save Layout.

Restore default layout: Desktop->Desktop Layout->Default

Variables

- All variables in MATLAB are stored as arrays
- Scalar is 1x1 array
- Vector is 1xn or nx1 array
- Matrix is mxn array
- …and so on for multi-dimensional arrays

Variables are created using an assignment statement

`>> a = 10` creates a scalar variable

All variables are displayed in the workspace window

`who` – gives list of variable names

`whos` – gives name, size, type and memory size

Variable display format

- Use `format` command to change display format
- `format long` - fixed point format with 15 digits
- `format short` – fixed point format with 4 digits

- Note that format does not affect the way in which the variable is stored in memory
- Type `help datatypes` to see other available data types

Variable Names

- Must start with a letter but may include _ and numbers
- Case sensitive
- Certain reserved words cannot be used. Type `iskeyword` to see these
- Built in function names may be used but this will block access to the original functions. (Clear workspace to restore)
- Beware of overwriting pi, i and j

  i and j are used as complex numbers by default

  Type `i` or `j` at the command prompt

```
0 + 1.0000i
```

Saving Workspace Data

MATLAB format:

Variables in the workspace can be saved to a file using the Save Workspace button or by typing `save <filename>` at the command line

Using `save('filename', 'var1', 'var2',…)` allows selected variables to be saved

Saves with .mat extension in format readable by MATLAB

Has advantage that it is operating system independent

ascii format:

Use a '-ascii' parameter to save data in ascii format.

```
>> save('Data.dat','x','y', '-ascii')
```

Conventionally a .dat file extension is used

Vectors

```
>> A = [1 2 3 4]
>> A = [1,2,3,4]
```

Both give 1x4 vector

```
A =
      1 2 3 4
```

`>> A = [1:4]` Colon operator gives regularly spaced elements

Specify increment using format [first:step:last]

`>> B = [1:2:9]` gives `B = 1 3 5 7 9`

Increment can be decimal as well as integer

`>> C = [1:0.2:2]` gives

```
C =
    1.0000 1.2000 1.4000 1.6000 1.8000 2.0000
```

More Vectors

Create a vector with a given number of equispaced points using linspace(start, end, number of elements)

```
>> vec = linspace( 1, 8, 5 )
    vec = 1.0000    2.7500    4.5000    6.2500    8.0000
```

logspace can be used to create logarithmically spaced points

y = logspace(a,b,n) generates n points between decades 10^a and 10^b

```
>> vec = logspace(1,2,5)
vec =
   10.0000   17.7828   31.6228   56.2341  100.0000
```

Character Vectors

Create character vectors using single quotes

```
charVec = 'Here are some characters'
```

Stored as 1x24 char array

Type `help strings` for more information, or `doc strings` for full documentation

Column Vectors

Either semicolon operator

```
>> A = [1;2;3;4]
```

can be used to create a column vector:

or transpose operator

```
>> A = [1:4]'

 A =

      1
      2
      3
      4
```

Indexing Vectors

A =

10	3	20	7	30		11	13

Subscript notation: A(3) gives 5

Subscript can be a vector: A([1 4 6]) gives 1 7 11

Colon notation to extract range: A(2:5) gives 3 5 7 9

Use ‘end’ either on its own or in range: A(4:end) gives 7 9 11 13

Replace element using index on left: A([1 3 5]) = [10 20 30]

Exercise 2.1

1)
```
>> A = [5:2:17]
A =
5     7     9    11    13    15    17
```

2)
```
>> B = [3:-1:-3]'
B =
      3
      2
      1
      0
     -1
     -2
     -3
```

3)
```
>> C = A([1 4 6])
C =
      5    11    15
```

4)
```
>> B([3 5 7]) = C
B =
      3
      2
      5
      0
     11
     -2
     15
```

Arithmetic Operations

Addition	a + b
Subtraction	a – b
Multiplication	a * b for matrix or scalar multiplication
	a.*b for element by element multiplication
Division	a / b for matrix or scalar division
	a./b for element by element division
Left division	a \ b for matrix or scalar left division
	a.\b for element by element left division
Exponentiation	a ^ b for matrix or scalar exponentiation
	a.^b for element by element exponentiation

Order of Precedence

Arithmetic operations will be carried out in the order:

- Evaluate parentheses working from inside to outside
- Exponential operations (^)
- Multiplication (*) and division (/) from left to right
- Addition (+) and subtraction (-) from left to right

Addition and Subtraction Operations

Scalar addition and subtraction performs the operation on each member of the vector or array:

```
>> A = [1:4]
A =
      1   2   3   4
>> A = A + 10
A =
     11  12  13  14
```

Note – this is achieved without using a loop

Array addition and subtraction is performed element by element

```
>> B = [5:8]
B =
      5   6   7   8
```

```
>> A = A + B
A =
     16  18  20  22
```

Multiplication Based Operations

Multiplication (and /,\,^) operate element by element on scalars in the same way as the addition operator.

```
>> A = [1:4]
A =
      1   2   3   4

>> A = A * 10
A =
      10   20   30   40
```

Multiplication Based Operations

To multiply vectors or arrays element by element the 'dot' operators must be used. Note that vectors must be the same size.

```
>>A = [1:4]
A =
     1  2  3  4

>> B = [2:5]
B =
     2  3  4  5

>> D = A.*B
D =
     2  6  12  20
```

Note that just using the * operator here will result in an error

Multiplication Based Operators

Using multiplication based operators without a dot operator will result in standard matrix operations

```
>> A = [1:4]
A =
     1   2   3   4

>> B = [1:4]'
B =
     1
     2
     3
     4
```

So that

```
>> A*B
ans =
    30
```

Inner dimensions must match

1 x 4   *   4 x 1

Outer dimensions give final size

but

```
>> B*A
ans =
     1     2     3     4
     2     4     6     8
     3     6     9    12
     4     8    12    16
```

Vectors as Function Input

Vectors can be used as input arguments to functions – both built-in and user defined

```
>> A = [0:pi/6:pi]

A =

  0 0.5236  1.0472   1.5708  2.0944  2.6180  3.1416

>> B = sin(A)

B =

  0  0.5000  0.8660  1.0000   .8660  0.5000  0.0000
```

Exercise 2.2

1) >> 1/(2+3^4) + 5/(6*7) + 7/8

```
ans =
        1.0061
```

2) >> A = linspace(0, 2*pi, 9)

```
A =
    0    0.7854    1.5708    2.3562    3.1416    3.9270  …      4.7124    5.4978    6.2832
```

>> SinAng = sin(A)

```
SinAng =
    0    0.7071    1.0000    0.7071    0.0000   -0.7071  …    -1.0000   -0.7071   -0.0000
```

3) >> SinAng = SinAng + 1

```
SinAng =
 1.0000    1.7071    2.0000    1.7071    1.0000    0.2929       0    0.2929    1.0000
```

Exercise 2.2

```
4) >> B = [1:9]
   B =
    1    2    3    4    5    6    7    8    9

>> SinAng.*B
ans =
    1.0000    3.4142    6.0000    6.8284    5.0000    1.7574         0    2.3431    9.0000
```

Left Division

$a\backslash b = a^{-1}b$ for matrix operations

$a.\backslash b = b/a$ when performed element by element on arrays or when one of the operands is a scalar

```
>> A = [1:4]
A =
     1     2     3     4

>> A.\2
ans =
    2.0000    1.0000    0.6667    0.5000
```

Using Left Division to Solve Simultaneous Equations

The pair of simultaneous equations

$$3x_1 + 2x_2 = 12$$
$$x_1 + 4x_2 = 14$$

can be written as $Ax = B$

where $A = \begin{bmatrix} 3 & 2 \\ 1 & 4 \end{bmatrix}$, $B = \begin{bmatrix} 12 \\ 14 \end{bmatrix}$, and $x = \begin{bmatrix} x1 \\ x2 \end{bmatrix}$

Solving using linear algebra gives $x = A^{-1}B$

In MATLAB set up the arrays

```
>> A = [3 2; 1 4]
A =
     3     2
     1     4

>> B = [12;14]
B =
    12
    14
```

and solve using the left division operator

```
>> x = A\B
x =
     2
     3
```

Matrices

Use the semicolon operator to separate the rows of a matrix

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

The colon operator can be used as for vectors.
The same result is given by

```
>> A = [1:5;5:9]
```

Exercise 2.3

```
>> A = [1 2 3; 2 3 -1; 4 -1 2]
A =
     1     2     3
     2     3    -1
     4    -1     2

>> B = [13;4;13]
B =
    13
     4
    13

>> X = A\B
X =
    2.0000
    1.0000
    3.0000
```

Character Matrices

Arrays of characters must have the same number of elements in each row.

