

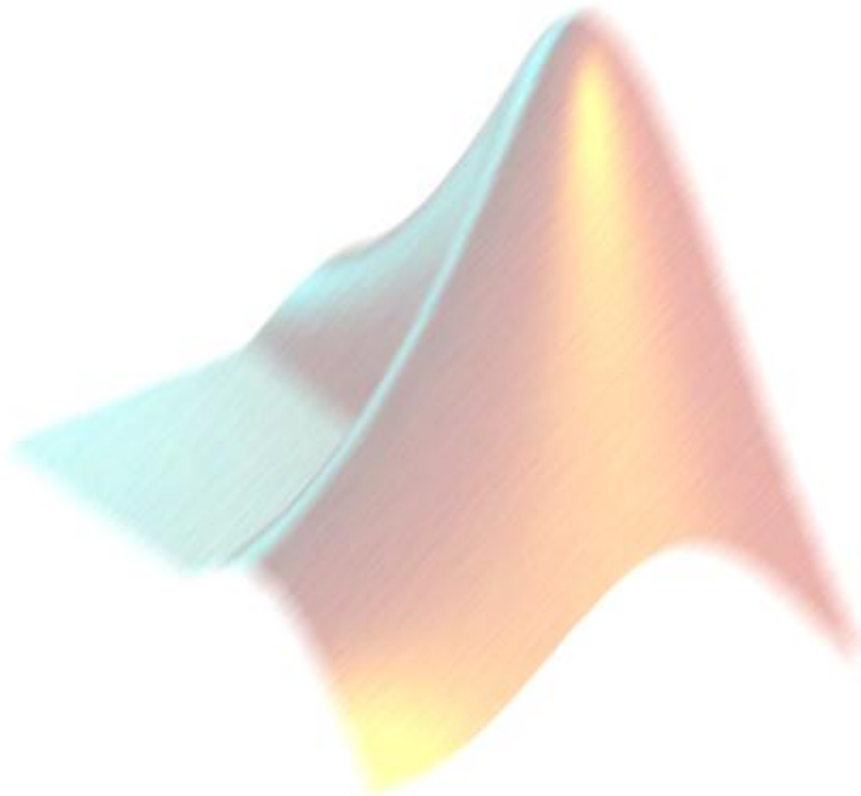
AP Induction Week Course

Introduction to Engineering Computation

Lecture Manual

MATLAB programming

Chapters 6 to 11



Department of Engineering Science

MATLAB Lecture Manual Contents

Chapter 6: 2D and 3D Arrays	2
Chapter 7: Graphics and Image Processing	19
Chapter 8: Strings.....	37
Chapter 9: Files	50
Chapter 10: Linear Equations and Linear Algebra	62
Chapter 11: Differential Equations and "Function" Functions	77

Chapter 6: 2D and 3D Arrays

As well as support for 1D arrays, MATLAB allows you to create and manipulate 2D and 3D arrays.

Learning outcomes

After working through this chapter, you should be able to:

- Explain what a 2D array is
- Create and manipulate 2D arrays
- Draw plots of 2D arrays
- Perform calculations with 2D arrays
- Manipulate 2D arrays using for loops
- Manipulate images via 3D arrays

2D arrays

Variables so far have been scalars (single value) and 1D arrays (lists of values).

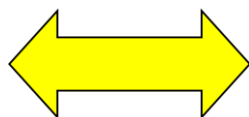
Some types of data are suited to being stored in 2D arrays. For example:

- data which corresponds to an underlying physical “grid”
- data from a table
- data representing the elements of a matrix

If a 1D array is like a *filing cabinet*, a 2D array is like a set of *cubby holes*

```
A = [1, 2, 3;  
     2, 4, 6;  
     -1, 0, 1]
```

```
A =  
  1   2   3  
  2   4   6  
 -1   0   1
```



A

$A(1,1) = 1$	$A(1,2) = 2$	$A(1,3) = 3$
$A(2,1) = 2$	$A(2,2) = 4$	$A(2,3) = 6$
$A(3,1) = -1$	$A(3,2) = 0$	$A(3,3) = 1$

Creating 2D arrays

Creating a 2D array is very similar to the way we create a 1D array. We enclose numbers within square brackets and values on the same row are separated by a comma or space.

A semi colon is used to indicate the start of a new row:

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

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

```
>> QuarterlyProd = [42, 52, 48, 47;  
    41, 48, 50, 42;  
    51, 38, 40, 41]
```

```
QuarterlyProd =  
    42    52    48    47  
    41    48    50    42  
    51    38    40    41
```

Accessing array elements

You can access a particular value in an array by using round brackets and two indices separated by a comma. These indices specify which row and column to look at:

For example, the array element in row 2 and column 3 of the QuarterlyProd array has the value 50.

```
>> QuarterlyProd(2,3)
```

```
ans =  
    50
```

If we want, we can change a particular array value:

```
>> QuarterlyProd(2,3) = 35
```

```
QuarterlyProd =  
    42    52    48    47  
    41    48    35    42  
    51    38    40    41
```

Extending arrays

You can add extra elements by creating them directly using round brackets, or by concatenating them (adding them onto the end). When extra elements are added MATLAB fills in any gaps with 0.

```
>> QuarterlyProd = [42, 52, 48, 47;  
    41, 48, 50, 42;  
    51, 38, 40, 41]
```

```
QuarterlyProd =
```

```
    42    52    48    47  
    41    48    50    42  
    51    38    40    41
```

```
>> QuarterlyProd(4, 1) = 45
```

```
QuarterlyProd =
```

```
    42    52    48    47  
    41    48    50    42  
    51    38    40    41  
    45     0     0     0
```

When concatenating you need to make sure the dimensions of the arrays being used are compatible.

<pre>>> A = [8, 9; 1 2] A = 8 9 1 2 >> B = [4 5] B = 4 5 >> C = [3; 5] C = 3 5</pre>	<pre>>> D = [A; B] D = 8 9 1 2 4 5 >> E = [A, C] E = 8 9 3 1 2 5 >> F = [A, C; B, 12] F = 8 9 3 1 2 5 4 5 12</pre>
---	---

2D array functions

Standard mathematical functions can be applied to 2D arrays too.

```
>> x = [1, 2, 3; 4, 5, 6];
>> y = sin(x)
y =
    0.8415    0.9093    0.1411
   -0.7568   -0.9589   -0.2794
```

Special array functions

The **size** function returns the number of rows and columns in an array. The syntax is:

```
[m, n] = size(A)
```

Where m = number of rows, n = number of columns

The transpose operator `'` swaps the rows and columns in an array.

Be careful as it is easy to miss the transpose operator, since it is so small.

```
>> A = [1 2 3;
        4 5 6];
>> [m,n] = size(A)

m =
    2
n =
    3
>> B = A'

B =
    1    4
    2    5
    3    6
```

Automatic 2D arrays

There are a number of useful functions for creating 2D arrays: `zeros`, `ones`, `rand`, `eye` and `meshgrid`.

The `zeros` function takes a number of rows and columns and creates an array of the specified size filled with zeros. The `ones` function is similar but creates an array full of ones. The `rand` function is also similar but fills the array with random values between 0 and 1. The `eye` function takes a number of rows and creates a square array with ones on the diagonal and zeros everywhere else. The `eye` function is handy for generating identity matrices.

```

>> zeros(2, 4)
ans =

    0    0    0    0
    0    0    0    0
>> ones(3, 2)
ans =

    1    1
    1    1
    1    1

>> eye(3)
ans =

    1    0    0
    0    1    0
    0    0    1

```

The `meshgrid` command will be covered in later chapters.

Drawing 2D arrays

The data in a 2D array can be represented as a surface in 3D by using the **surf** command.

```

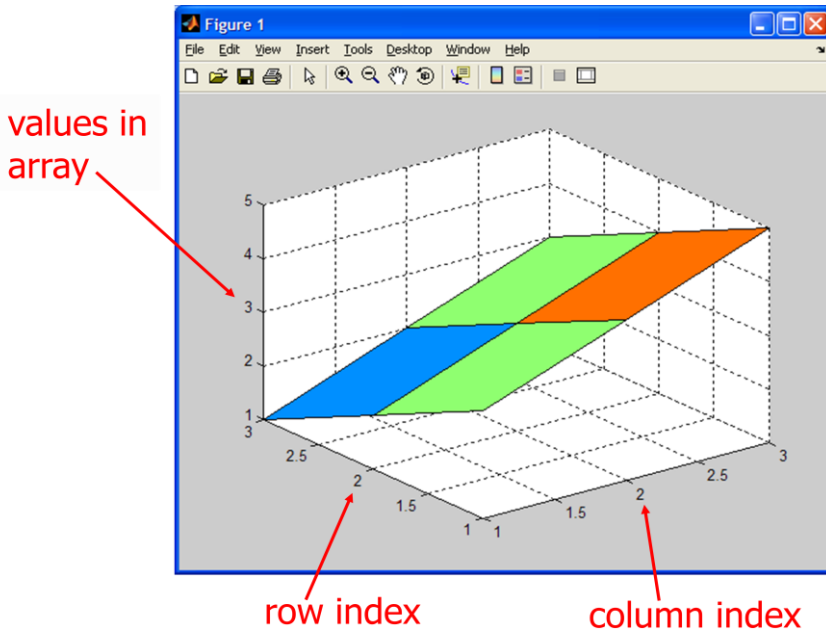
>> M = [3 4 5;
2 3 4;
1 2 3]

M =

    3    4    5
    2    3    4
    1    2    3

>> surf(M)

```



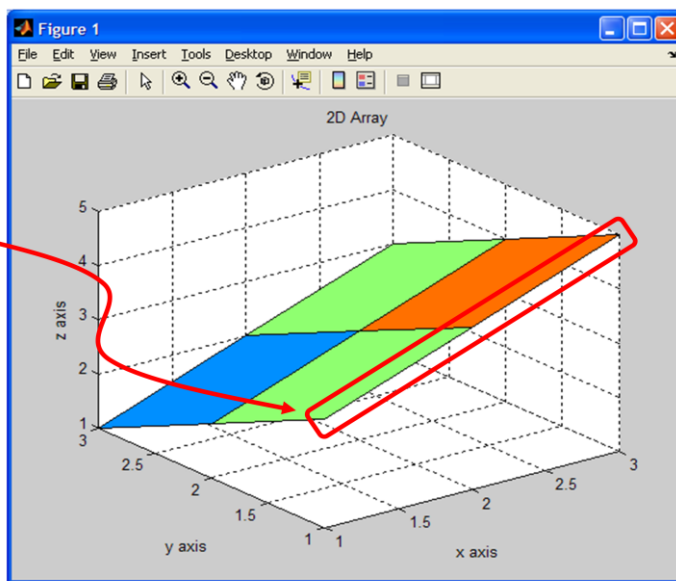
We can add labels in with the `xlabel`, `ylabel` and `zlabel` commands. You might expect the first index (the row index) to correspond to the x axis, but by default it actually corresponds to the y axis.

```
>> M = [3 4 5;
2 3 4;
1 2 3]

M =

    3     4     5
    2     3     4
    1     2     3

>> surf(M)
>> xlabel('x axis')
>> ylabel('y axis')
>> zlabel('z axis')
>> title('2D Array')
```



In the example above $M(1,3)$ is equal to 5 and presents the value at the point (3,1).

Arithmetic with 2D arrays

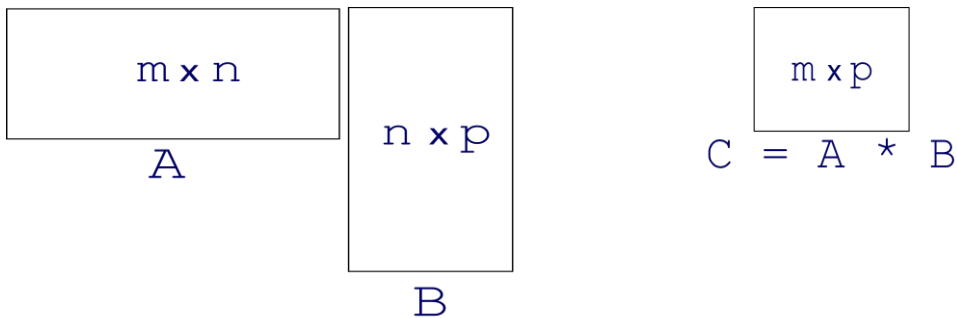
Two 2D arrays can be added or subtracted using the + and - operators, as long as the arrays have same size. This is identical to matrix addition or subtraction.

Hint Use the **size** command to find out how big an array is, or check in the workspace window

Two 2D arrays can be multiplied with the * operator. This performs matrix multiplication. As with all matrix multiplication, the first array must have same number of columns as the second array has rows.

You can check how many rows or columns an array has with the **size** command:

- **size(A, 1)** gives number of rows of A
- **size(A, 2)** gives number of columns of A



```
>> A = [3 1 0;  
        1 -2 4];  
>> B = [2;  
        4;  
        1];  
>> C = A * B  
  
C =  
  
    10  
    -2
```

$$\begin{aligned} C &= A \times B \\ &= \begin{bmatrix} (3 \times 2) + (1 \times 4) + (0 \times 1) \\ (1 \times 2) + (-2 \times 4) + (4 \times 1) \end{bmatrix} \\ &= \begin{bmatrix} 10 \\ -2 \end{bmatrix} \end{aligned}$$

Element by element operations

In mathematically based work matrix multiplication is very useful. However in some applications we want to perform an element-wise multiplication, ie we want to multiply each element in the first array by the corresponding element in the second array. For this to work the two arrays must be the same size.

To perform multiplication element-wise use a `.` before the operator

<pre>>> A = [3 1 0; 1 -2 4]; >> B = [4 2 -1; 0 1 3]; >> C = A .* B C = 12 2 0 0 -2 12</pre>	$C = A .* B$ $= \begin{bmatrix} (3 \times 4) & (1 \times 2) & (0 \times -1) \\ (1 \times 0) & (-2 \times 1) & (4 \times 3) \end{bmatrix}$ $= \begin{bmatrix} 12 & 2 & 0 \\ 0 & -2 & 12 \end{bmatrix}$
--	---

The dot operator can also be applied with other mathematical operations, just as we did with 1D arrays:

- Using `.^ 2` squares elements in the array term by term instead of multiplying the whole array by itself
- Using `./` divides the array element by element

```
>> denom = [2, 3, 4, 5, 6];
>> numer = [1, 2, 3, 4, 5];
>> fracs = numer ./ denom
```

```
fracs =

    0.5000    0.6667    0.7500    0.8000    0.8333
```

Subranges

We can select any submatrix using 1D arrays of indices. This is similar to taking a slice of an array.

<pre>>> A = [1 4 5 6; 8 3 2 8; 0 6 7 9]; >> B = A(2:3, 2:4) B = 3 2 8 6 7 9</pre>	$A = \begin{pmatrix} 1 & 4 & 5 & 6 \\ 8 & 3 & 2 & 8 \\ 0 & 6 & 7 & 9 \end{pmatrix}$
<pre>>> C = A([2 1], [1 3 4]) C = 8 2 8 1 5 6</pre>	$A = \begin{pmatrix} 1 & 4 & 5 & 6 \\ 8 & 3 & 2 & 8 \\ 0 & 6 & 7 & 9 \end{pmatrix}$

Colon operator

Using a colon `:` instead of an index array refers to ALL rows or columns of the array

<pre>>> A = [1 4 5 6; 8 3 2 8; 0 6 7 9]; >> B = A(2, :)</pre> <p>B =</p> <pre> 8 3 2 8</pre>	<pre>>> C = A(:, 2)</pre> <p>C =</p> <pre> 4 3 6</pre> <pre>>> D = A(1:2, :)</pre> <p>D =</p> <pre> 1 4 5 6 8 3 2 8</pre>
--	---

2D arrays and for loops

We have seen how to use a *for loop* to loop through the contents of a 1D array. In order to loop through all elements of a 2D array we can use two for loops, with one inside the other. This is called a nested for loop.

The flow chart for a nested for loop is shown overleaf:

For loop flow chart

We will draw this in class.

Editing a greyscale image

A greyscale image is made up of lots of tiny pixels, with each pixel having an intensity value representing how light or dark that point on the image is. We can use intensity values ranging from 0 to 1 to represent different shades of grey (0 being black and 1 being white).