O	N	E		
S	E	V	E	N
F	I	V	E	

```
>> charArray = [‘ONE';‘SEVEN';‘FIVE']
Error using vertcat
Dimensions of matrices being concatenated are not
consistent.
```

Character Matrices

Use `char` function to create arrays with padded strings

```
>> charArray = char('one','seven','five')

charArray =
one
seven
five
```

Creates 3x5 char array with spaces padding shorter strings

O	N	E		
S	E	V	E	N
F	I	V	E	

String Scalars

Used to store a group of characters as a `string` type

```
>> str = “one”
str =
      “one”

>> strArray = [“one”; “seven”; “five”]
strArray =
   3x1 string array
      “one”
      “seven”
      “five”
```

Addition operator can be used to concatenate strings:

```
>> str = “one” + “ seven” + “ five”
   str =
      “one seven five”
```

Indexing String Scalars

To access the string:

```
>> strArray(2)
      ans = “seven”
```

To access the char vector:

```
>> strArray{2}
      ans = ‘seven’
>> strArray{2}(3)
      ans = ‘v’
```

Matrix Indexing

$$A = \begin{matrix} 1 & 2 & 3 & 4 & 5 \\ 5 & 6 & 7 & 8 & 9 \\ 9 & 10 & 11 & 12 & 13 \end{matrix}$$

row column

Subscript notation: A(3,2) = 10

Colon notation to extract range in both rows and columns: A(2:3, 3:4) = $\begin{matrix} 7 & 8 \\ 11 & 12 \end{matrix}$

Colon notation to extract whole row or column: A(2, :) = 5 6 7 8 9

Or several rows or columns: A(: , [1 3 5]) = $\begin{matrix} 1 & 3 & 5 \\ 5 & 7 & 9 \\ 9 & 11 & 13 \end{matrix}$

Linear Indexing (1)

$$A = \begin{matrix} 1 \;\; 1 & 2 \;\; 4 & 3 \;\; 7 & 4 \;\; 10 & 5 \;\; 13 \\ 5 \;\; 2 & \boxed{6 \;\; 5} & 7 \;\; 8 & 8 \;\; 11 & 9 \;\; 14 \\ 9 \;\; 3 & 10 \;\; 6 & 11 \;\; 9 & 12 \;\; 12 & 13 \;\; 15 \end{matrix}$$

Linear index is obtained by counting down each column in turn

Linear index: A(5) = 6

Linear Indexing (2)

A =					
	1 1	2 4	$\boxed{3 \; 7}$	4 10	5 13
	5 2	6 5	7 8	$\boxed{8 \; 11}$	9 14
	9 3	10 6	11 9	12 12	13 15

Find indices for elements (1, 3) and (2, 4)

Use `sub2ind( size, row vector, column vector )` to convert from subscript to linear index

(1, 3) and (2, 4)

Row vector `[1   2]`   Column vector `[3   4]`

```
>> Index = sub2ind( size(A), [1  2], [3  4] )
Index =
      7   11
```

Then extract actual elements using the linear index vector: `A(Index) =  3 8`

Combining Matrices

Larger arrays can be built up from smaller ones:

```
>> A = [1:5;5:9]
A =
     1     2     3     4     5
     5     6     7     8     9

>> B = [1:2:9;2:2:10]
B =
     1     3     5     7     9
     2     4     6     8    10

>> D = [A;B]
D =
     1  2  3  4  5
     5  6  7  8  9
     1  3  5  7  9
     2  4  6  8  10

>> E = [A B]
E =
     1  2  3  4  5  1  3  5  7  9
     5  6  7  8  9  2  4  6  8  10
```

Note that the dimensions of arrays being joined must be consistent.

Matrix Functions

A = 1 2 3 4 5
    5 6 7 8 9

length(X) returns the largest dimension of an array: length(A) = 5

size(X) returns a vector containing the length of each dimension of the array:

```
>> dims = size(A)
dims =
      2     5
```

The rows and columns can also be returned as separate variables:

```
>> [row,column] = size(A)
row =
     2
column =
     5
```

Useful automatically generated matrices

Identity matrix: eye(X)

Only one parameter as the identity matrix is square

```
>> eye(2)
ans =
     1     0
     0     1
```

p… for matrices of more than two dimensions

All ones: ones(m, n, p…)

```
>> ones(2,3)
ans =
     1     1     1
     1     1     1
```

All zeros: zeros(m, n, p…)

```
>> zeros(3,2)
ans =
     0     0
     0     0
     0     0
```

Matrix Memory Management

Extra elements may be added to an array simply by allocating an element at a given position

```
A =  1 2
     3 4
```

Adding an element at (5,5) grows the array:

```
>> A(5,5) = 10
A =
       1  2  0  0  0
       3  4  0  0  0
       0  0  0  0  0
       0  0  0  0  0
       0  0  0  0  10
```

A new block of memory will be allocated each time extra elements are added to the array – this is slow!

If the final size of the array is known create an array of the final size (typically using the zeros() function) and then assign elements as required

Sparse Arrays

For arrays with large numbers of zero elements the `sparse(N)` function can be used to squeeze out zero values.

A =

1	2	0	0	0
3	4	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	10

`sparse(A)` reduces the size of A from 200 to 84 bytes

Exercise 2.4

1) >> A = [9 12 13 0;10 3 1 5;2 5 10 3]

```
A =
    9   12   13    0
   10    3    1    5
    2    5   10    3
```

>> B = [1 4 2 11;9 8 16 7;12 5 0 3]

```
B =
    1    4    2   11
    9    8   16    7
   12    5    0    3
```

2) >> C = A(3,3)

```
C =
   10
```

>> D = A(:,3)

```
D =
   13
    1
   10
```

>> E = [B(1,:);B(3,:)]

```
E =
    1    4    2   11
   12    5    0    3
```

>> F = [A;B]

```
F =
    9   12   13    0
   10    3    1    5
    2    5   10    3
    1    4    2   11
    9    8   16    7
   12    5    0    3
```

>> G = [A(:,1) B(:,4)]

```
G =
    9   11
   10    7
    2    3
```

Exercise 2.4

3) >> E(2,2) = 20

```
E =
     1     4     2    11
    12    20     0     3
```

```
>> A(1,:) = 0
A =
     0     0     0     0
    10     3     1     5
     2     5    10     3
```

```
>> F(:,3) = 1:6
F =
     9    12     1     0
    10     3     2     5
     2     5     3     3
     1     4     4    11
     9     8     5     7
    12     5     6     3
```

```
>> A(:,1) = B(:,2)
A =
     4     0     0     0
     8     3     1     5
     5     5    10     3
```

Scripts

Script files are created by selecting New Script or typing ‘edit’ at the command prompt.

MATLAB script files are saved with a ‘.m’ extension (by default to the current folder)

Scripts are run by typing the script name at the command prompt

Comment lines start with %

The first comment line in a script is the H1 line which is used as a help for the script.

A semicolon at the end of the line will suppress output to the monitor

Simple User Input

Use `input(str)` command to fetch user input

```
>> value = input( 'Enter a value: ');
Enter a value:  10
```

User input

A second parameter 's' indicates the input is a character or string.

```
>> str = input('Input yes or no ','s')
Input yes or no yes

str =
      yes
```

Output

Use the disp(str) function for simple output

To display a variable use `disp(A)`

Or to output a string:

```
% Create a string
str = 'Output from disp function';
% Pass the string as a parameter to the disp function
disp( str );

>> Output from disp function
```

Output

Simple strings can be assembled to give output data.

Use the num2str function to convert numerical data into a string

```
>> Vel = 20;

>> strTitle = ['Starting speed = ' num2str(Vel) ' m/s'];

>> disp(strTitle);


Starting speed = 20 m/s
```

Or using string scalars:

```
>> strTitle = “Starting speed “ + num2str(Vel) + “ m/s”
```

Freefall Script

Write a script called FreeFallExample.m which calculates the distance travelled by a freely falling object at ten time increments from release to time, t, using the equation

$d = 0.5gt^2$

The time will be user input.
Output the distances calculated.