The intensity value for a pixel in row m and column n of an image can then be stored in a 2D array. To make a pixel in row 2, column 3 of an image a nice shade of grey we could use:

```
myPic(2,3) = 0.5
```

To make a pixel in row 4, column 5 white we would use

```
myPic(4,5) = 1;
```

It would take a long time to make an interesting picture setting each pixel value in turn, so we will use a nested for loop to create an image with the upper left corner black and the lower right corner white:

```
% Create a rectangular image 100x200 pixels in size
% the upper left corner is black
% the lower right corner is white

% cycle through each row
for i = 1:100
    % cycle through each column
    for j = 1:200
        % set the pixel value for row i, column j
        image(i,j) = (i+j)/300;
    end;
end;

% display the image
imshow(image)
% write the image to a file
imwrite(image, 'greyRectangle.jpg');
```

This produces the following image:



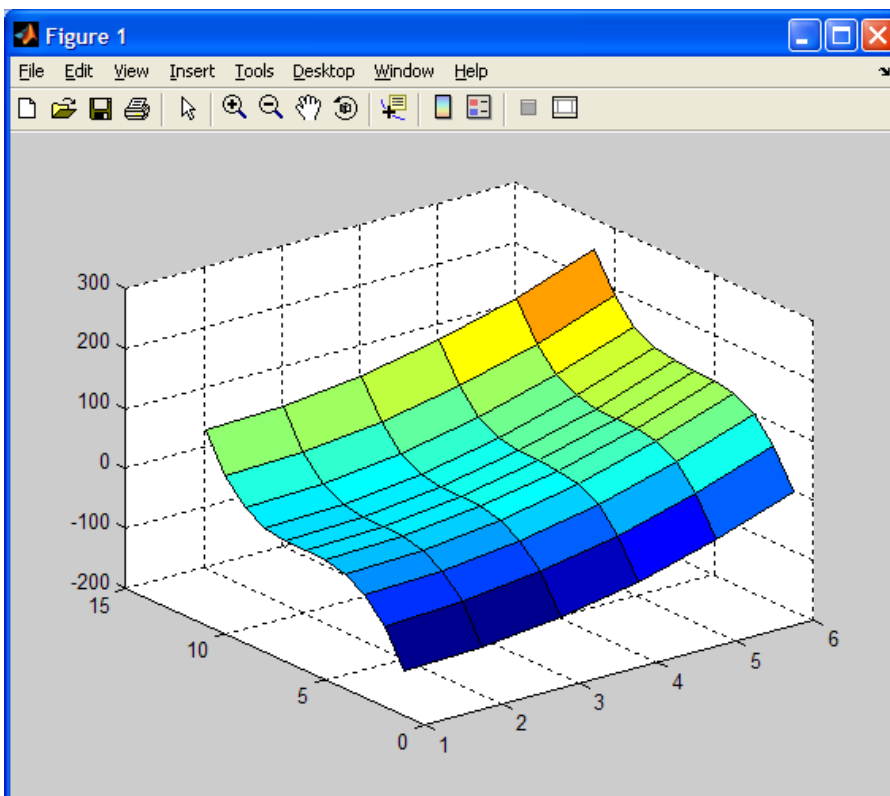
Plotting 3D polynomials

We will use a nested for loop to plot $f(x,y)=5x^2+y^3$, $x \in [0,5], y \in [-5,5]$

```
% plot f(x,y) = 5x^2 + y^3  
  
x = 0:5;  
y = -5:5;  
for i = 1:length(x),  
    for j = 1:length(y),  
        Z(j, i) = 5 * x(i)^2 + y(j)^3;  
    end;  
end;  
  
surf(Z)
```

Note that we have used i as the second index for the 2D array, since the second index corresponds to the x axis on the surface plot.

The result of running our for loop code is a plot of the 3D polynomial.



3D arrays and image processing

Arrays in MATLAB can be three dimensional. If a 2D array is like a cubby hole a 3D array is like layering several cubby holes on top of each other. This means to identify a single item in a 3D array we use three indices. You can think of these as a row number, a column number and a "layer" (ie which cubby hole to use).

One use of 3D arrays is to describe values for data that vary over three dimensions (eg temperature in a room). The three indices describe which point we are dealing with (remember they must be whole numbers though)

3D arrays are very useful for graphics applications, as colour images can be nicely represented using 3D arrays. A 2D image is made up of lots of pixels. Each pixel is a tiny square of colour. In the RGB (red-green-blue) colour scheme the pixel colour is stored by recording three colour intensities for the red, green and blue intensities. This means every pixel in a 2D image has three separate values (one for red, one for green and one for blue).

8 bit image formats use a value between 0 and 255 to represent each intensity (0 means none of that colour, while 255 means lots of it). With just 3 colour values (each between 0 and 255) we can represent over 16 million different colours.

To work with images we can store all the colour intensities for all the pixels in a 3D array. Our three indices for the array are: the row, the column and the colour (1 = red, 2 = green, 3 = blue).

For example to set the amount of red for a pixel in row 2, column 3 of our image we would use

```
% some red
myPic(2,3,1) = 128
```

We could also set the amount of blue and green

```
% lots of green
myPic(2,3,2) = 255
```

```
% no blue
myPic(2,3,3) = 0
```

Red and green mix to make yellow so this pixel will be a greeny shade of yellow.

To make a small 30x40 red image we could do the following:

```
% create a 3D array with 30 rows, 40 columns and 3 "layers" (one for
each colour)
myPic = zeros(30,40,3)
```

```
% set all values in layer 1 (the colour red) to be 255
myPic(:, :, 1) = 255;
% we leave all values in layer 2 and 3 as zero
```

We can display our image using the **image** or **imshow** function:

```
imshow(myPic)
```

If we wanted to add in a row of bright yellow we can do the following

```
% change green intensity to 255 for all pixels in row 1.  
myPic(1, :, 2) = 255;
```

All pixels in the first row now have lots of red, lots of green but no blue. Mixing red and green creates yellow, so this row will be yellow.

To read an existing image into MATLAB we use the `imread` function, which takes as input the filename of an image enclosed in single quotes:

```
myPicture = imread('photo.jpg')
```

This command will create a 3D array called `myPicture` which contains the image intensity values for the image contained in the named file. We can determine the number of rows and columns in the image by using the `size` command.

```
size(myPicture)
```

```
ans =  
    250   166    3
```

The 3D array has 250 rows, 166 columns and 3 layers (1 layer for each colour).

We can use nested for loops to loop through all colour values for all pixels and modify them. The resulting image can then be written out using the `imwrite` command. The following program will create a negative of a photo, by inverting the colour.

```
% create negative image from a photo  
  
% read in image  
myPicture = imread('photo.jpg')  
  
[rows,cols,colours] = size(myPicture);  
  
for i=1:rows  
    for j=1:cols  
        for k=1:3  
            myPicture(i,j,k) = 255 - myPicture(i,j,k);  
        end  
    end  
end  
  
imshow(myPicture);  
imwrite(myPicture, 'negative.jpg');
```




Chapter 6 Summary Programs

We can now write programs that use 2D and 3D arrays.

Solve a simple matrix problem?

For example, the following program reads in a greyscale image, flips the image upside down and then adds a black border to it.

```
% flip greyscale image upside down and add a black border

% read in image
myPicture = imread('greyscalePhoto.jpg');

[rows,cols] = size(myPicture);

% flip image by swapping rows. We only need to loop through the
% first half of the rows in order to swap them all.
% we want to swap:
% row 1 with the last row,
% row 2 with last row - 1,
% row 3 with last row - 2,
% row i with last row + 1 - i

for i=1:floor(rows/2)
    for j=1:cols
        % store the old pixel value from the top half
        % before we over write it
        oldpixel = myPicture(i,j);
        % replace the pixel value on the top half with one from
        % the bottom half of the image
        myPicture(i,j) = myPicture(rows + 1 - i,j);
        % set the pixel of the bottom half to what was in the top
        myPicture(rows+1-i,j) = oldpixel;
    end
end

% add border
myPicture(:,1:3) = 0;
myPicture(:,(cols-2):cols) = 0;
myPicture(1:3,:) = 0;
myPicture((rows-2):rows,:) = 0;

imshow(myPicture);
imwrite(myPicture, 'flipped.jpg');
```

An example of this program running is given below:



Chapter 7: Graphics and Image Processing

You have already encountered the plot function, for creating simple xy plots. MATLAB provides a large number of other graphics functions for more advanced plotting. It is also possible to use MATLAB to manipulate images, by reading the colour values into a 3D array and then changing the colour values.

Learning outcomes

After working through this chapter, you should be able to:

- Label your plots
- Create different types of 1D data plots (log graphs, bar graphs and polar plots)
- Control line types, axis types and colours on 1D plots
- Create several figures at the same time
- Plot several sets of data on the same graph
- Create subplots
- Create different types of 2D data plots (surface maps, contour plots and quiver plots)
- Make MATLAB movies
- Manipulate RGB images via 3D arrays

Labelling plots

The purpose of a plot is to communicate information. That information cannot be communicated effectively **unless** the plot is labelled. It is **very important** that you label your plots. Labels can be added by using the insert menu on a figure window but it is often more convenient to use the MATLAB labelling functions.

Title

The title function allows us to create a title for our plot. Make sure your title is meaningful and describes what is being plotted. The title command takes as input some words enclosed in single quotes and writes these out as the plot title.

```
title('Vertical movement of an anchored boat');
```

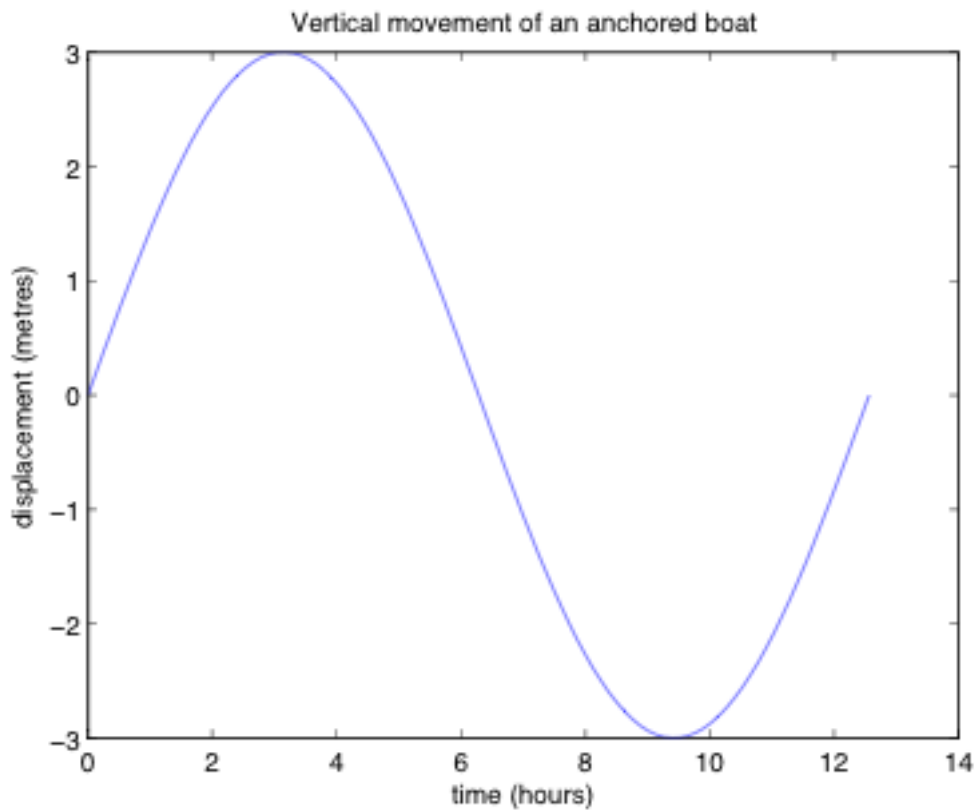
Labels for the x and y axes

You should label BOTH axes, indicating the name of the quantity being plotted and the units used. The `xlabel` and `ylabel` functions are used to label the x and y axes. As with the title command, they take as input some words enclosed in single quotes and write these out as the labels.

```
xlabel('time (hours)')  
ylabel('displacement (metres)');
```

Note that the title, `xlabel` and `ylabel` commands need to be called **after** the plot function.

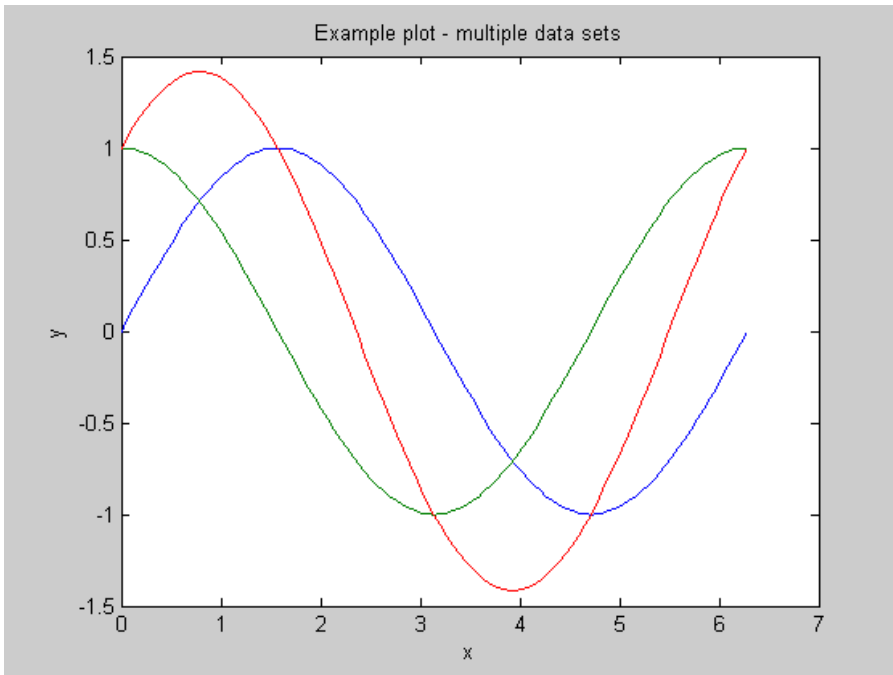
```
x = linspace(0,4*pi,100);  
y = 3 * sin(0.5*x);  
plot(x,y);  
title('Vertical movement of an anchored boat');  
xlabel('time (hours)')  
ylabel('displacement (metres)');
```



Plotting multiple data sets

The plot command can be used to plot several lines on the same graph, eg:

```
x = 0 : 2*pi/100 : 2*pi;  
y1 = sin(x);  
y2 = cos(x);  
y3 = sin(x) + cos(x);  
plot(x,y1,x,y2,x,y3)  
xlabel('x')  
ylabel('y')  
title('Example plot - multiple data sets')
```



An alternative method to plotting several sets of data on the same plot is to use the hold command.

The `hold` command allows you to **hold on** to your current plot, so that subsequent plot commands are performed on the current plot, rather than creating a new plot.

```
x = 0 : 2*pi/100 : 2*pi;
y1 = sin(x);
y2 = cos(x);
y3 = sin(x) + cos(x);
plot(x,y1)
hold on
plot(x,y2)
plot(x,y3)
xlabel('x')
ylabel('y')
title('Example plot - multiple data sets')
```

Note that if we do NOT use the hold on command, each time the plot function is called any previous plot is thrown away and replaced by the latest one.

To stop holding on to a plot use the `hold off` command.

Controlling 1D plots

Line colours, types and symbols.

By default 1D plots use a solid blue line. You can specify other colours and line styles, if you prefer.

For many data sets it makes more sense to plot individual points rather than a line. You can specify to plot a symbol at each data point. Generally measured data is plotted using individual points while functions are plotted using a line. Sometimes it is a good idea to do both.

To control line colours, types and symbols we pass in an optional third argument to the plot command. This argument is a sequence of control characters enclosed in single quotes.

Here are a few examples of how to specify line colours, types and plot symbols:

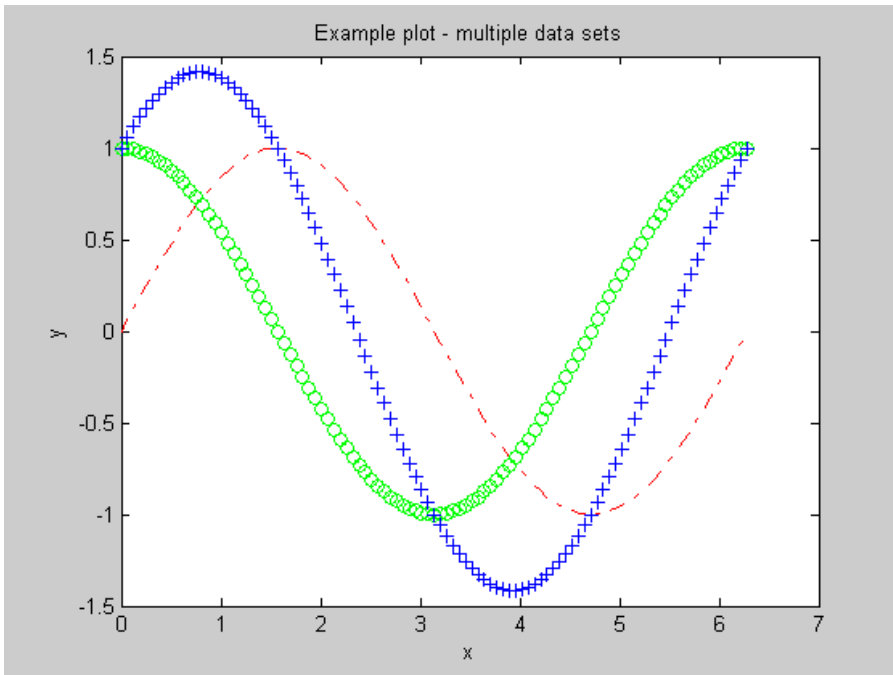
<code>plot(x,y,'r--')</code>	Plot the data using a red dashed line.
<code>plot(x,y,'go')</code>	Plot the data using just green circles at each point (and no line)
<code>plot(x,y,'c:+')</code>	Plot the data using a dotted cyan line and a plus symbol at each data point

A summary of some of the control characters is given in the table below:

Colour	Symbol	Line style
r red	. point	- solid
g green	o circle	: dotted
b blue	x cross	-. dashdot
c cyan	+ plus	-- dashed
m magenta	* star	
y yellow		
k black		

You can specify an element from any or all of these three columns. The order is not important but you cannot have more than one element from each column.

```
x = 0 : 2*pi/100 : 2*pi;
y1 = sin(x);
y2 = cos(x);
y3 = sin(x) + cos(x);
plot(x,y1,'r-.',x,y2,'go',x,y3,'b+')
xlabel('x')
ylabel('y')
title('Example plot - multiple data sets')
```



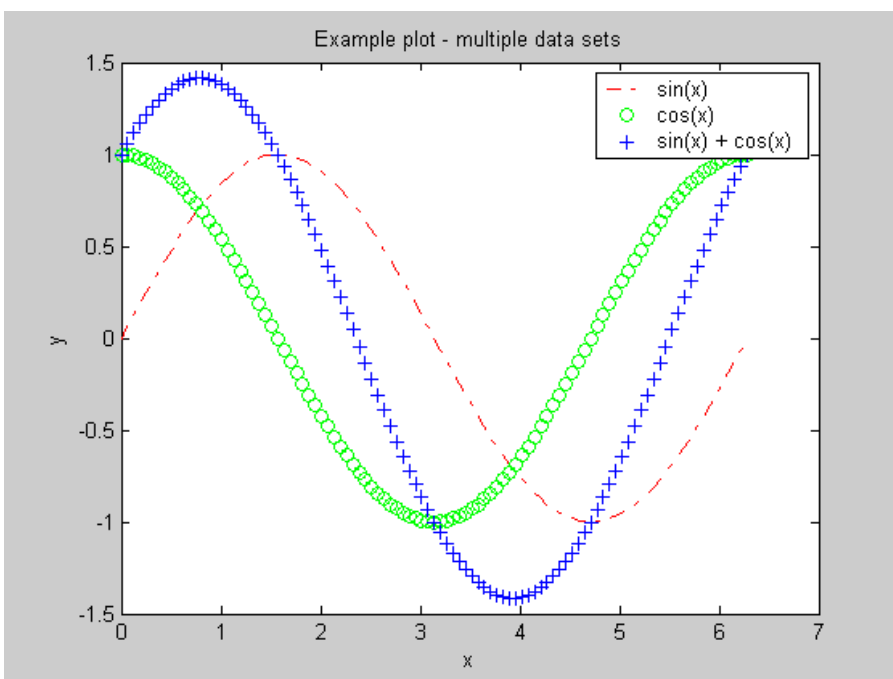
For full details on all available colours, symbols and line styles see the MATLAB help for the plot function.

Legends

If more than one set of data has been plotted on the same graph it is important to add a legend, so that we can tell what each line represents. It is a good idea to use different line types as well as colours, since that makes it easier to distinguish each line if the plot is printed out in black and white.

This can be done with the legend command. Once it has been added you can move the position of the legend on the figure with the mouse.

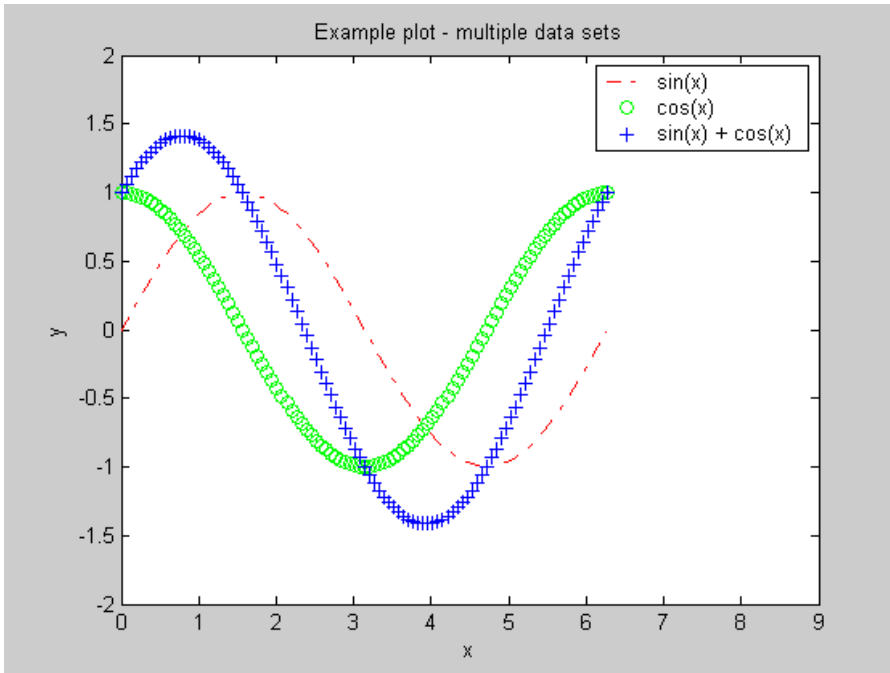
```
legend('sin(x)', 'cos(x)', 'sin(x) + cos(x)')
```



Axes

MATLAB will automatically determine the maximum and minimum values for the axes. To override these use the `axis` function. The `axis` function takes as input a four element array which specifies the axes limits in the following order: `[xmin, xmax, ymin, ymax]`. The x axis will then range from `xmin` to `xmax`, while the y axis will range from `ymin` to `ymax`.

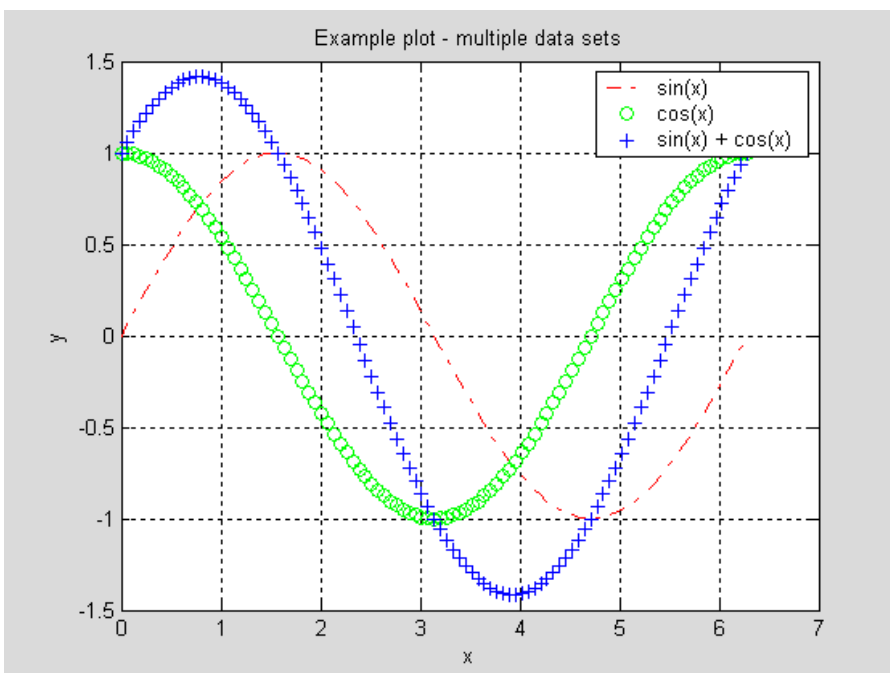
```
axis([ 0, 9, -2, 2 ])
```



Grid lines

If you like grid lines on your plots you can add them using the `grid on` command.

```
grid on
```



Creating additional figures

What happens if you enter the following?

```
x = 0 : 2*pi/100 : 2*pi;
y1 = sin(x);
y2 = cos(x);

plot(x,y1)
title('Example plot #1')

plot(x,y2)
title('Example plot #2')
```

As we have NOT used the hold on command, MATLAB will create a figure for the the first plot and then it will overwrite the first plot with the second plot. We end up with only one figure, containing a plot of $y=\cos(x)$.

If we wanted to view the two plots at the same time we can place each one in its own figure. We do this by creating an additional figure window before making the second plot.

```
plot(x,y1)
title('Example plot #1')

figure
plot(x,y2)
title('Example plot #2')
```

Note that the second figure may be sitting on top of the first, so you may need to move it to the see the figure underneath. MATLAB automatically numbers each new figure with a new number.

An even better option than just using the figure command is to create numbered figure windows for each plot:

```
figure(1)
plot(x,y1)
title('Example plot #1')

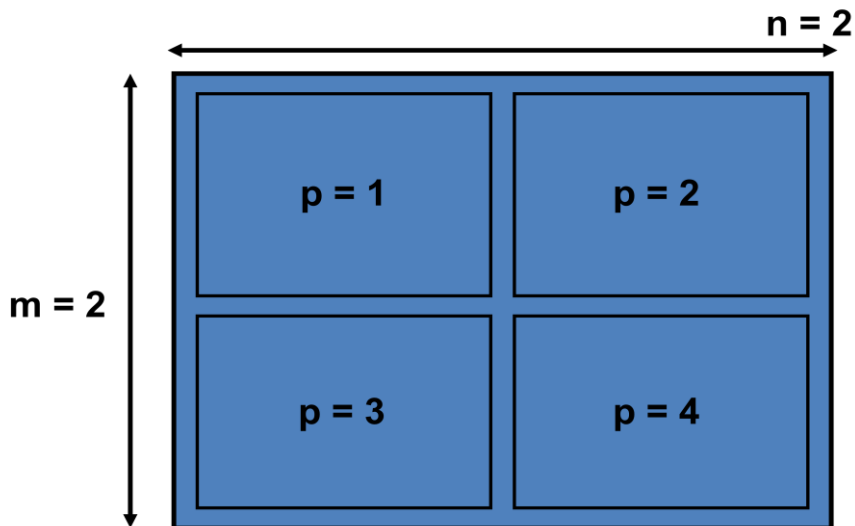
figure(2)
plot(x,y2)
title('Example plot #2')
```

Explicitly numbering figures makes it easier to tell what data we are dealing with in any given figure.

Subplots

Sometimes it makes sense to present data as a set of plots contained inside the same figure, this can be done with the subplot (m, n, p) command.

The subplot command specifies the number of rows (m) and the number of columns (n) in the subplot. The plot number (p) indicates what position to plot the data at.

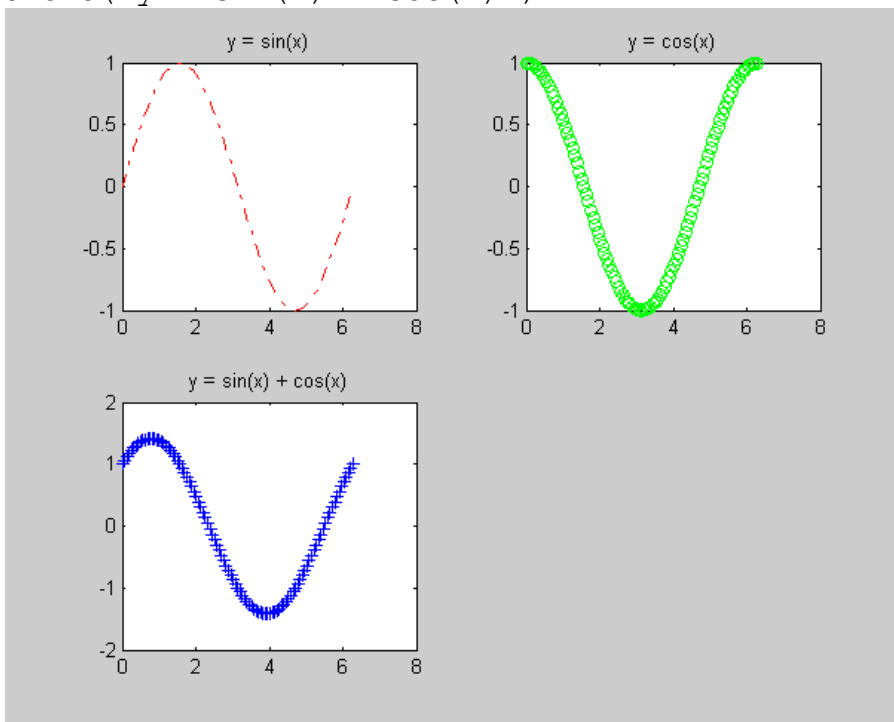


Note that you do not have to use up every position.

```
subplot(2,2,1)
plot(x,y1,'r-.')
title('y = sin(x)')
```

```
subplot(2,2,2)
plot(x,y2,'go')
title('y = cos(x)')
```

```
subplot(2,2,3)
plot(x,y3,'b+')
title('y = sin(x) + cos(x)')
```



You may want to resize the figure window with the mouse if you are using subplots.

Other Types of 1D plots

As well as basic xy plots MATLAB supports a number of other common plot types.

Log graphs

You can create line graphs with log scaling on either or both axes by using the commands `semilogx`, `semilogy` and `loglog`. These commands use the same syntax as the `plot` function.