Freefall Script

Exercise 4.1

Write a script called FreeFall.m which calculates the distance travelled by a freely falling object at each time increment between two given times using the equation

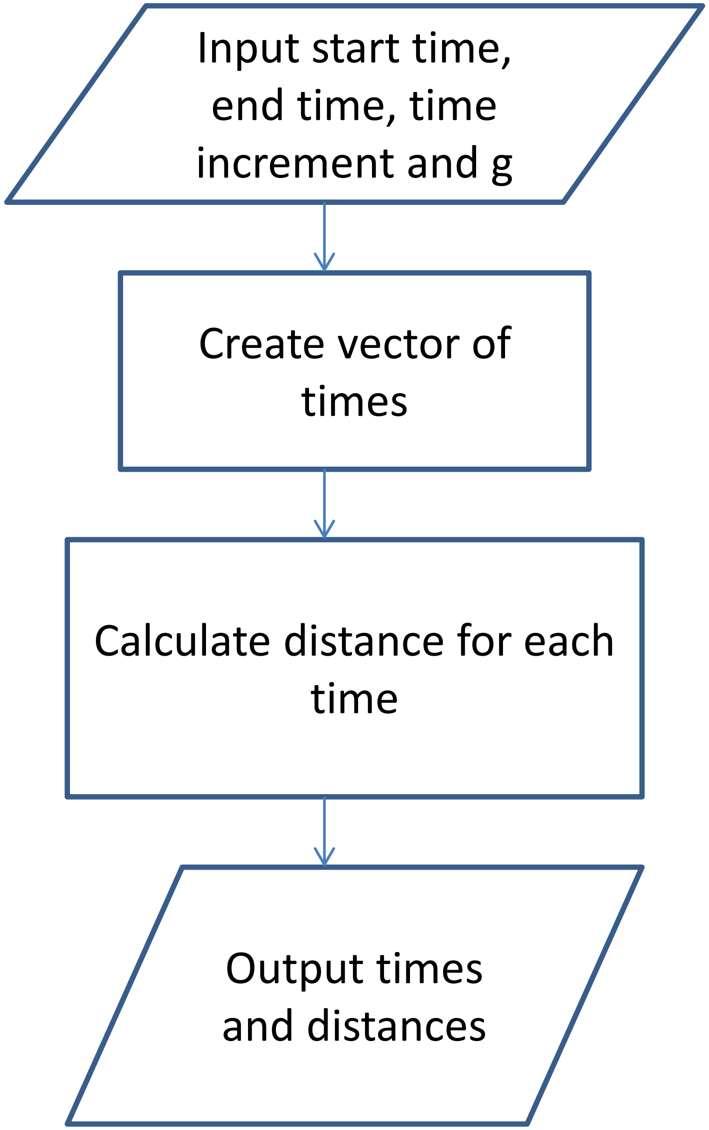
$d = 0.5gt^2$

The **start** and **end times** and **time increment** will be **user input**. Output the distances calculated.

(For more of a challenge present the output as two columns showing times and distances)

Exercise 4.1

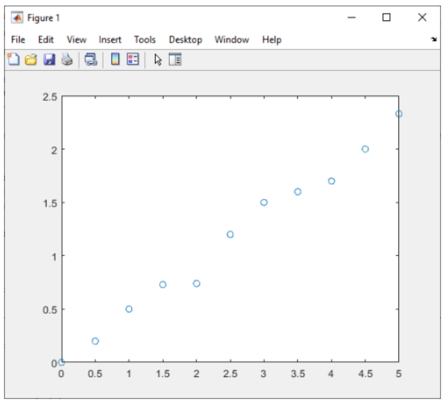
Exercise 4.1


Input start time, end time, time increment and g
Create vector of times
Calculate distance for each time
Output times and distances

Simple x-y plots

plot(x,y) plots vector x against vector y

```
>> t = 0:0.5:5
>> distance = [0, 0.2, 0.5, 0.73, 0.74, 1.2,…
1.5, 1.6, 1.7, 2.0, 2.33]
>> plot(t, distance, 'o')
```

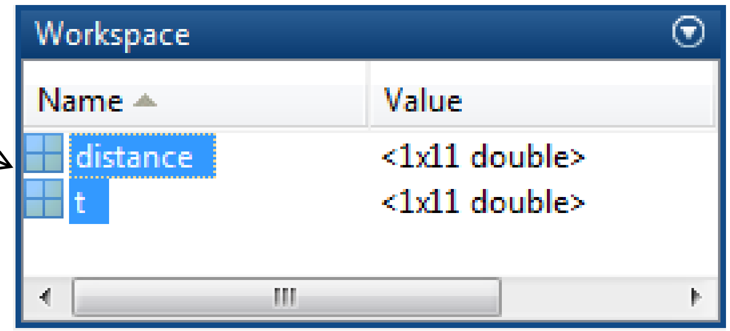


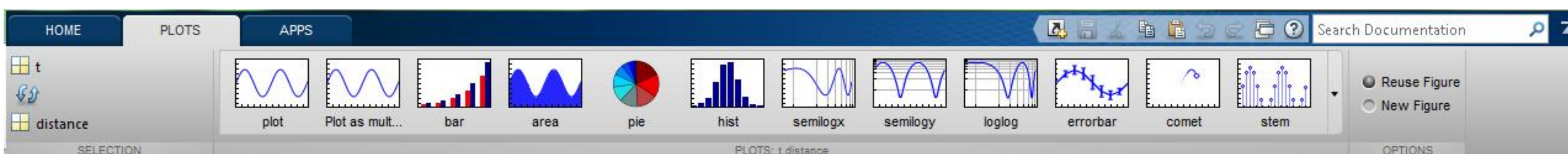
Third parameter used to specify line style, marker type and colour

plot command automatically fits axes to the data.
To control axis scaling use: axis([xmin, xmax, ymin, ymax])

Plot from Workspace

Select variables to plot





Select plot type from ribbon

Annotating Figures

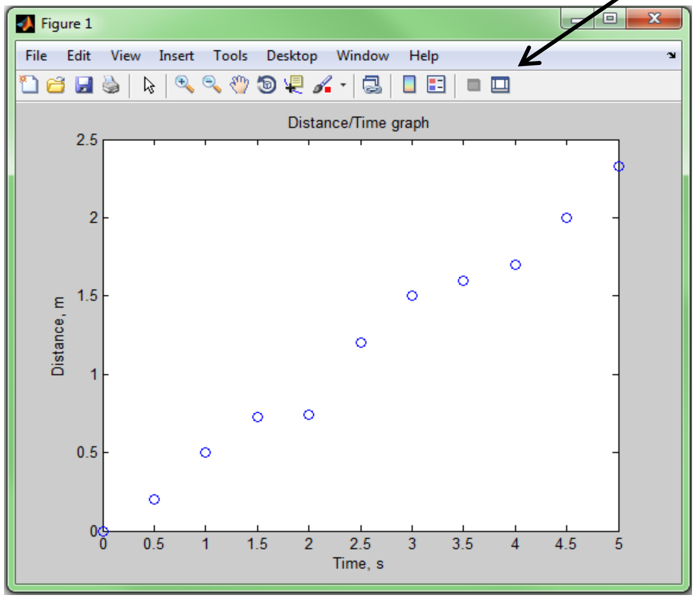
From the script:

```
>> xlabel( 'Time, s')
>> ylabel('Distance, m')
>> title('Distance/Time
graph')
```

Note – plot command must be executed first

Or using Plot Tools:

- Use Show Plot Tools and Dock Figure icon
- Or type 'plottools' at the command prompt
- Or using the live editor



Script for similar plots can be created using File->Generate code

Plotting Multiple Data Sets

Various methods of plotting multiple data sets on one figure:

`hold on` - any plot commands will plot on same figure until `hold off` is executed

Send multiple sets of data in one plot command

```
>> plot( t, distance, ‘s’, t, distance2, ‘o’ )
```

Collect multiple sets of data into one matrix

```
>> distances = [distance;distance2]
>> plot( t, distances, ‘o’ )
```

Creating multiple figures:

Select the current figure to be plotted to using the `figure(n)` command.