`semilogx(x,y)`

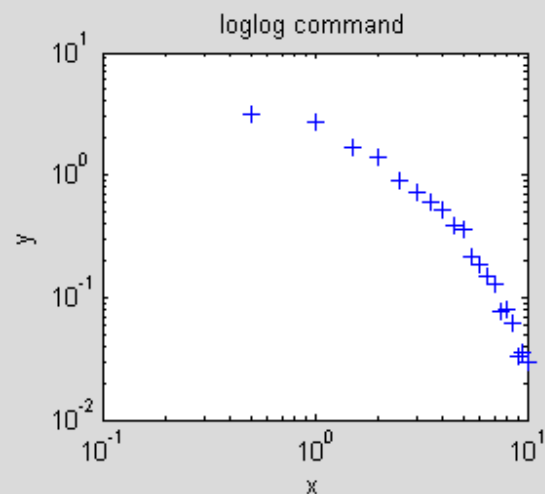
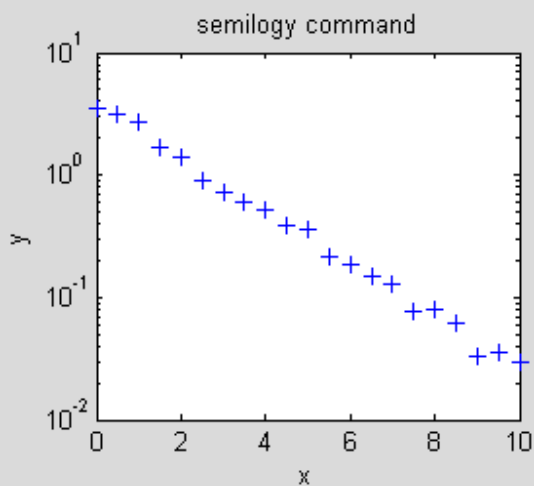
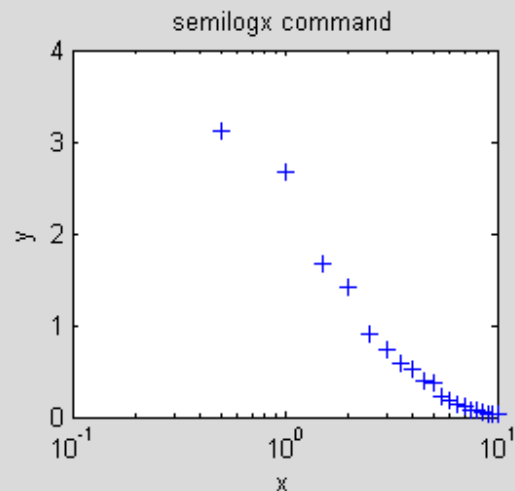
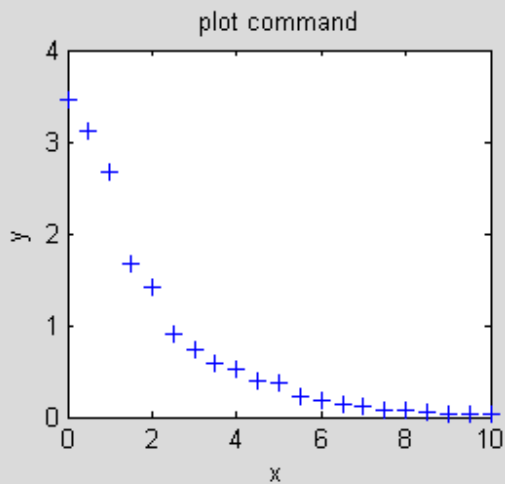
`semilogy(x,y)`

`loglog(x,y)`

Log graphs can be useful when you are deciding on what kind of model to fit to a data set (eg power model or exponential model).

$$y = mx + c$$

$$y = m \log(cx)$$



$$y = ce^{mx}$$

$$y = c x^m$$

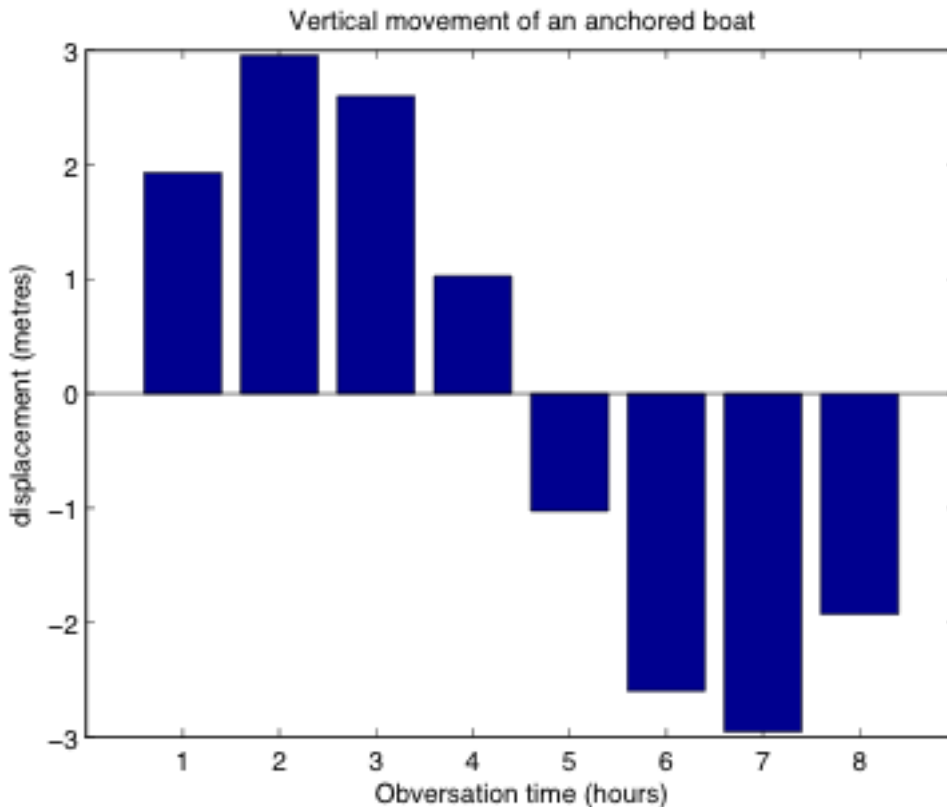
Bar graphs

You can create bar graphs with the `bar` function. The syntax is the same as `plot` function:

```
bar(x, y)
```

For each `x` value a bar is drawn with the corresponding `y` value used as the height. You must make sure that there are no duplicates in the `x` array.

Here is a plot of some hourly observations of an anchored boat moving with the tide.



Polar graphs

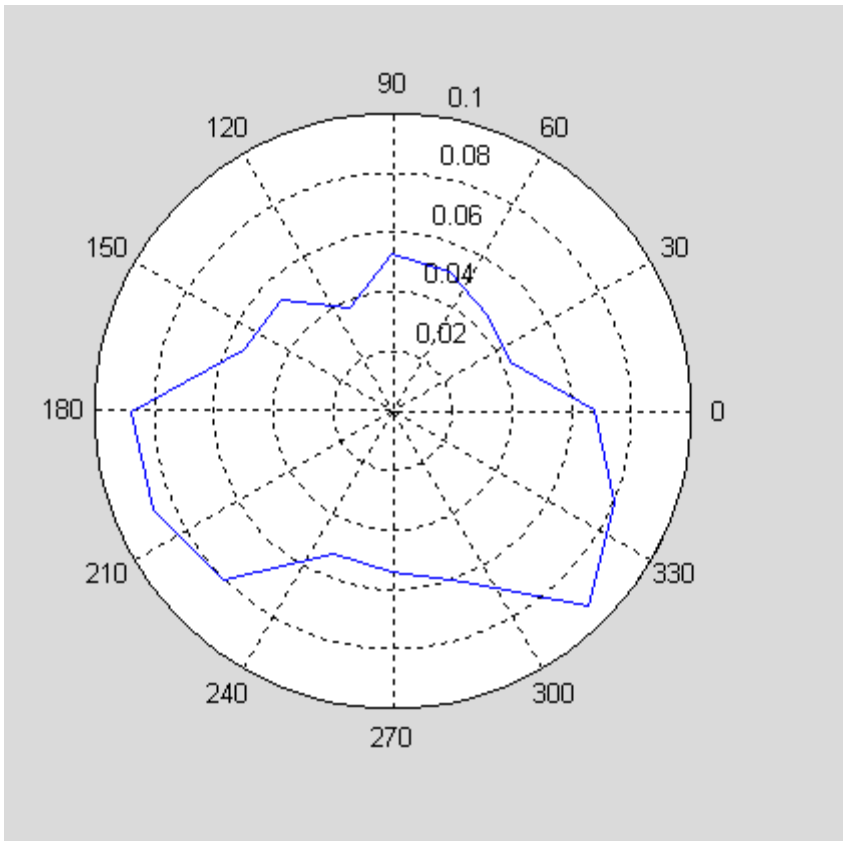
In some applications we need to depict data that has an angle dependence. For example, if you were designing navigational software for a yacht you would need to know how often the wind blows from each direction.

A polar plot is one way to depict angle dependent data. The syntax of the MATLAB function for producing polar plots is:

```
polar(angleData, plotData).
```

The `angleData` is an array of angles (in radians). The `plotData` is an array containing the corresponding data values for each angle.

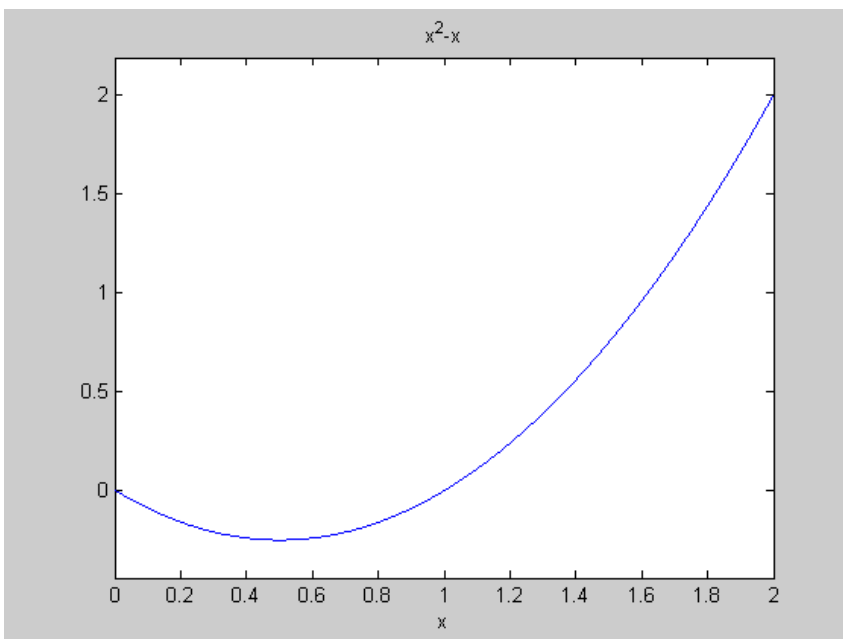
Here is a plot of wind directions in Evansville, IN.



Function plotting

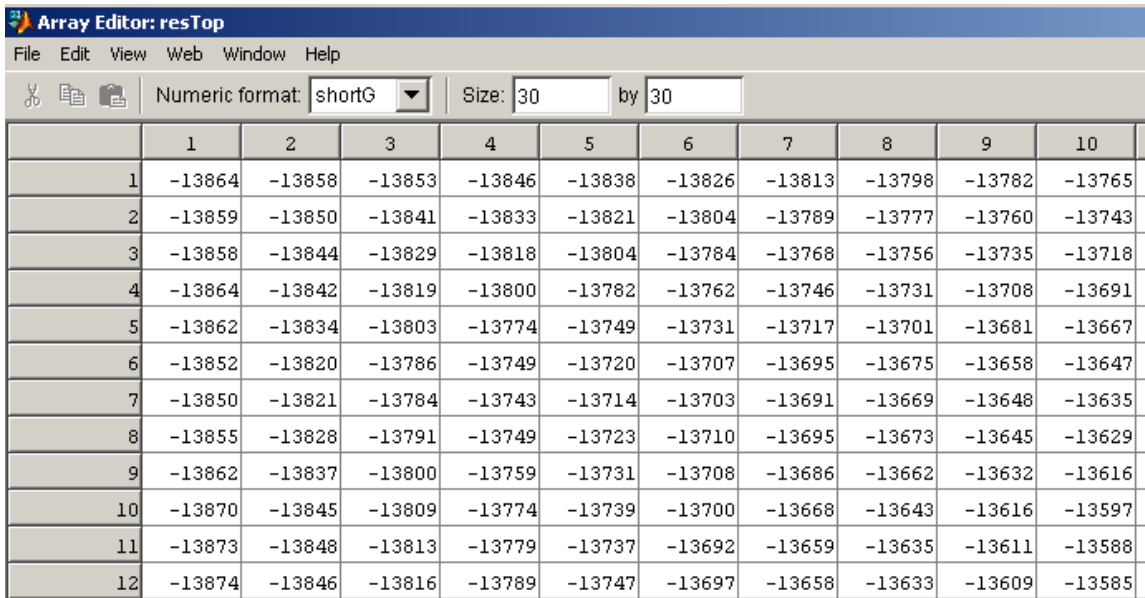
•Note that you can also plot functions directly (instead of building arrays with the function values and plotting them). To do so use the **ezplot** command. The ezplot function takes a function in single quotes and a two element array that specifies the domain of the function:

```
ezplot('x^2-x', [0,2])
```



Plotting 2D arrays

Suppose we have a 2D array containing the depths to the top of an oil reservoir.



The screenshot shows the MATLAB Array Editor window titled "Array Editor: resTop". The window has a menu bar with "File", "Edit", "View", "Web", "Window", and "Help". Below the menu bar is a toolbar with icons for copy, paste, and delete. The "Numeric format" is set to "shortG" and the "Size" is "30 by 30". The main area displays a 12x10 grid of numerical values representing depths.

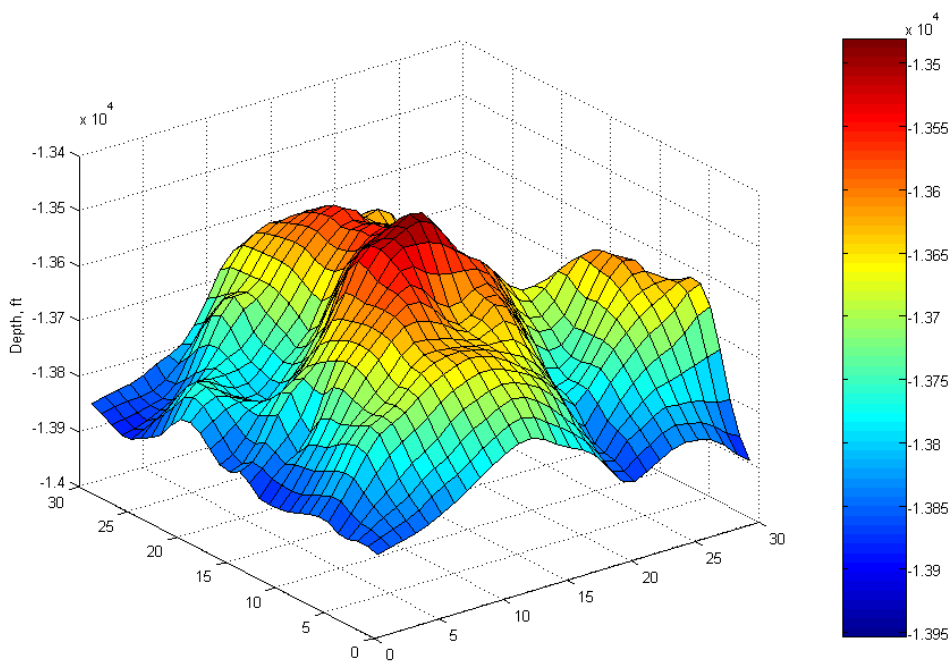
	1	2	3	4	5	6	7	8	9	10
1	-13864	-13858	-13853	-13846	-13838	-13826	-13813	-13798	-13782	-13765
2	-13859	-13850	-13841	-13833	-13821	-13804	-13789	-13777	-13760	-13743
3	-13858	-13844	-13829	-13818	-13804	-13784	-13768	-13756	-13735	-13718
4	-13864	-13842	-13819	-13800	-13782	-13762	-13746	-13731	-13708	-13691
5	-13862	-13834	-13803	-13774	-13749	-13731	-13717	-13701	-13681	-13667
6	-13852	-13820	-13786	-13749	-13720	-13707	-13695	-13675	-13658	-13647
7	-13850	-13821	-13784	-13743	-13714	-13703	-13691	-13669	-13648	-13635
8	-13855	-13828	-13791	-13749	-13723	-13710	-13695	-13673	-13645	-13629
9	-13862	-13837	-13800	-13759	-13731	-13708	-13686	-13662	-13632	-13616
10	-13870	-13845	-13809	-13774	-13739	-13700	-13668	-13643	-13616	-13597
11	-13873	-13848	-13813	-13779	-13737	-13692	-13659	-13635	-13611	-13588
12	-13874	-13846	-13816	-13789	-13747	-13697	-13658	-13633	-13609	-13585

It would be more useful to visualize this data in 2 or 3 dimensions.

Surface plots

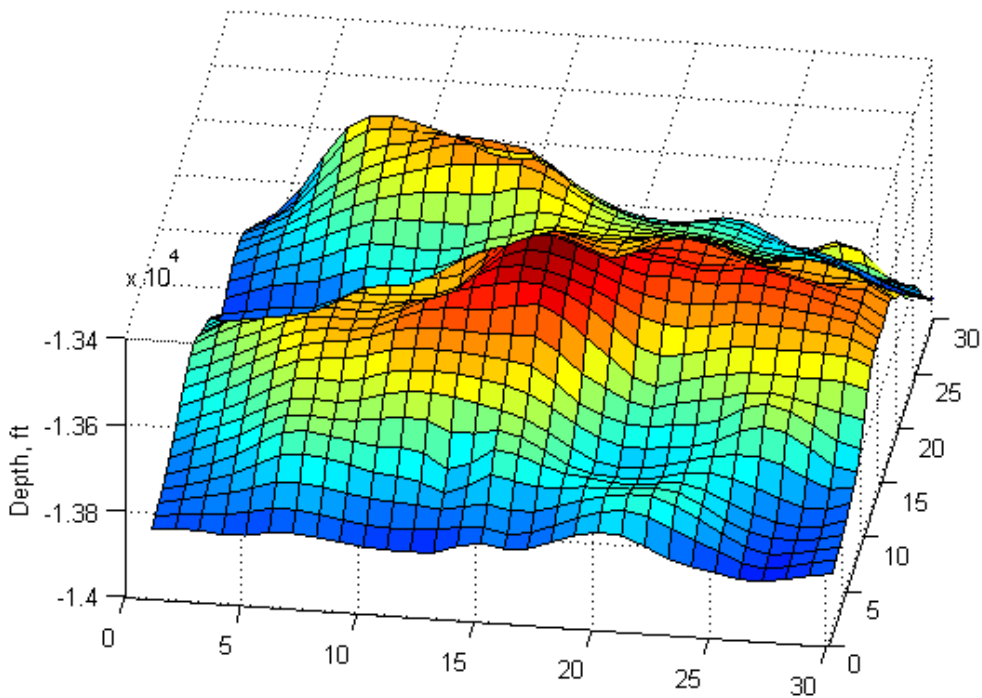
We can create a 3d visualisation of the top of the oil reservoir by using the `surf` command to generate a surface from the depth data..

```
surf(resTop)
zlabel('Depth, ft')
colorbar
```



Note that we now have a z axis we can label with the `zlabel` function. We can also add a colour bar to show what depth each colour represents. Note the american spelling of color for the `colorbar` command.

You can view the data in a surface plot from other angles by rotating the plot using the mouse (choose *Tools->Rotate 3D* from the figure menu).

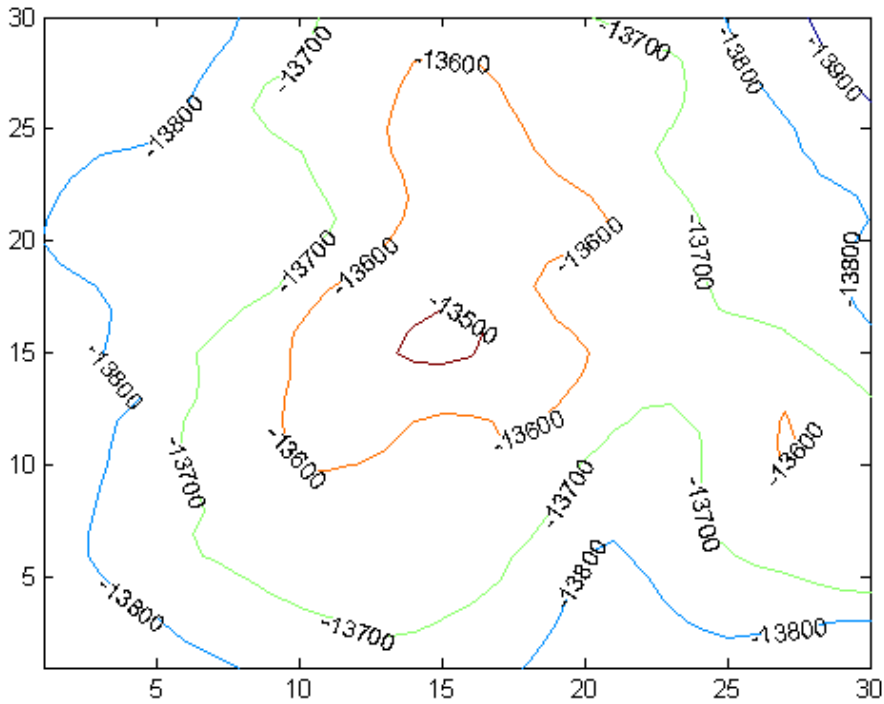


If you want to create a 2D plot which views the surface from directly above you can use `pcolor` instead of `surf`.

Contour plots

A contour plot is also a useful way to represent this kind of data. MATLAB's `contour` command will create contour plots from data in a 2D array.

```
contour(resTop)
```

To fill the area between the contours with a color use `contourf`.

Using `meshgrid` to create a mesh

Some 2D and 3D plots need 2D arrays of x and y values that describe a **mesh**.

The `meshgrid` function allows us to easily generate these 2D arrays from 1D arrays.

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

```
y = [0 0.5 1];
```

```
[X,Y] = meshgrid(x,y)
```

Creates:

X =

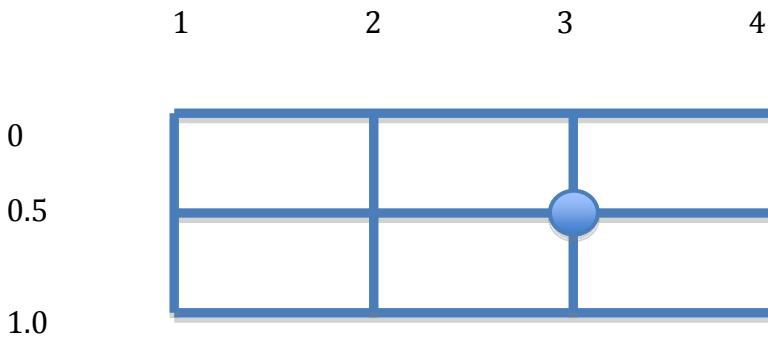
1	2	3	4
1	2	3	4
1	2	3	4

Y =

0	0	0	0
0.5000	0.5000	0.5000	0.5000
1.0000	1.0000	1.0000	1.0000

Together these two arrays describe 12 points on a mesh.

You can visualise the mesh as follows:



Note how the x values go **across** while the y values go **down**. Notice how if we overlaid the X array onto this mesh, it contains the x values for each point on the mesh. If we overlaid the Y array onto this mesh it contains the y values for each point on the mesh.

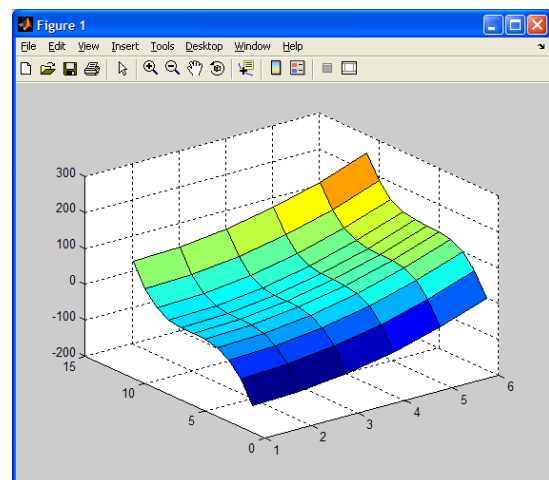
We can find the x and y values of any point in the mesh by looking up the values from the appropriate row and column or our 2D mesh arrays. eg The point shown in row 2 and column 3 of the mesh has x value 3 and y value 0.5.

<pre>X(2,3) ans = 3</pre>	<pre>Y(2,3) ans = 0.5</pre>
--------------------------------	---------------------------------

The 2D arrays representing the mesh can be used by `surf` and some other functions for plotting 2D arrays. In the case of the oil reservoir example, we could use `meshgrid` to easily create 2D arrays that describe the physical position of each depth measurement, to give meaningful x and y data.

The `meshgrid` command makes it very easy to generate plots of 3D polynomials. Here is an example of a plot of the surface described by $z = f(x,y) = 5x^2 + y^3$

```
x = 0:5;
y = -5:5;
[X, Y] = meshgrid(x, y);
Z = 5 * X .^ 2 + Y .^ 3;
surf(Z)
```



IMPORTANT: If we had wanted to display the actual x and y values, rather than just the indices for the array we would use `surf(X, Y, Z)`

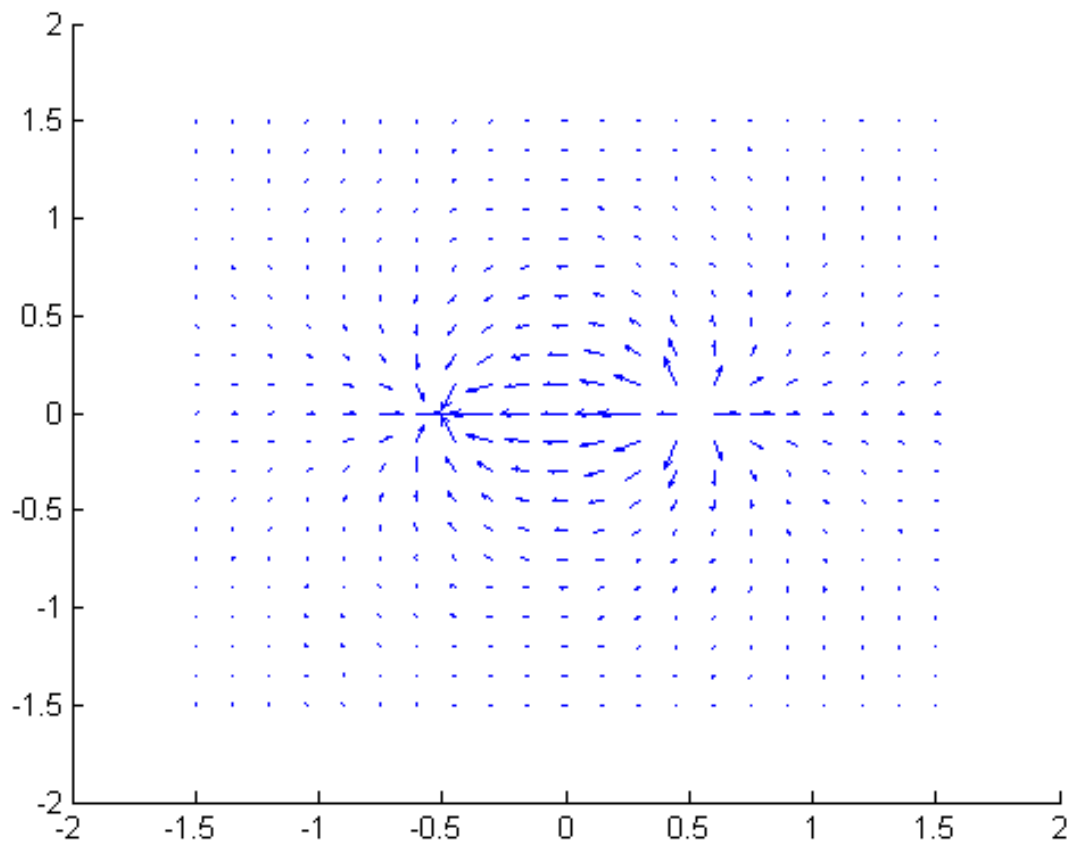
Quiver plots

Quiver plots are another useful way to represent many kinds of engineering data. These plots are useful for displaying vector quantities (e.g. velocity, electric or magnetic fields etc.) with arrows indicating both direction and magnitude. Quiver plots are often combined with surface plots and/or contour plots.

A quiver plot requires **four** 2D arrays. The first two arrays describe the x and y coordinates of a mesh (eg an X and Y array). The second two arrays describe the two components of a vector at each point in the mesh. We often refer to these as the U and V arrays.

Assume we have a grid described by an X and Y array on which a magnet sits. Assume that the strength and direction of the magnetic field at each point is described by two arrays U and V. Given this information we could generate a quiver plot as follows:

```
quiver(X, Y, U, V)
```



The arrows on the quiver plot are vectors with components (u,v) which describe the direction and strength of the magnetic field.

Putting plots into documents

If you want to put your plot into another document (such as a Microsoft Word document) first choose *Edit->Copy Figure* from the menu on the figure.

The figure can then be pasted into the other document.

Alternatively you can choose *File>Save As* from the menu on the figure and then select an image file type such as jpg, bmp or eps. These image files can then be imported into word or displayed on a web page.

Animating plots

Animation is quite simple in MATLAB. We just plot data repeatedly on a single figure, with a slight pause between each plot.

For example to plot the function $y=\sin(x+t)$ over the x range 0 to 2π and the time range 0 to 5 we can do the following:

```
x = 0:2*pi/100:2*pi;
for t=0:0.05:5
    y=sin(x+t);
    plot(x,y)
    pause(0.2)
end
```

To export this animation we can create a movie file that can be viewed by media players such as quick time.

To create a movie we "grab" a sequence of frames, storing each frame in an array and then writing the frame data out as an .avi file.

```
nFrame = 1;
x = 0:2*pi/100:2*pi;
for t=0:0.05:5
    y=sin(x+t);
    plot(x,y)
    movieData(nFrame) = getFrame
    nFrame = nFrame + 1;
end
movie2avi(movieData,'animation');
```

Chapter 7 Summary Program

We can now write programs that produce a variety of 2D and 3D plots

```
% this program plots two functions of one variable,  
% combines them to produce a function of two variables  
% and then plots this function as a surface in 3D.  
  
% set up x and y values which will define a grid  
x=linspace(0,pi,100);  
y=linspace(0,2,100);  
  
% calculate values for 2D functions  
fx = sin(x);  
gy = exp(y);  
  
% set up grid and calculate values for 3D surface function  
[X,Y] = meshgrid(x,y);  
Z = sin(X) .* exp(Y);  
  
% first plot the 2D functions, on separate axes but the  
% same figure  
figure(1);  
subplot(2,1,1);  
plot(x,fx);  
title('sine wave');  
xlabel('x');  
ylabel('sin(x)');  
  
subplot(2,1,2);  
plot(x,gy);  
title('exponential function');  
xlabel('x');  
ylabel('exp(x)');  
  
% now plot the 3D surface on a second figure  
figure(2);  
surf(X,Y,Z);  
title('An interesting surface formed from sin(x)*exp(y)');  
xlabel('x axis');  
ylabel('y axis');  
zlabel('height');
```

Chapter 8: Strings

So far we have dealt mainly with variables that contain numbers. Variables can also contain "strings" of characters. A string is an array of characters. Strings are useful for storing text based data and for displaying messages to users.

Learning outcomes

After working through this chapter, you should be able to:

- Explain what a string is
- Create strings and string variables
- Use and manipulate strings and strings variables
- Compare strings
- Search for substrings
- Format strings using `sprintf`
- Extract values from strings using `sscanf`
- Create and access cell arrays.

What are strings?

The scalars, vectors and matrices we have created to contain variables have generally contained numbers. It is also possible to create arrays that contain characters. MATLAB uses Unicode to store characters.

Each character on your keyboard is represented using a special value. Unicode is a character encoding scheme that includes not only English characters but nearly all characters from every major world language, both past and present. It is capable of describing over a million different characters.

You should also be aware of the older ASCII code for storing characters that is based on the order of the English alphabet. ASCII uses only 7 bits to represent a character and can only store 127 values. This is enough to describe all the characters on a standard English keyboard and it was commonly used to store characters before Unicode was created.

To represent a word or a sentence we can create an array of characters. This is called a string. MATLAB uses row arrays of characters to represent strings.

Creating strings

String variables are named just like numerical variables. They are assigned values by using either other string variables or string constants. A string constant is a sequence of characters between single quotes. Here is an example of creating a string using a string constant:

```
message = 'Hello World';
```

This string is an array of 11 characters. We can check this with the size command

```
size(message)
```

```
ans =
```

```
1 11
```

The 11 characters (including the space) have been stored in a 1x11 row array. We can access individual array elements, to see what letter is stored there

message(2)	message(7)
ans =	ans =
e	W

We can also create a second string variable as follows:

```
sameMessage = message;
```

sameMessage is a second string that will also contain the 11 characters 'Hello World'.

We can also create strings from numerical values using functions like `num2str` and `int2str` and `char`. These functions convert a numerical value into the equivalent string. It is important to realize the difference between the **number** 123 and the **string** '123'. The number can be used in numerical calculations, whereas the string consists of the 3 characters '1', '2' and '3' and it is used for displaying the characters '123' rather than for calculations. The following program illustrates the difference between the two kinds of variables:

```
numbers = 123;  
letters = '123';  
disp(numbers+10)  
disp(letters+10)
```

Running this results in:

```
133
```

```
59 60 61
```

Adding 10 to the `letters` string resulted in a value of 10 being added to each of the 3 characters stored in the string. The character '1' is represented in Unicode by using the value 49, hence the first value in the array was printed out as 59. Likewise '2' is represented using the value 50 and '3' the value 51.

To see what Unicode character is represented by the value 59 we could use the `char` function, which converts positive integers into the equivalent character:

```
char(59)
```

The value 59 represents the semi-colon

```
ans = character.
```

```
;
```

Confusing numerical values and strings can produce very unusual errors in your programs.

We can also create strings using the input command. If we wish to interpret what the user types as a string rather than a numerical value, the optional 's' argument must be passed to the input function:

```
name = input('Please enter your name:', 's');
```

The 's' tells the input function to interpret whatever is typed as a string.

String variables are useful for preloading strings we may wish to use in a range of different situations. eg:

- Input commands to the user
- Messages for display
- Plot titles, labels etc

Storing text in strings is also often useful as we can then use functions to search through our strings and look up text information.

Manipulating strings

As strings are simply character arrays we can use many of MATLAB's methods for handling arrays with them.