Any subsequent plots will be performed on this figure

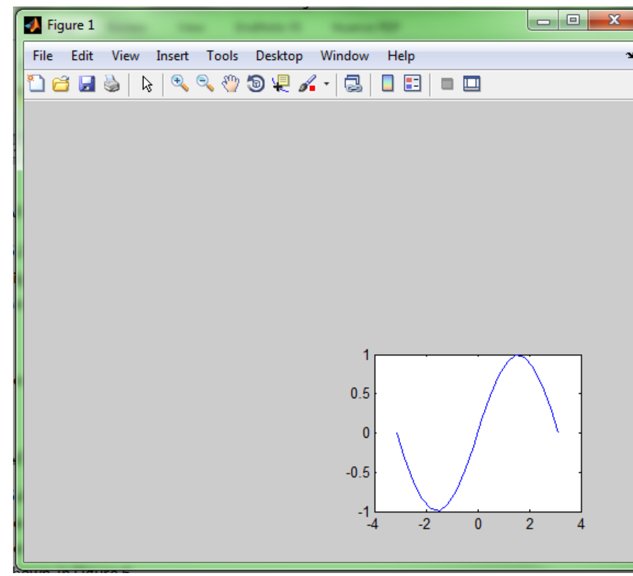
Subplots

Several plots in the same window

p = 1	p = 2
p = 3	p = 4

Use `subplot(m,n,p)` command.
Window is split into grid of m rows by n columns.
p selects the window for plotting.

```
>> subplot(2,2,4)
% Call plot commands before calling
subplot again for next location
```



Saving Figures

Save figures using `saveas` function:

saveas(gcf, ‘filename’, ‘format’)

returns the handle to the current figure

```
>> saveas( gcf, ‘Figure’,‘png’)
```

Saves to Figure.png

File->Save   Saves a Matlab .fig file
File->Save As…   Saves to standard image formats

Exercise 5.2

Suggested initial values:
Start velocity = 60 m/s
Launch angle = pi/3

Plot horizontal distance on x-axis and vertical distance on y-axis

Format the title (to include the velocity and angle) using a vector of strings

Remember to convert the angle back into degrees to display in the title

Use `num2str` to create strings from variables to include in the string, eg '30 degrees'

Debugging

3 types of errors: syntax, runtime and logical

Syntax errors:

At the command line an error message is displayed
In the edit window: orange bar – warning
red bar - error

Runtime errors:

Occur as code executes, eg out of bounds index
Within a script an error message is given including the line number

Note that in MATLAB divide by 0 does not generate an error – a value of Inf is assigned and the program continues to run

Logical errors:

Hardest to find. Program runs but gives incorrect results
In MATLAB can use Code Cells or the built-in debugger

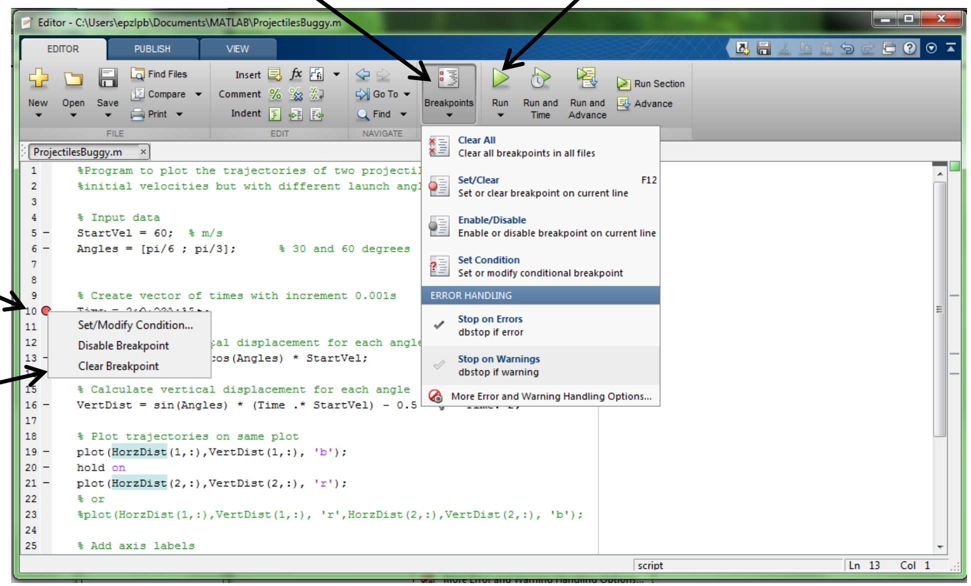
Using the Built-in Debugger

Set/clear breakpoint (F12)

Save and run (F5)

Breakpoint

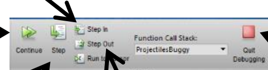
Right click on breakpoint to display options



When running debug the menu is displayed:

Step into (F11)

Continue (F5)



Exit debug mode (Shift + F5)

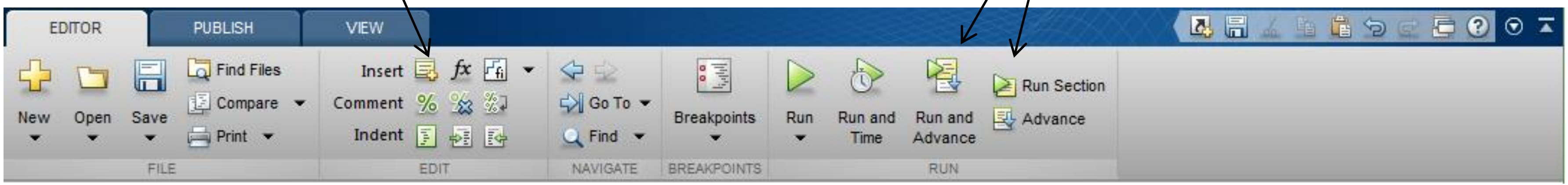
Step (F10)

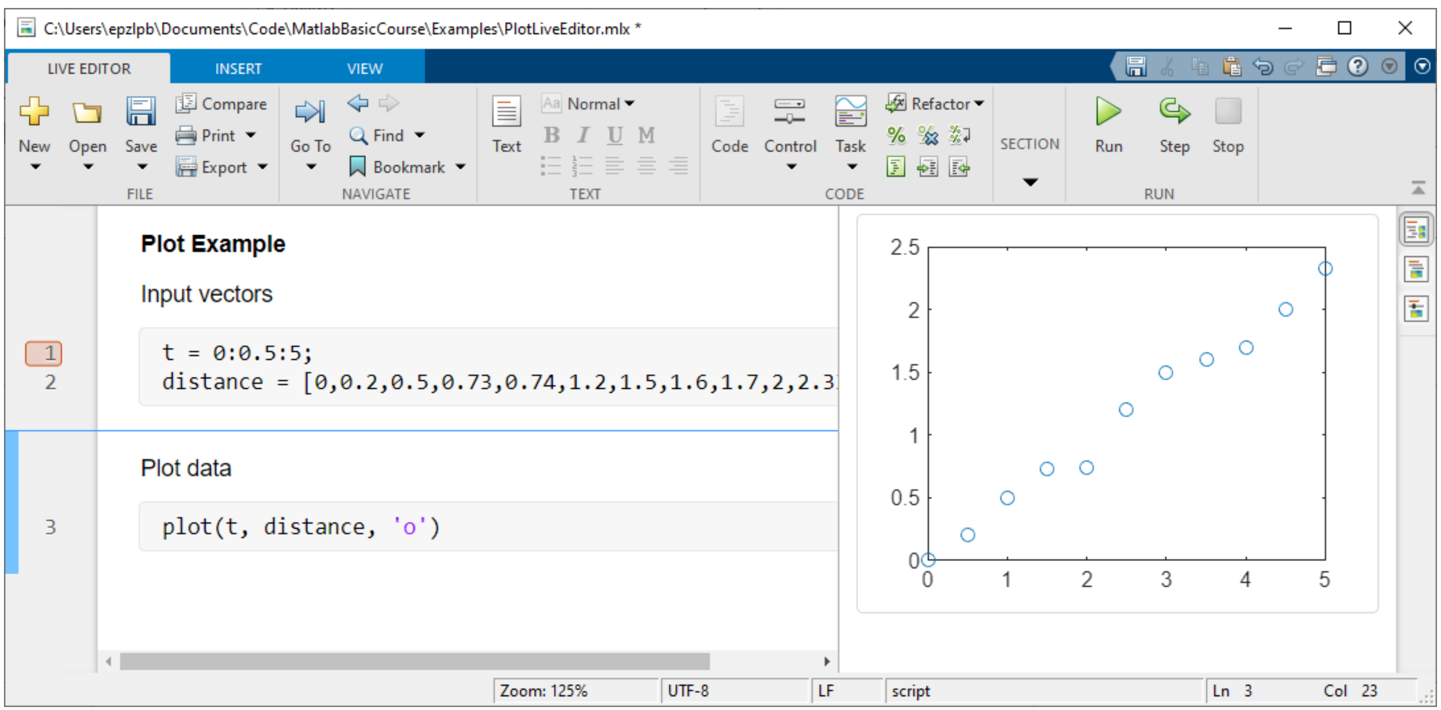
Step out (Shift + F11)