Accessing individual elements

Individual characters can be accessed, just as we access individual elements of an array:

```
department = 'engsci'
```

```
c = department(1)
```

```
c
```

```
    = 'e'
```

```
department(1)='E';
```

```
department(4)='S'
```

```
department
```

```
    = 'EngSci'
```

Obtaining subranges

```
department = 'EngSci'
```

```
department(1:3)
```



```
department(4:6)
```

Concatenation

We can concatenate strings together, just as we can concatenate normal arrays together. Concatenation works on both string variables and string constants.

```
department = 'EngSci'
task = 'problem solving'

totalMessage = [department ' equals ' task]

totalMessage =

    'EngSci equals problem solving'
```

Note that it is NOT possible to concatenate an array of characters with an array of numerical values. The following will produce an unexpected result.

```
message = ['value of 10 squared is ' 10^2]
```

This is because we are attempting to concatenate the array 'value of 10 squared is ' with the numerical value 100.

Instead we must first convert the numerical value of 100 to the string '100':

```
message = ['value of 10 squared is ' num2str(10^2)];
```

Changing case

Strings are case-sensitive. 'a' is not equal to 'A'. All characters in a string can be converted to lower or upper case by using the lower or upper functions.

```
department = 'EngSci';

display(lower(department));
display(upper(department));
```

Running the above code produces the output:

```
engsci
ENGSCI
```

Comparing strings

Two strings are equal if they contain exactly identical characters in each position, including characters such as spaces. This means they need to be exactly the same length. Comparing two strings is very useful for checking what text a user has typed in (eg if they are prompted to type 'yes' or 'no' to a question we can compare the string entered with the 'yes' and 'no' strings and then take appropriate action).

String comparison is also handy when you wish to search through a cell array of strings, to find one particular string that matches up with a string the user has entered. This allows us to do things like search through lists of names until we find the correct person.

There are several functions for comparing two strings

<code>strcmp</code>	Compare two strings
<code>strncmp</code>	Compare first n positions
<code>strcmpi</code>	Compare ignoring case

The `strcmp` takes two strings as inputs and returns 1 if they are identical and 0 otherwise. Strings are case sensitive, so the string 'Hello' is NOT the same as the string 'hello'.

The `strncmp` takes two strings as inputs and an integer n. `strncmp` returns 1 if the first n positions of the two strings match exactly. Otherwise 0 is returned. `strncmp` is case sensitive.

The `strcmpi` function takes two strings as inputs and returns 1 if they are identical, ignoring case. Otherwise 0 is returned.

String comparison examples

```
string1 = 'engsci'  
string2 = 'EngSci'
```

<pre>strcmp(string1,string2) ans = 0</pre>	<pre>strcmp(string1,'engsci') ans = 1</pre>
<pre>strncmp('engsci','Engsci',3) ans = 0</pre>	<pre>strncmp(string2,'Engsci',3) ans = 1</pre>
<pre>strcmpi(string1,string2) ans = 1</pre>	<pre>strcmp(string1,'engsc') ans = 0</pre>

The `strcmpi` function is particularly handy when comparing user input, as if a user answered yes to a question we don't care whether they typed 'YES', 'yes' or 'Yes'.

```

ansr = input('Does strcmpi ignore case when comparing strings?','s')
if (strcmpi('yes',ansr))
    disp('Excellent answer!');
else
    disp('Sorry, you are wrong');
end

```

Regardless of what case the user types the word yes in, the text 'Excellent answer!' will be displayed.

An alternative to using the strcmpi function is to convert all strings to the same case before comparing them, eg:

```

ansr = input('Is there more than one string compare function?','s')
if (strcmp('no',lower(ansr)))
    disp('Wrong! There are several. ');
else
    disp('Correct');
end

```

Searching for substrings

Sometimes instead of comparing two strings to see if they are identical we wish to see if a small "substring" occurs in a larger string, and if so where it occurs.

The `strfind` function allows us to search a string for occurrences of a shorter string. It takes as inputs the string and a shorter pattern string to search for. The output is an array that contains the starting index of each occurrence of the pattern. If the pattern was not found then the array will be empty (which means it has zero length).

```
strfind('Bananarama','ana')
```

```
ans =
     2     4
```

```
dept = 'Engineering Science'
strfind(dept,'ng')
```

```
ans =
     2    10
```

```
strfind(dept,'science')
```

```
ans =

[]
```

Note the word science was not found, as `strfind` is case sensitive. If we wanted to search for the substring 'science' regardless of case we could have done:

```
strfind(lower(dept),'science')
```

```
ans =
```

```
13
```

If all you care about is whether a substring was found or not you can simply examine the length of the array to determine how many times the substring was found.

```
phrase = input('Enter a sentence:', 's');  
theLocations = strfind(lower(phrase), 'the');  
  
if (length(theLocations) == 0 )  
    disp('Your sentence did not contain the letters "the"');  
else  
    disp('Your sentence contained the letters "the"');  
end
```

Formating with sprintf

The `sprintf` command allows us to create nicely formatted strings for display to users, or to use with string commands. The syntax of the `sprintf` command is:

```
s = sprintf(format, variables);
```

The first argument is a format string that includes special characters, which define how to write out the data. Following the format string are the variables to write out. The output from `sprintf` is the format string with each format specifier replaced by one of the formatted variables. Variables are processed in order, so that the first format specifier is replaced by the value of the first variable, the second specifier by the second variable and so forth.

A simple example will help to make things clearer.

```
name = 'Bob';  
mark = 98;  
  
s = sprintf('%s scored a mark of %i out of 100', name, mark)  
  
s =  
    Bob scored a mark of 98 out of 100
```

The first specifier is `'%s'`, which specifies a string value. The first variable, `name`, is inserted as a string in place of the `'%s'`. The second specifier is the `'%i'`, which specifies an integer value. The second variable, `mark`, is inserted as an integer in place of the `'%i'`. The result is a nicely formatted string that is stored in `s`.

There is a wide range of specifiers available. Here are some of the more common ones:

specifier	output	example
<code>%s</code>	string	hello
<code>%c</code>	character	c

%d or %i	decimal integer	-23
%e	scientific notation	1.2345e+10
%f	decimal floating point	23.1234
%g	the shorter of e or f	

Inserting a number between the % character and the specifier allows you to specify the minimum width to reserve for displaying the value. This is handy when wanting to format output in columns.

```
string = sprintf('The sqrt of %5d is %3d',1,1);
disp(string);
string = sprintf('The sqrt of %5d is %3d',100,10);
disp(string);
string = sprintf('The sqrt of %5d is %3d',10000,100);
disp(string);
```

```
The sqrt of    1 is    1
The sqrt of   100 is   10
The sqrt of 10000 is 100
```

Notice how 5 characters are set aside for the first value and spaces are added to make sure the value takes up 5 characters worth of space.

When formatting numerical values you can control the number of decimal places to use. Insert a decimal point followed by a number between the % character and the specifier to indicate how many decimal places to use:

```
s = sprintf('1/900 with 8dp and scientific notation is %.4e',1/900)
```

Running this results in:

```
s =
1/900 with 8dp and scientific notation is 1.1111e-003
```

Note that the variables passed in can either be just a variable or an expression that MATLAB will calculate before formatting:

```
s = sprintf('pi with 2dp is: %.2f and pi squared is: %.2f',pi,pi^2)
```

Running this results in:

```
s =
pi with 2dp is: 3.14 and pi squared is: 9.87
```

If necessary you can control both the minimum width and the number of decimal places to use:

```
s = sprintf('pi with 2dp is: %6.2f',pi);
disp(s);
s = sprintf('pi with 3dp is: %6.3f',pi);
disp(s);
s = sprintf('pi with 4dp is: %6.4f',pi);
disp(s);
```

Running this results in:

```
pi with 2dp is:    3.14
pi with 3dp is:   3.142
pi with 4dp is:  3.1416
```

Special characters

Most characters you will deal with are either letters, numbers or punctuation symbols. There are however some special characters used to describe the layout of text on a page. When you type a document using a word processor, every key stroke is stored as a character. This includes things like when you hit the tab key or the enter key. Hitting tab in a word processor will insert a single “tab” character while hitting the enter key will insert a “newline” character.

If we wish to insert a tab or new line character into a Matlab string that will be processed by `sprintf`, we use a backslash followed by a special character code to do so. The backslash is sometimes referred to as an escape character, as it escapes normal character entry and allows us to enter a special character. For example a tab character can be inserted using `\t` while a newline character can be inserted using `\n`.

As the backslash is used for entering special characters, if you ever want to enter a backslash you need to use the special character code `\\`.

```
>> message = 'We use a \\n character to \nmove to a new line';
>> sprintf(message)
```

```
ans =
```

```
We use a \n character to
move to a new line
```

One other character which is tricky to enter in a string is the single quote character, as this is used to signal the end of a string. If you do want to enter a single quote character, enter two single quote characters in a row. This works both for strings passed to `sprintf` and the `disp` command.

```
>> message = 'Peter''s tip on entering single quotes, use two!';
>> disp(message)
```

```
Peter's tip on entering single quotes, use two!
```

Processing with sscanf

We can also use format specifiers to help us scan strings for formatted data.

The `sscanf` function keeps reading data of a specified format from a string until the end of the string is reached or it runs out of compatible data.

The syntax of the `sscanf` command is as follows:

```
data = sscanf(s, format, size);
```

The variable `s` is the string to scan. The `format` is a string detailing the format of the data we wish to read. The same format specifiers are available as those that we met when using `sprintf`.

The `size` argument is optional. If it is left off the `sscanf` command will attempt to read all values of the specified format until the end of the string is reached or incompatible data is found. If a size is specified it will attempt to read enough data to fill an array of the specified size.

The data read by `sscanf` is returned as an array.

```
string = 'e is approx 2.7183';  
e = sscanf(string, 'e is approx %f');
```

```
e =
```

```
    2.7183
```

The variable `e` is a 1x1 array containing an approximate value of the constant `e`.

```
string = 'x = 2.3 y = 1.5';  
point = sscanf(string, 'x = %f y = %f');
```

```
point =
```

```
    2.3000  
    1.5000
```

The variable `point` is a 2 element array which contains the `x` and `y` coordinates of the point.

Cell Arrays

Strings are arrays of characters. Sometimes we wish to work with arrays of strings. Using 2D arrays is one possibility but every row in a 2D array has the same length. Strings often have different lengths so instead of using a standard 2D array MATLAB provides us with a special kind of array where each element of the array is a string. These arrays are called Cell arrays. They are created and indexed with **curly braces**:

```
helloWorld = { 'hello', 'world' };
```

helloWorld is a two element *cell array*. It has length two. If we had used square brackets instead of curly braces, MATLAB would have concatenated the two strings into one. The length of this array would have been ten as 'helloworld' has ten characters.

You must use curly braces if you want to create a cell array.

We can access the first string in the array using curly braces:

```
greetingString = helloWorld{1};
```

greetingString now contains the string 'hello'.

Note that using round brackets results in something quite different:

```
greetingCellArray = helloWorld(1);
```

greetingCellArray now contains a 1x1 cell array, which has one string as it's element.

When working with cell arrays it is very important to remember to use curly braces. If you use round brackets to try and index a cell array, you will get a 1x1 cell array back rather than a string. This can be very confusing if you do not spot your error.

Some MATLAB string functions will also work on cell arrays but the results can be rather confusing. If working with a cell array it is often a good idea to use a for loop to work through your cell array, allowing you to work on one string at a time.

```
myMessage = { 'Remember', 'to', 'use', 'curly', 'braces' }
for i = 1:length(myMessage)
    word = myMessage{i};

    if (strcmp('curly',word))
        disp(['curly is word ' num2str(i)]);
    end
end
```

Running this will result in the output: curly is word 4

Next chapter we will learn about importing data from files into MATLAB. When importing a file into MATLAB it is quite common for a cell array to be created.

Chapter 8 Summary Program

We can now write programs that manipulate strings. For example, the following program searches for the exam result of a student name that is entered by the user.

```
% search for a student and display their exam results

% normally we would import some examination data from a file
% for this example we will just create a cell array of names
% and an array of marks
name = {'Adam Blogs','Adam Smith','John Smith'};
mark = [67, 42, 91];

searchName = input('Please enter a student name:', 's');

% total number of students
n = length(mark);

studentFound = 0;
i = 0;

% search until student found or we run out of students
while (~studentFound & i < n )

    % i is the index into the student mark arrays.
    i = i + 1;
    % get name from cell array, note curly braces
    studentName = name{i};
    locations = strfind(lower(studentName), lower(searchName));
    % check if search name found in one or more locations
    % of the student name string
    if (length(locations) > 0)
        disp(['Found student ' studentName])
        response = input('Is this the correct student?', 's');

        % check if first letter of reponse is y or Y
        if( strcmpi('Y', response(1)) )

            % use ... to spread this command over 2 lines
            display([studentName ' got a mark of ' ...
                    num2str(mark(i))]);

            studentFound = 1;
        end
    end
end
end
```

```
% summary program continued

if (studentFound == 0)
    message = sprintf('%d students were scanned',n);
    disp(message);
    message = sprintf('Student %s was not found',searchName);
    disp(message);
end
```

Examples of this program running are given below (user input is in bold):

```
Please enter a student name:adam
Found student Adam Blogs
Is this the correct student?n
Found student Adam Smith
Is this the correct student?y
Adam Smith got a mark of 42
```

```
Please enter a student name:smith
Found student Adam Smith
Is this the correct student?no
Found student John Smith
Is this the correct student?no
3 students were scanned
Student smith was not found
```

Chapter 9: Files

MATLAB commands are not the only thing we want to be able to store in files. Often we wish to manipulate some data in order to calculate a result. Usually this data is stored in a file. It is important to be able to get that data from the file into MATLAB, in a format that MATLAB can work with. We also may wish to write out the results of a calculation to a data file.

Learning outcomes

After working through this chapter, you should be able to:

- Load and save data to a MAT file
- Use the import wizard to import data from files
- Use the MATLAB file commands for reading data directly from a file
- Use the MATLAB file commands for writing data out to a file

MAT files

MATLAB has a file format designed for saving variables from the MATLAB workspace. Variables can then be loaded back into the workspace from the file.

These files are called MAT files and they have a .mat extension. By default the files are not human readable, which means that you cannot view the values of the variables by opening a MAT file in a text editor.

We use the `save` and `load` commands to save and load variables in the workspace.

Saving variables

Once we have created some variables in the workspace it is then possible to save them to a file for later use. For example:

```
A = [1 2; 3 4]
b = [1; 0]
time = 3;
```

```
save example A b time
```

The `save` command will create a file called `example.mat` and save the contents of variables `A`, `b` and `time` to the file. If we do not specify which variables to save, MATLAB will save **all** variables in the workspace to the file.

If you want to create a human readable data file you can save the data as ascii characters (strings) by using the `-ascii` option:

```
save example -ascii A b time
```

The file created can then be read by humans but it will take up more space on your computer's hard disk.

Loading variables

If we clear all our variables out (either by restarting MATLAB or using the clear command) it is now still possible to get one or more of our saved variables back again by using the load command:

```
load example A b
```

This loads variables A and b into the workspace from our file example.mat. Note that if wanted to load just the variable A we could type:

```
load example A
```

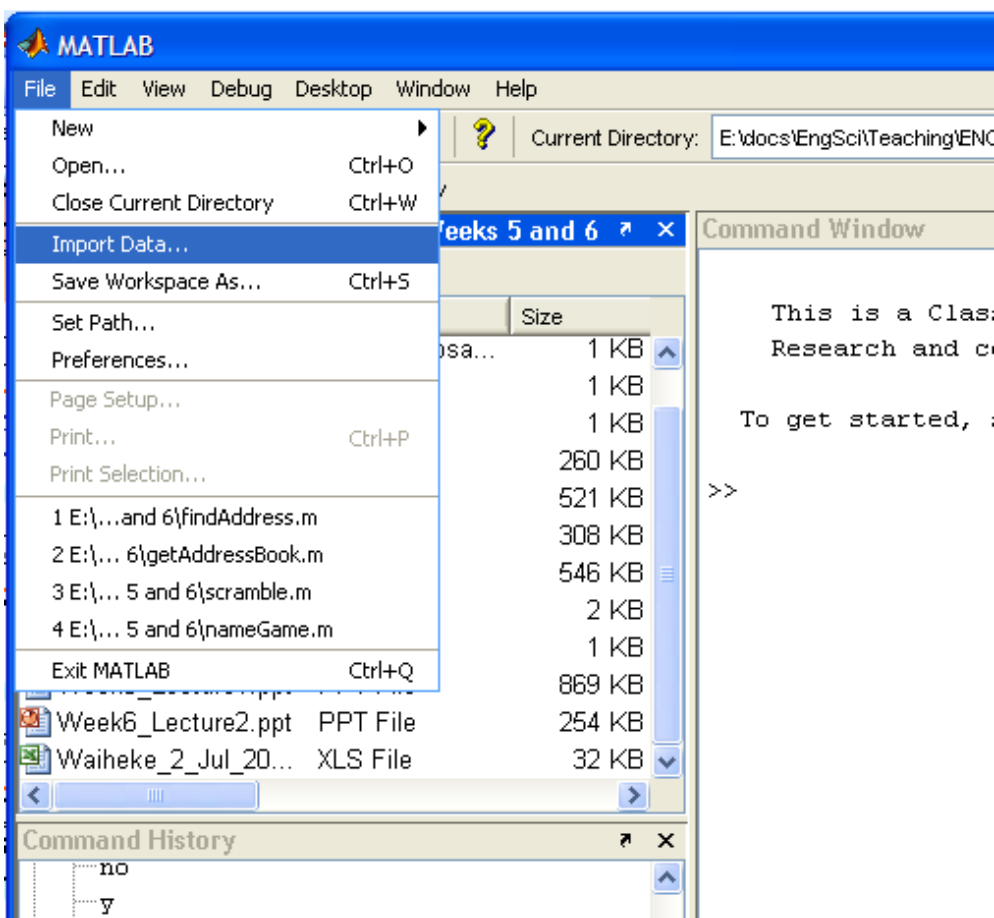
If we want to load everything stored in the file example.mat we would type:

```
load example
```

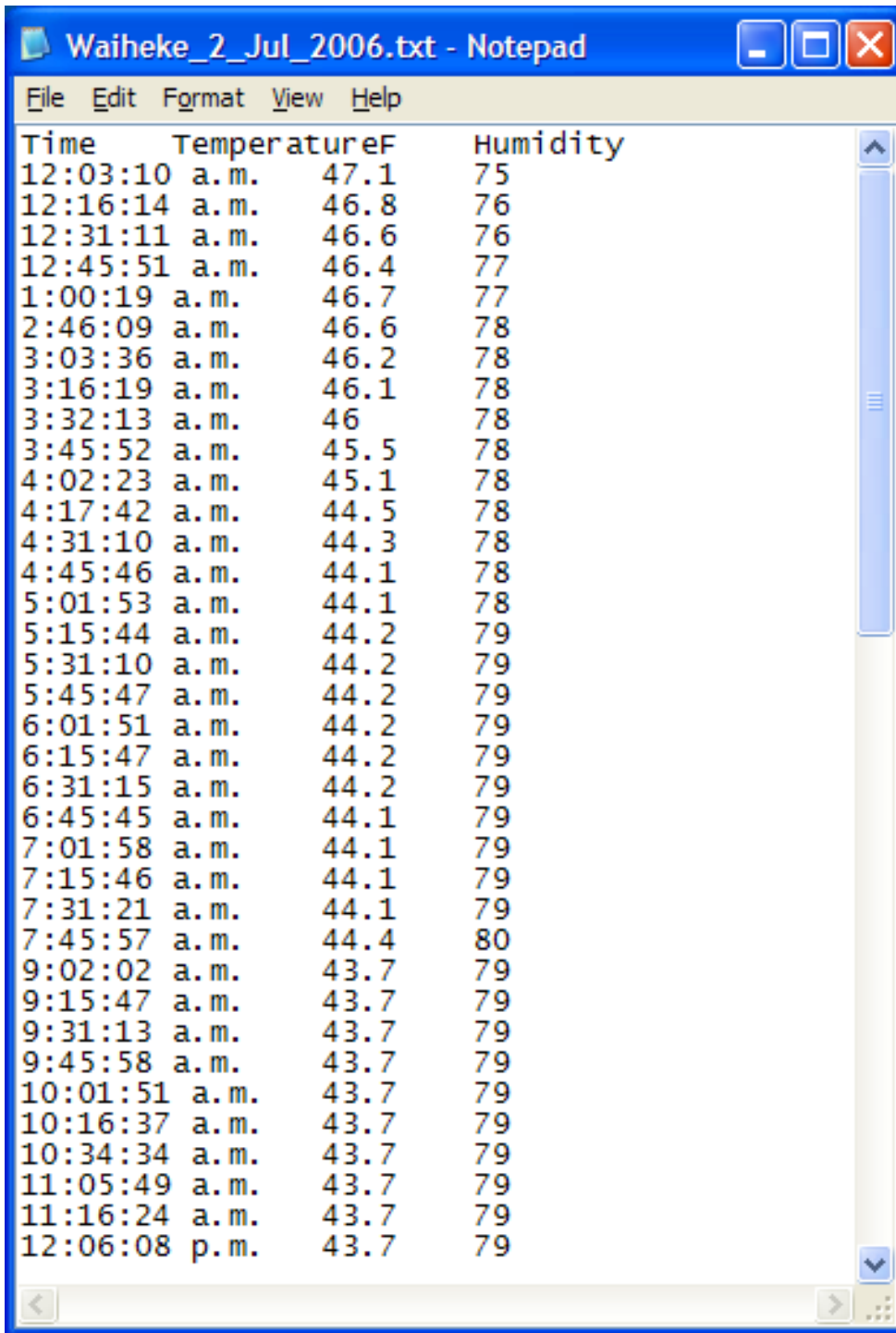
Importing data from files using the wizard

MAT files are ideal for saving the value of variables that were created using MATLAB. Sometimes we need to read in values from files that were created by other applications such as excel. MATLAB provides an import wizard, which helps you to read data from other file formats.

To start the import wizard go to the file menu and choose "Import Data"



You will then be prompted to select a file to import data from. This could be a txt file, a spreadsheet or some other kind of file. For example, we may wish to import the data contained in the following text file:

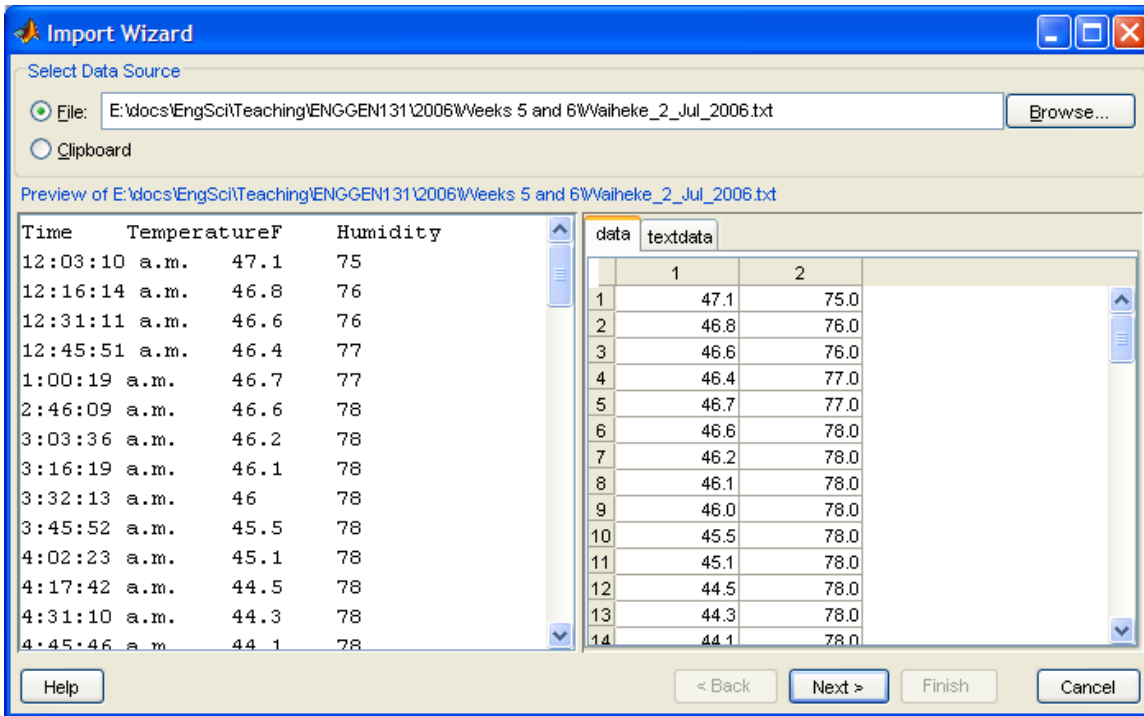


The screenshot shows a Notepad window with the following data:

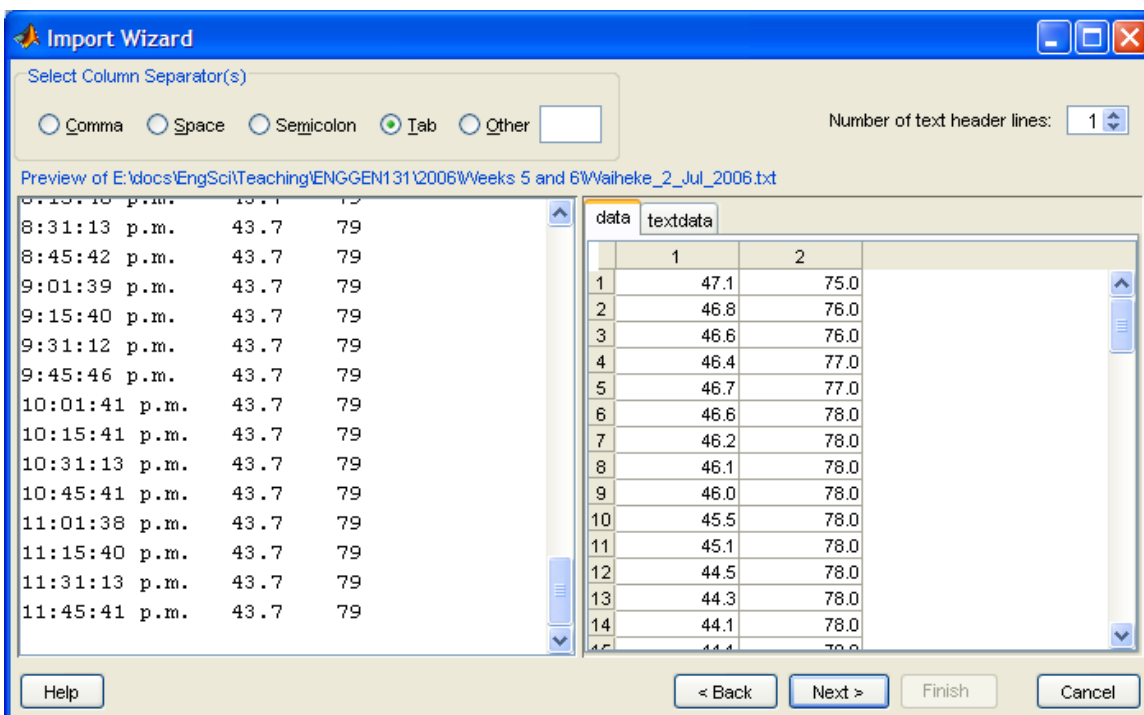
Time	TemperatureF	Humidity
12:03:10 a.m.	47.1	75
12:16:14 a.m.	46.8	76
12:31:11 a.m.	46.6	76
12:45:51 a.m.	46.4	77
1:00:19 a.m.	46.7	77
2:46:09 a.m.	46.6	78
3:03:36 a.m.	46.2	78
3:16:19 a.m.	46.1	78
3:32:13 a.m.	46	78
3:45:52 a.m.	45.5	78
4:02:23 a.m.	45.1	78
4:17:42 a.m.	44.5	78
4:31:10 a.m.	44.3	78
4:45:46 a.m.	44.1	78
5:01:53 a.m.	44.1	78
5:15:44 a.m.	44.2	79
5:31:10 a.m.	44.2	79
5:45:47 a.m.	44.2	79
6:01:51 a.m.	44.2	79
6:15:47 a.m.	44.2	79
6:31:15 a.m.	44.2	79
6:45:45 a.m.	44.1	79
7:01:58 a.m.	44.1	79
7:15:46 a.m.	44.1	79
7:31:21 a.m.	44.1	79
7:45:57 a.m.	44.4	80
9:02:02 a.m.	43.7	79
9:15:47 a.m.	43.7	79
9:31:13 a.m.	43.7	79
9:45:58 a.m.	43.7	79
10:01:51 a.m.	43.7	79
10:16:37 a.m.	43.7	79
10:34:34 a.m.	43.7	79
11:05:49 a.m.	43.7	79
11:16:24 a.m.	43.7	79
12:06:08 p.m.	43.7	79

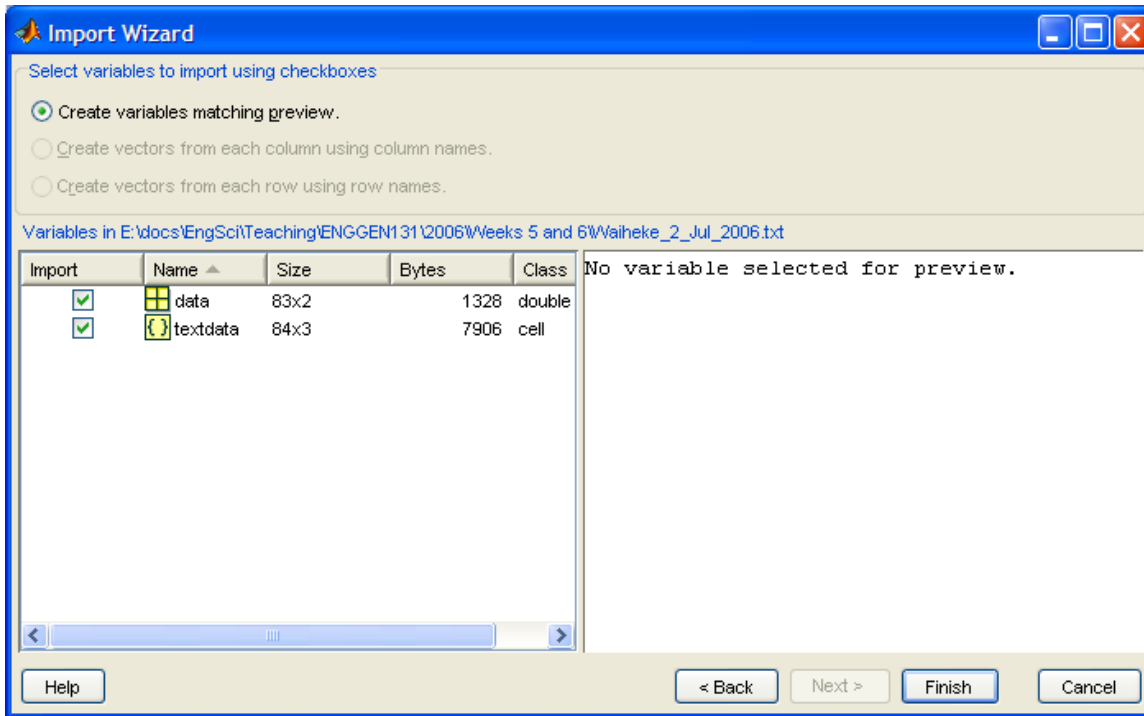
This file contains temperature and humidity data for a weather station on Waiheke island. The import wizard will attempt to read the data into arrays. Notice that the very first row contains headings for our three columns. The first column of our data file is a string specifying the time while the other two columns contain numerical values (temperature in Fahrenheit and humidity).

When we use the import wizard to open a file, the wizard will attempt to read data and create arrays to hold that data. It will create a preview of the arrays that will be created.



You may need to specify the character used to separate columns, if the import wizard has not picked the correct character. In this case the tab character was successfully used:





Generally the columns of your arrays will match up to columns in your data file. Notice that two arrays have been created in the above example. The data array contains numerical values. The textdata array contains strings (ie it is a cell array).

Only two columns from the file contain numerical values so the data array has only two columns. 83 rows contained numerical data hence the array size 83x2.

All three columns contain text (As the first row is all text) so our textdata array will have 3 columns. There are 84 rows that contain text (since every value in the first column is a string), hence the array size of 84x3.

Once you have finished importing the data, the variables will be created in the workspace. It is sometimes convenient to assign data values to more meaningful names.

eg in this case we could use array slicing to create sensibly named arrays

```
time = textdata(2:84,1)
temperatureF = data(:,1);
humidity = data(:,2);
```

Depending on the file format the wizard is processing you may be given the option to "Create vectors from each column using column names", rather than having to do it manually.