Code Sections and Live Editor

Insert section break

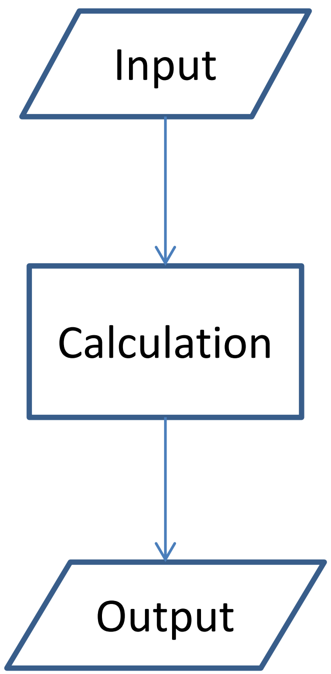
Run section options



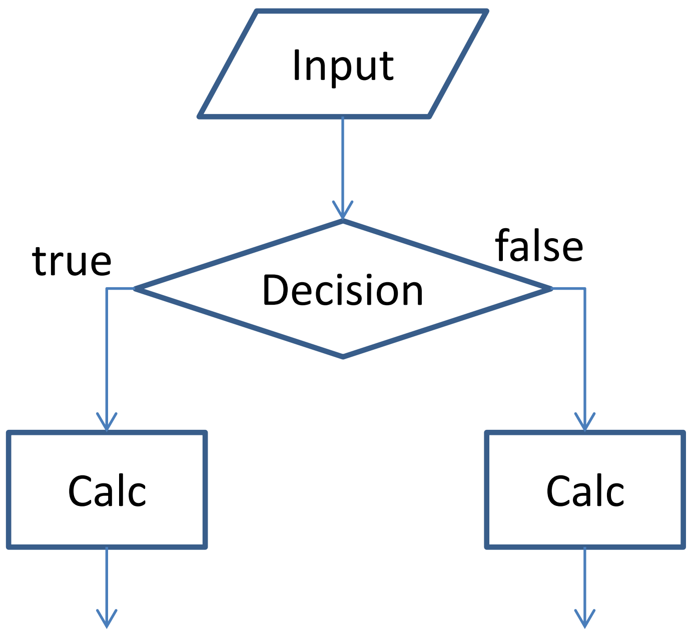


Program Structure

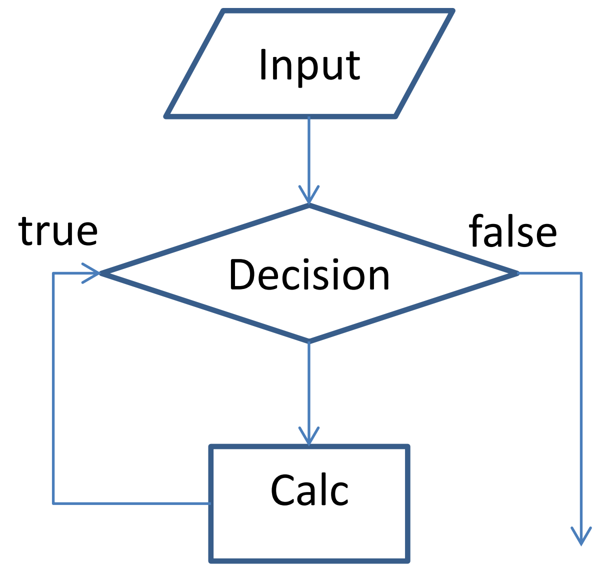
Sequential



Selection



Repetition



Relational and Logical Operators

Logical data type: `a = true` or `a = false` – occupies 1 byte
convert to logical using b = logical(x)

Relational
a < b   less than
a <= b   less than or equal to
a > b   greater than
a >= b   greater than or equal to
a == b   equal to
a ~= b   not equal to

Logical
a & b   and
a | b   or
~ a   not
xor(a,b)   exclusive or

Comparing Matrices

Comparisons on matrices will compare corresponding elements and create a matrix of the results using the logical data type

```
>> a = 1:5

a =
     1     2     3     4     5

>> b = [3 1 2 4 0]

b =
     3     1     2     4     0

>> c = a > b

c =
     0     1     1     0     1
```

Logical Indexing

Relational and logical operators are performed element-wise on vectors or matrices resulting in a logical matrix:

```
A =  1 2 3 4 5
     3 4 5 6 7
```

Note that for compound tests each test is written out in full: `A>2` and `A<6`

```
>> B = A > 2 & A < 6
B = 0 0 1 1 1
    1 1 1 0 0
```

Can't use 2 < A < 6 type format

Logical matrix B can be used as a mask to perform operations on selected elements of a matrix

```
>> A(B) = sqrt(A(B))
A =  1.0000   2.0000   1.7321   2.0000   2.2361
     1.7321   2.0000   2.2361   6.0000   7.0000
```

Logical Indexing(2)

```
A =  1 2 3 4 5          B = 0 0 1 1 1
     3 4 5 6 7              1 1 1 0 0
```

Extract the elements which satisfy the test: `>> C = A(B)`

```
C =
      3
      4
      3
      5
      4
      5
```

To find the indices of the elements which satisfy the test use the find() function

```
>> D = find(A>2 & A<6)
D =
     2
     4
     5
     6
     7
     9
```

find()

Index = find(x,1) – returns the linear index of the first nonzero element, or first element which satisfies a specified condition

```
>> B = [0 0 0 3 4 5 6]

B =
     0     0     0     3     4     5     6

>> find( B,1 )

ans =
     4

>> find( B>4, 1)

ans =
     6
```

Floating Point Comparisons

Comparison of floats or doubles

Small differences resulting from floating point arithmetic may cause errors when checking for equality.

```
>> a = 0.5-0.4-0.1
a =
    -2.775557561562891e-17
```

Use a tolerance check:

```
      >> abs(a) < tolerance

A
      0.9999999999

if ( A == 1)   % will be false

Tol = 1e-6;
abs( A - 1.0 ) < Tol   % will return true
```

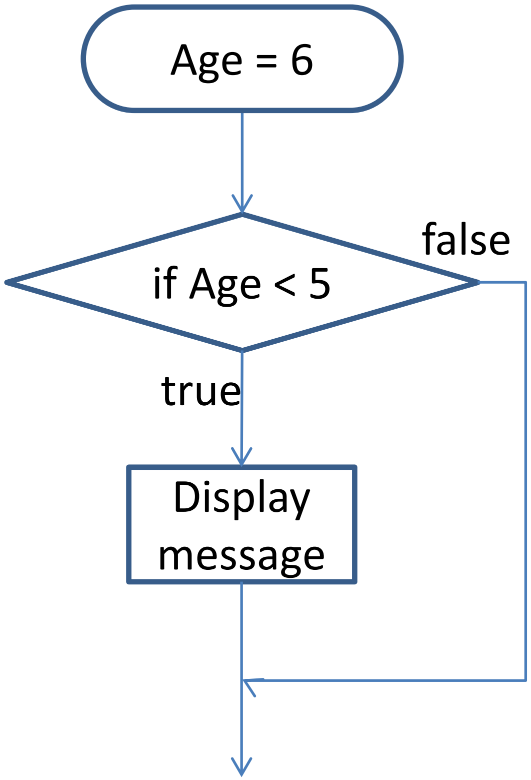
Exercise 8.1

```
1) >>  A = [9 12 18 0;10 3 1 7;2 5 14 22]
A =
     9    12    18     0
    10     3     1     7
     2     5    14    22

>> B = A >=7 & A < 15
B =
     1     1     0     0
     1     0     0     1
     0     0     1     0

2)              >> A(B) = A(B).^2
A =
    81   144    18     0
   100     3     1    49
     2     5   196    22
```

```
3)              >> find(A>100)
ans =
     4
     9

4)              >> B = A(A>100)
B =
   144
   196
```

Conditional Statements

Selection

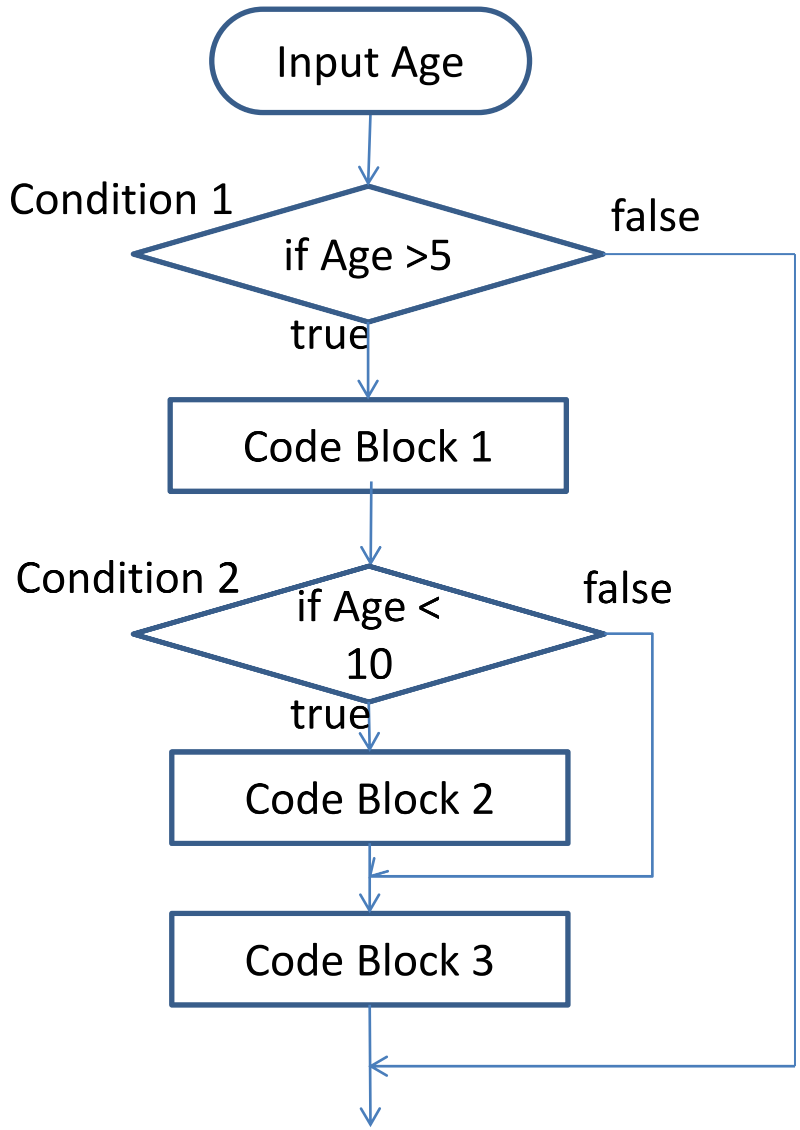



```
if condition
      Code block
end
```

```
Age = 6;
if Age < 5
    disp('a is less than 5');
end
```

Note that if the condition is applied to an array it will only be true if *all* elements of the array satisfy the condition

Nested if statements



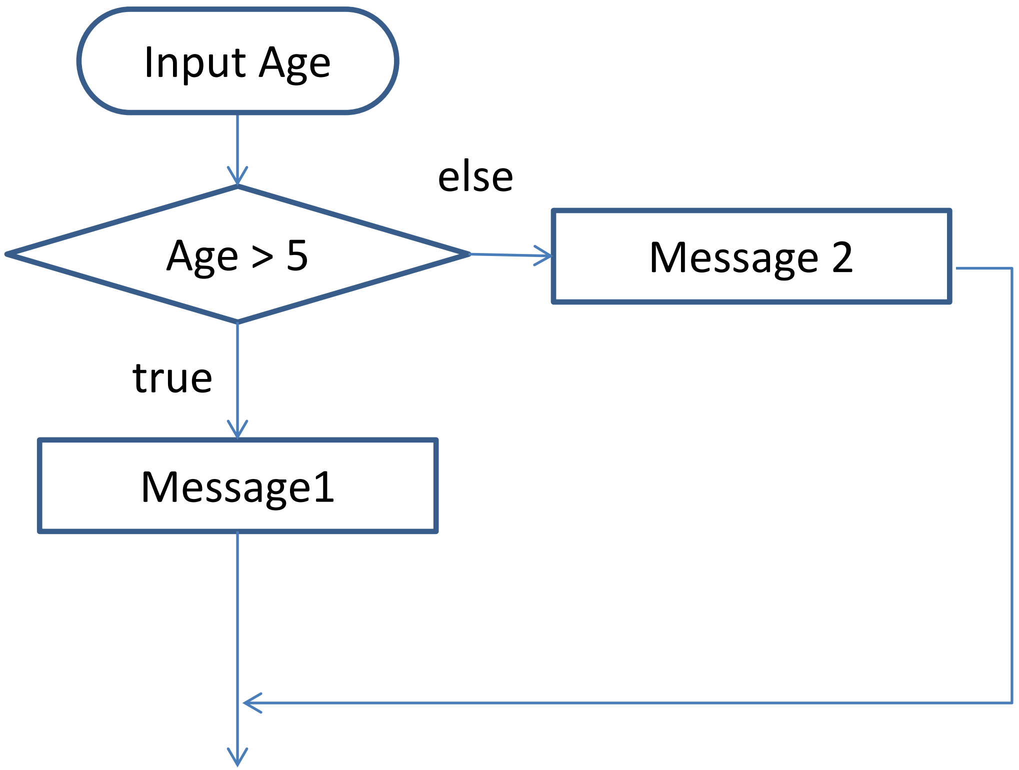
if statements can be nested:

```
if condition1
    Code block 1
    if condition 2
        Code block 2
    end    % End of condition 2
    Code block 3
end        % End of condition 1
```

Remember the ‘end’ statement at the end of each code block

Indentation is not necessary but is good style and makes code more readable

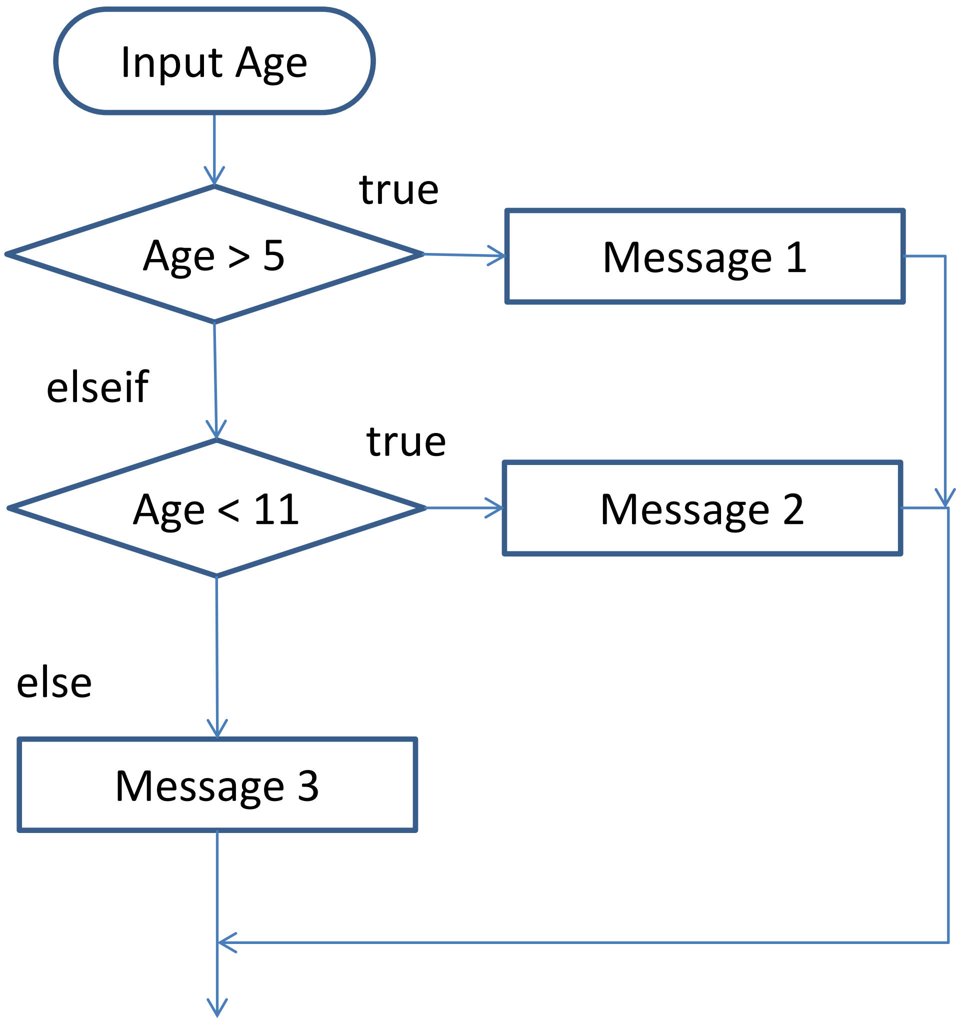
if/else Statements




```
if condition 1
        Code block 1
else
        Code block 2
end
```

```
if age < 5
    disp('Age is less than 5');
else
    disp('Age is 5 or over');
end
```

if/elseif/else Statements




```
if condition 1
        Code block 1
elseif condition 2
        Code block 2
else
        Code block 3
end
```

if/elseif/else Statements

```
if age < 5
    disp('Age is less than 5');
elseif age < 11
    disp('Age is between 5 and 10');
else
    disp('Age is 11 or over');
end
```

Don't overspecify conditions:

elseif age < 11 & age >=5 - second condition is redundant

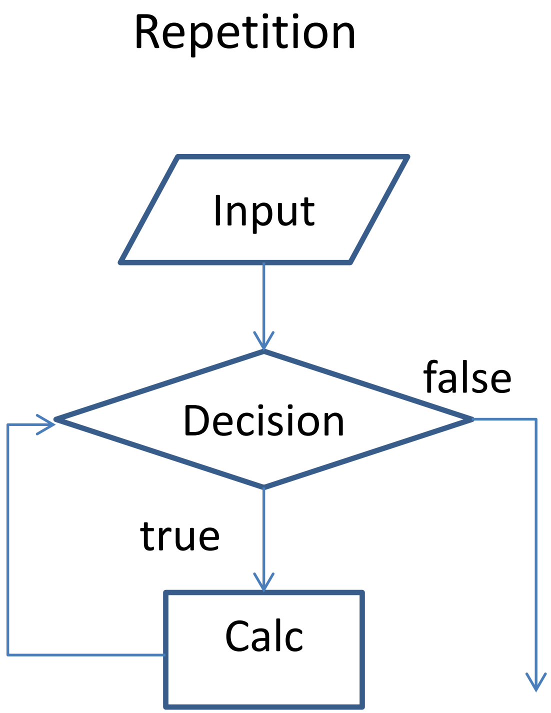
elseif age < 11 & age >5 - would give incorrect result for age = 5

Switch Statement

switch variable
    case value1
        Code block 1
    case value2
        Code block 2
    …
    case value n
        Code block n
    otherwise
        Code executed if expression none of values specified
end

```
%Menu and switch/case statement example
Colour = menu('Select Colour', 'Red',
'Blue', 'Green');
switch Colour
    case 1
        disp('You chose red');
    case 2
        disp('You chose blue');
    case 3
        disp('You chose green');
    otherwise
        disp('You didn''t make a
selection');
end
```

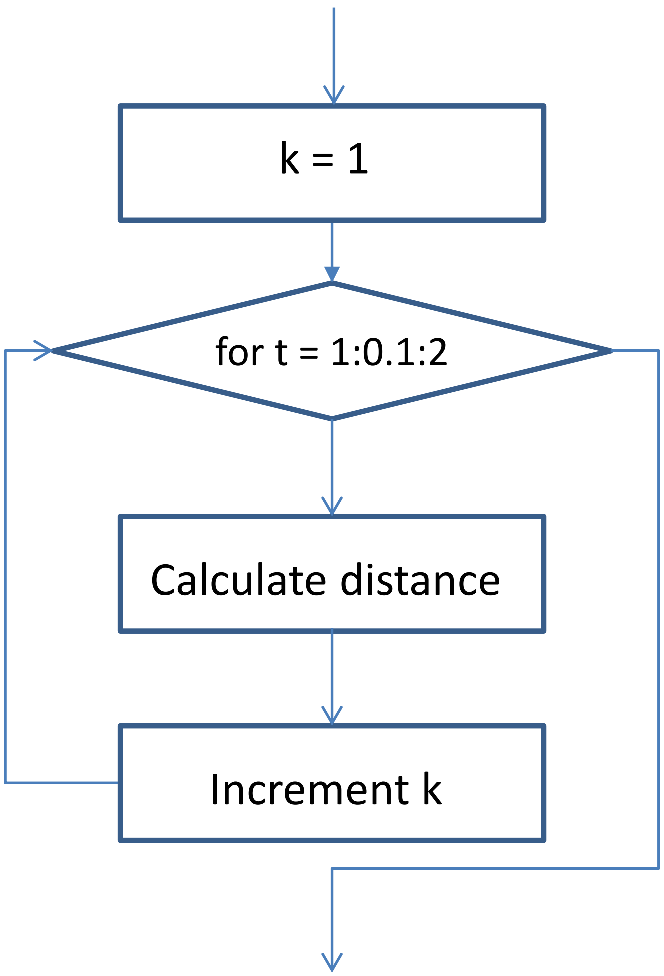
Repetition Operators



For loop:
Use to iterate through a given set of values (given by an index matrix)

While loop:
Loop until a specified condition is satisfied

for Loops




```
for index = [matrix]
        Code block
end
```

Note that step doesn't have to be integer

```
k = 1;
for t = 1:0.1:2
    % Calculate distance in freefall
    d(k) = 0.5 * 9.81 * t^2;
    k = k+1;
end
```

This will grow the array d by one element on each iteration

for Loops

```
k = 1;
for t = 1:0.1:2
    % Calculate distance in freefall
    d(k) = 0.5 * 9.81 * t^2;
    k = k+1;
end
```

Note that vectorised form is much more efficient:

```
t = [1:0.1:2];
d = 0.5 * 9.81 * t.^2;
```

```
for index = [matrix]
      Code block
end
```

If 2D matrix the index will contain a column of the matrix for each iteration of the loop – ie a column vector

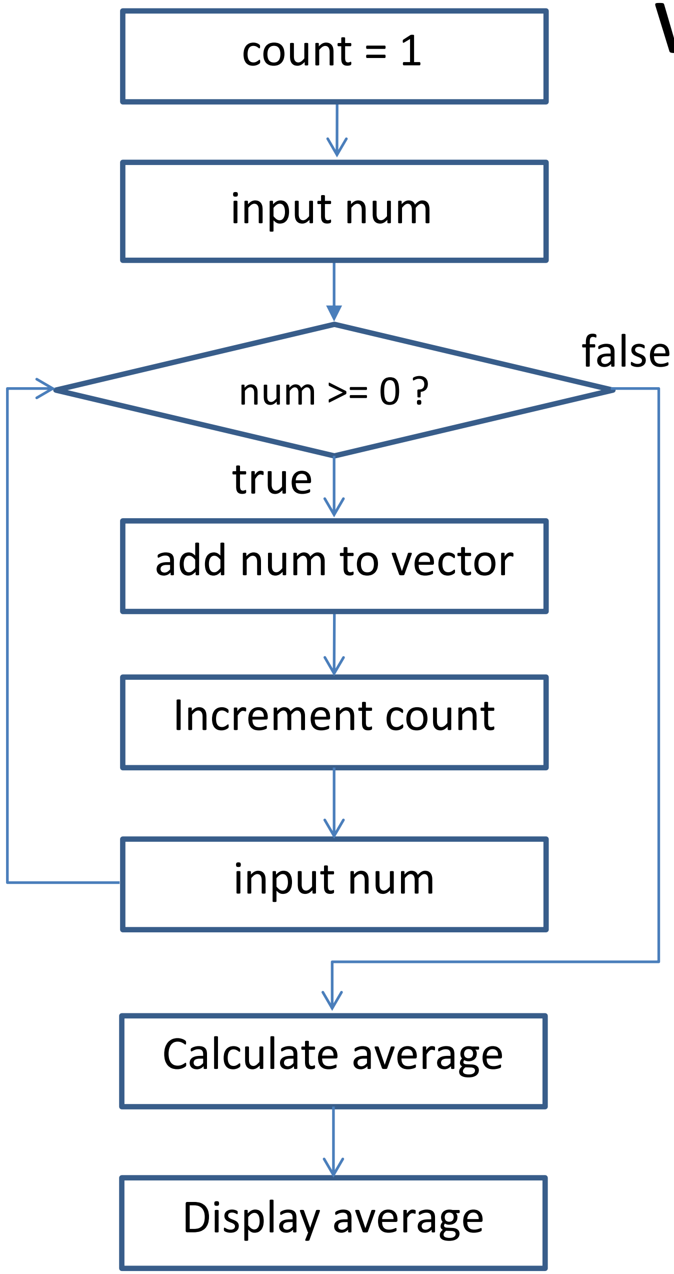
tic/toc

```
tic
Some code
.
.
.
toc
```

Times the code executed between the tic and toc commands

```
Elapsed time is 0.000039 seconds.
```

while Loops



Use while loop to input values and calculate an average

```
while condition==true
        Code block
end
```

```
count = 1;
num = input('Input first number: ');

% negative number terminates loop
while num  >= 0
    numbers(count) = num;
    count = count + 1;
    num = input('Input next number: ');
end
average = mean(numbers);
disp(['Average is ', num2str(average)]);
```

Exercise 9.2

Change the Projectiles script to loop until a valid input has been entered.

What sort of loop will be used?

While loop – continuing until condition is satisfied

Probably need to set angle to an invalid value to start so that the loop is executed at least once

Functions

Must have function declaration

Must be the same as the filename – funcName.m

Dummy input arguments

```
function [argout1, argout2..] = funcName( argin1, argin2,..)
% H1 comment line
% Other comments
Code Block
(return)
(end)
```

Dummy output arguments

Optional

Calling Functions

Called from a script or the command window by using the function name

```
distance = DistBetweenPoints( x1, y1, x2, y2 );
```

Use square brackets on the left for functions which return more than one value

```
[row, column] = find(4 )
```

The function .m file cannot be run on its own.

Note – parameters are passed by value. A copy of the arguments is passed to the function

Functions and Arrays

Arrays can be passed as parameters to and from functions

Ensure that any multiplication based operations within the function use the dot operator

Subfunctions

Can have several functions in one file:

- First function is primary function and has same name as filename

- Subfunctions follow in the same file and are only visible to other functions in that file (equivalent to private functions in C++)

- Help still provided by H1 line but accessed using >
```
>> help myFunction>mySubFunc
```

Anonymous Functions

Anonymous functions are local functions only available until the workspace is cleared

```
Fnhandle = @ (input arguments) functioncode;
```

Shows function handle being created

```
logfunc = @(x) log(x) + x

>> logfunc(2.3)

ans =

    3.1329
```

Can be saved in .mat file using load and save commands

Function Functions

Function functions take another function as input:

```
fplot( @sin, [-pi,pi] );
```
plot sin in range –pi to pi

Obtains function handle

```
fplot( logfunc, [1,100] )
```

Already function handle, doesn't need @

Persistent variables

Local variables are cleared when exit from functions
Persistent variables remain between function calls

declare variable, initialized to empty matrix

```
persistent count
% Check if count has been initialised
and set to 0 if not
if isempty( count )
   count = 0;
end
count = count + 1;
```

Persistent variable name must not be the same as any other in the workspace

Clear persistent variables from memory using `clear functions`

Exercise 10.2

```
>> poly1 = @(x) 3*x.^2 + 4*x +5;

>> plot([-10:10],poly1(-10:10) )

>> fplot(poly1, [0,20])

>> poly1(1:10)
ans =
    12    25    44    69   100   137   180   229   284   345
```

Tables

- Useful for heterogenous, column oriented or tabular data
- Variables can have different data types
- All columns must have the same number of rows
- Not restricted to column vectors (eg could have matrix but number of rows condition still applies

Create table from workspace data using the `table` function:

```
TableName = table(var1, var2, var3,… );
```

or by using `readtable` to load a data set:

```
TableName = readtable(‘data.dat’);
```

Indexing into Tables

Address column data using dot notation:

```
PatientData.Gender  % Accesses the whole column
```

Then normal indexing for the data type:

```
PatientData.Gender(10) % Accesses data in column
```

Access particular rows and columns using subscript notation:

```
PatientData(1:3, :);
ans =

     Gender      Age     Height     Weight
    ________     ___     ______     ______

    'Male'       38      71         176
    'Male'       43      69         163
    'Female'     38      64         131
```

Add, Remove and SaveTable Data

Extra data can be added by using a new column name and assigning it a data item with the correct number of rows:

```
PatientData.ID = (1:100)’;
```

Delete column using []

```
PatientData.ID = []
```

Table data can be written to a file using the `writetable` function:

```
writetable( PatientData, ‘PatientData.txt’);
```

Table Metadata

Table metadata is contained in table.Properties:

```
PatientData.Properties
ans =

             Description: ''
    VariableDescriptions: {}
           VariableUnits: {}
          DimensionNames: {'Row'  'Variable'}
                UserData: []
                RowNames: {}
           VariableNames: {'Gender'  'Age'  'Height'  'Weight'}
```

Edit the metadata using dot notation for the relevant property:

```
PatientData.Properties.VariableUnits{'Height'} = 'inches‘
PatientData.Properties.VariableUnits
ans =
    ''    ''    'inches'    ''
```

Plotting Data from Tables

Use dot notation to plot data for a whole column:

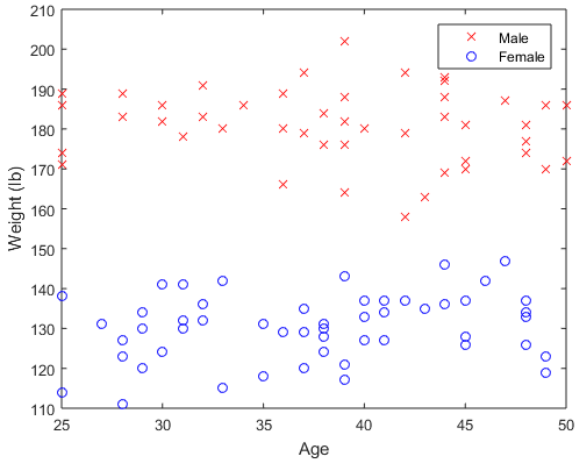
```
plot(PatientData.Age, PatientData.Weight, ‘xr’)
```

Logical indexing can be used to select data to plot. To plot data for just the males in the data set:

```
ind = strcmp( PatientData.Gender, 'Male');
plot(PatientData.Age(ind), PatientData.Weight(ind), 'xr')
```

Invert the selection to plot the data for females:

```
plot(PatientData.Age(~ind), PatientData.Weight(~ind), 'ob')
```



Importing Data

Import Wizard

Use the Import Data button
or right click on filename and select Import Data…

Data can be imported as row or column vectors where headers are present
Otherwise imported as matrix

Use Import Selection-> Generate Script or Generate Function to create code to reproduce import of the selected data

Also launch using

```
>> uiimport('SimonVega.jpg')
```

Cleaning Data

Bad or missing data will be imported as NaN

This can be removed using logical indexing

```
badData = isnan(Age);
CleanAge = Age(~badData);
```

Gives logical vector with value of 1 where data is NaN

Use ~ (not) so that good data has value of 1 in logical vector

Toolboxes

Signal Processing

- https://uk.mathworks.com/products/signal.html

Image Processing

- https://uk.mathworks.com/products/image.html

Optimisation

- https://uk.mathworks.com/products/optimization.html

Symbolic Maths

- https://uk.mathworks.com/products/symbolic.html

Data Acquisition

- https://uk.mathworks.com/products/daq.html

Control System

- https://uk.mathworks.com/help/control/index.html

Curve Fitting

- https://uk.mathworks.com/products/curvefitting.html

Statistics and Machine Learning

- https://uk.mathworks.com/products/statistics.html

More to Explore

Version Control using GIT – keep track of changes to your code

- https://uk.mathworks.com/help/matlab/matlab_prog/set-up-git-source-control.html

App Designer – for generating user interfaces

- https://uk.mathworks.com/help/matlab/gui-development.html?category=gui-development&s_tid=CRUX_topnav

Profiler – use to find bottlenecks in programs and speed up code

- https://uk.mathworks.com/help/matlab/matlab_prog/profiling-for-improving-performance.html

Testing Frameworks – use to set up automated tests for your code

- https://uk.mathworks.com/help/matlab/matlab-unit-test-framework.html