File I/O

Unfortunately sometimes the import Wizard fails to import our data correctly and we need another way of reading data from files. Note that it is impossible for MATLAB to understand all file formats, as people create new formats all the time, so it is important that we have ways of dealing with unusual file formats.

Fortunately we can write our own MATLAB code to read formatted files. We can also use MATLAB to write out nicely formatted files, allowing us to create our own file formats for storing our data.

MATLAB provides a number of I/O (Input/Output) functions that can be used for reading input from files and writing output to files. The I/O functions are based on the standard C library functions for reading and writing to files. You will encounter these C library functions in the C part of the course.

Before reading or writing a file you need to open the file. After working with a file you should always close it.

Opening and closing files

To work with a file we need to first open it and get an identifier for the file. It is possible to have several files open at once and the file identifier provides an id we can use to tell MATLAB which file to read or write from. To open a file we use the `fopen` command:

```
fid = fopen(filename, permission);
```

The `filename` is a string containing the name of the filename to open. The `permission` variable is a string which tells the `fopen` command what kind of access we want to the file (eg permission to read or permission to write).

Some possible values for the permission string are listed in the table below.

'r'	Reading only
'w'	Writing only
'r+'	Reading and writing
'a'	Appending only (adding on to the end of a file)

If the `fopen` command successfully opens a file it will return an integer file id. If the file fails to open it will return the value -1. A file may fail to open if the incorrect file name has been provided or if the file is being used by some other application. It is a good idea to check the value of the file id and display an error message if the file failed to open.

To open a file called `mydata.dat` for reading we would use the following command:

```
fid = fopen('mydata.dat','r');
```


It is a good idea to check if the file opened successfully by testing the `fid` value:

```
if fid == -1
    disp('Error opening file')
end
```

Once we have finished reading or writing to a file it is important to tidy up after ourselves and close the file. A file that is still open may not be able to be used by other applications. Closing the file also frees up some computer memory. To close the file we use the `fclose` command, passing in the file id of the file to close:

```
status = fclose(fid);
```

The `fclose` command returns a status variable that indicates whether or not the file was closed successfully. If the file closed successfully a value of 0 is returned. If there was a problem closing the file a value of -1 is returned.

Reading from files

There are two possible functions to use when reading files. The `fscanf` function allows you to potentially read the contents of an entire file with just one command. You can also use it to read a set number of values from a file. It is very convenient when the file format is very simple. The `fgetl` command allows you to read a file line by line. For more complicated file formats this is useful as you can then use string processing to extract data from each line in turn.

fscanf

The `fscanf` function keeps reading data of a specified format until the file is finished or it runs out of compatible data.

The syntax of the `fscanf` command is as follows:

```
data = fscanf(fid, format, size);
```

The `fid` is the identifier of the file we are reading from. The `format` is a string detailing the format of the data we wish to read. The same format specifiers are available as those that we met when reading and writing strings. It may help to remember that `fscanf` is very similar to `sscanf`, the difference being that it scans a file instead of a string.

Here is a reminder of some of the more common format options.

specifier	output	example
%s	string	hello
%c	character	c
%d or %i	decimal integer	-23
%e	scientific notation	1.2345e+10
%f	decimal floating point	23.1234
%g	the shorter of e or f	

The `size` argument is optional. If it is left off the `fscanf` command will attempt to read all values of the specified format until the end of the file is reached or incompatible data is found. If a size is specified it will attempt to read enough data to fill an array of the specified size.

The data read by `fscanf` is returned as an array.

The following code opens a file and then reads in as many real numbers from a file as it can:

```
fid = fopen('mydata.dat','r');
data = fscanf(fid, '%f');
fclose(fid);
```

The following code opens a file and then reads in only the first 5 integer values from the file:

```
fid = fopen('mydata.dat','r');
data = fscanf(fid, '%i', 5);
fclose(fid);
```

The following code opens a file and then reads the first 8 read numbers into a 2x4 array:

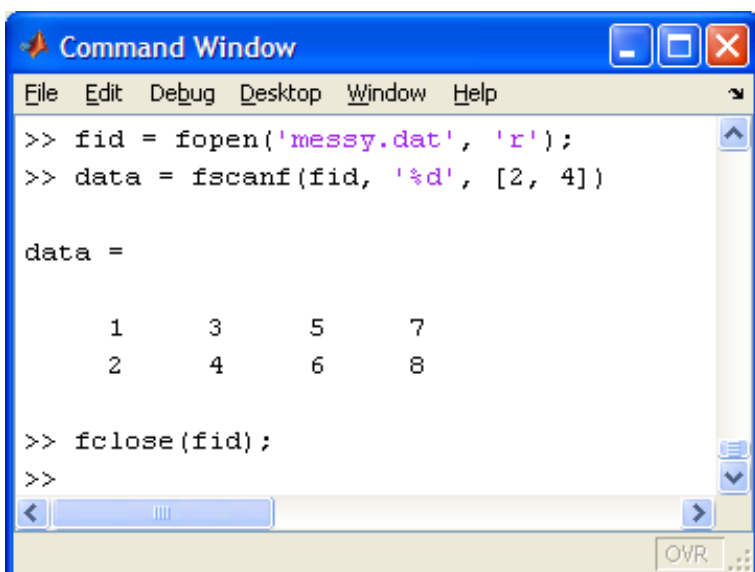
```
fid = fopen('messy.dat','r');
data = fscanf(fid, '%d', [2,4])
fclose(fid);
```

Note that the values are read from the file row by row (going across) but data is written column by column.

If the contents of `messy.dat` is as follows:

```
1 2 3
4 5
6 7 8 9
10
11 12 13
```

When we use the above code we get the following result:



```
Command Window
File Edit Debug Desktop Window Help
>> fid = fopen('messy.dat', 'r');
>> data = fscanf(fid, '%d', [2, 4])

data =

     1     3     5     7
     2     4     6     8

>> fclose(fid);
>>
```

fgetl

The `fgetl` commands gets a single line from a file and stores the line as a string. The string can then be processed using the various string functions we have already encountered such as `sscanf` or `str2num`.

The syntax of the `fgetl` command is as follows:

```
line = fgetl(fid)
```

The `fgetl` commands returns the next line of the file associated with file identifier `fid`. The MATLAB string returned does NOT include the end of line character `'\n'`. If the end of file is encountered then the value `-1` is returned.

Here is an example of using `fgetl` to read a line from our `messy.dat` file:

```
fid=fopen('messy.dat','r');

% get first line from the file
line = fgetl(fid);

% read three integers from the string into an array
values = sscanf(line,'%d %d %d');

fclose(fid);
```

It is common to use a while loop to loop through a file and process each line one at a time. This can be done as follows:

```
% Open the file for reading
fid=fopen('messy.dat','r');

if( fid == -1)
    disp('Error opening the file');
else
    % get first line from the file
    line = fgetl(fid);

    % loop through the file until we run out of lines
    while (line ~= -1)
        % display contents of line string, could process it instead
        disp(['Line read was:' line])
        % get next line
        line = fgetl(fid);
    end

    % close the file once we are finished with it
    fclose(fid);
end
```

Writing to files

Just as for reading from files, we need to first open a file before we can write to it, only this time we need to open it with write permission, using 'w'.

```
fid = fopen('squares.txt','w');
```

To write to a file we use the `fprintf` command. This is very similar to the `sprintf` command, except that instead of returning a string, the string is written to the file.

The syntax of the `fprintf` command is:

```
fprintf(fid, format, variables);
```

It writes out the variables using the specified format until all values of the specified format have been written. The format specifiers are the usual specifiers we use with the `fscanf`, `sscanf` and `sprintf` commands.

If the variables include arrays then the values are written out column by column.

```
% This script writes out the first 10 square numbers
% to a file

x = 1:10
squares = x.^2;

% open the file for writing to
myfid = fopen('squares.txt','w');

% write out array to file, one value per line
fprintf(myfid,'%d\n',squares);

% close the file once we are finished with it
fclose(myfid);
```

Chapter 9 Summary Program

We can now read data from files and write data to files.

For example, the following program writes some exponential data to a file and the second program reads that data back in.

```
% This script writes data to exponential.txt
% each line consists of an exponent and the value
% of e to that exponent

% calculate data to write

x = 0:.1:1;
y = [x; exp(x)];

% open the file for writing to
myfid = fopen('exponential.txt','w');

% write out array to file, column by column
fprintf(myfid,'%6.2f  %12.8f\n',y);

% close the file once we are finished with it
fclose(myfid);
```

```

% This script reads the data from exponential.txt
% one line at a time

% Open the file for reading
filename = 'exponential.txt';
fid=fopen(filename,'r');

if( fid == -1)
    disp(['Error opening file ' filename]);
else
    % get first line from the file
    line = fgetl(fid);

    % loop through the file until we run out of lines
    while (line ~= -1)

        % read two values from the line into a 2 element array
        values = sscanf(line,'%f %f');
        message = sprintf('e to the power of %g is %g',values);
        disp(message);

        % get next line
        line = fgetl(fid);
    end

    % close the file once we are finished with it
    fclose(fid);
end
end

```

Chapter 10: Linear Equations and Linear Algebra

MATLAB has very good support for working with both vectors and matrices. This makes it an ideal tool for solving linear algebra problems. In particular it is very easy to solve systems of linear equations that are written in Matrix form.

Learning outcomes

After working through this chapter, you should be able to:

- Solve systems of linear equations using MATLAB
- Solve basic linear algebra problems with MATLAB
- Use matrix transformations (rotation, translation, scaling, shearing)
- Apply transition matrices

Solving systems of linear equations

Systems of linear equations appear in many different branches of engineering. In MM1 you were introduced to solving equations using Gaussian elimination. You will also have learnt how to use matrix algebra to solve systems of linear equations in matrix form. MATLAB supports both methods.

Solving a linear equation in 3 lines

Consider the following system of linear equations:

$$\begin{aligned}x_1 + 2x_2 + 3x_3 &= 1 \\2x_1 + 5x_2 + 3x_3 &= 6 \\x_1 + \quad + 8x_3 &= -6\end{aligned}$$

This system of linear equations can be written in matrix form as:

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 5 & 3 \\ 1 & 0 & 8 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 6 \\ -6 \end{bmatrix}$$

which is of the general form: $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 5 & 3 \\ 1 & 0 & 8 \end{bmatrix}$ $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$ $\mathbf{b} = \begin{bmatrix} 1 \\ 6 \\ -6 \end{bmatrix}$

The solution of $\mathbf{Ax} = \mathbf{b}$ is $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$, ie \mathbf{x} is the inverse of \mathbf{A} multiplied by \mathbf{b} .

To solve this equation in MATLAB we simply type the following:

```
A = [1 2 3; 2 5 3; 1 0 8];  
b = [1; 6; -6];  
x = inv(A) * b
```

The result is:

```
x =  
  
    2.0000  
    1.0000  
   -1.0000
```

Note that MATLAB displays only 4 decimal places by default. The 4 zeros tell us that the first value is close to *but not exactly* 2. Similarly the second and third values are not exactly whole numbers.

We can use the "left division" operator to find a solution using a method based on using Gaussian elimination:

```
A = [1 2 3; 2 5 3; 1 0 8];  
b = [1; 6; -6];  
x = A \ b
```

The result is:

```
x =  
  
    2  
    1  
   -1
```

MATLAB does all the hard work for us and the solution by Gaussian elimination matches exactly the real solution, so the left division operator seems like a better method to use. In general the left division method is more accurate. (Note we can always check our solution by calculating $A*x$ to see if we get b)

Some background theory

In MM1 you were introduced to the matrix form of systems of linear equations:

$$Ax = b.$$

Recall that if the matrix A has an inverse you can multiply both sides of the equation by the inverse of A to get:

$$A^{-1}Ax = A^{-1}b$$

$$Ix = A^{-1}b$$

$$x = A^{-1}b$$

That is, *the solution is the inverse of A multiplied by b.*

Remember that this method only works if **A** has an inverse. A matrix has an inverse if, and only if, the determinant of the matrix is nonzero. To find the determinant of a square matrix **A** we can use the `det` command:

```
det(A)
```

If the determinant is nonzero then the `inv` command can be used to calculate the inverse.

Once the matrix **A** and the column vector **b** have been assigned values it only takes one line of code to solve the system.

```
x = inv(A) * b
```

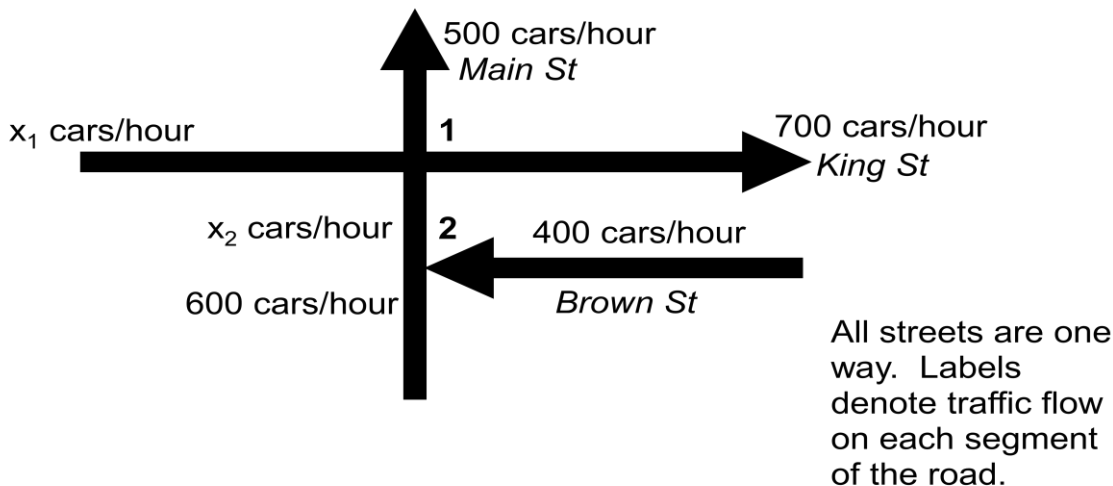
MATLAB also supplies the left division method, which performs the equivalent to Gaussian elimination but is very quick to type:

```
x = A\b
```

The left division method is preferred as computing the inverse of a matrix and then performing a matrix multiplication takes more time than performing Gaussian elimination. Also the left division method is less likely to introduce errors due to rounding. It can still be worth checking the determinant of **A** is nonzero, to check whether our equations have a solution.

A traffic flow example

Consider the problem of determining how many cars per hour travel through two sections of street.



We know how many cars go into and out of Main St. We also know how many cars go into Brown St and how many come out of King St.

We would like to determine how many cars per hour come into King St and how many travel on the small section of Main St between Brown and King St.

This particular problem is easy to solve by hand but we can use the same principles to determine traffic flow in much larger road networks.

Recall the five steps for problem solving

1. State the problem clearly
2. Describe the input and output information
3. Work the problem by hand (or with a calculator) for a simple set of data
4. Develop a solution and convert it to a computer program
5. Test the solution with a variety of data

State the problem clearly

Determine x_1 and x_2 using the known traffic flows of 400, 500, 600 and 700 cars/hour on segments of Main St, King St and Brown St (which are one way streets).

Describe the input and output information

Input:

- Main St (north of intersection 1), 500 cars/hour
- King St (west of intersection 1), 700 cars/hour
- Brown St, 400 cars/hour
- Main St (south of intersection 2), 600 cars/hour

Output:

- x_1 cars per hour travelling on King St (east of intersection 1)
- x_2 cars per hour travelling on Main St (between intersection 1 and intersection 2)

Work the problem by hand (or with a calculator) for a simple set of data

Balance the flow of cars into and out of each intersection.

$$\text{Intersection 1 } x_1 + x_2 = 500 + 700$$

$$\text{Intersection 2 } x_2 = 600 + 400$$

Clearly x_2 is equal to 1000 cars/hour, so

$$x_1 + 1000 = 1200$$

$$\Rightarrow x_1 = 200$$

Develop a solution and convert it to a computer program

The two equations we need to solve can be expressed as

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1200 \\ 1000 \end{bmatrix}$$

To solve this problem in MATLAB we can use the “left division” operator

```
A = [1, 1; 0, 1];  
b = [1200; 1000];  
x = A\b
```

```
x =
```

```
    200  
   1000
```

Test the solution with a variety of data

To test our solution we verify that $A*x$ is equal to b .

```
A*x
```

```
ans =
```

```
   1200  
   1000
```

Solving basic linear algebra problems

Much of the drudgery of working with vectors can be removed by using MATLAB functions. The table below lists some of the key functions available for working with vectors

Function	Use	Example
norm	Length of a vector	norm(a)
dot	Dot product (scalar product)	dot(a,b)
cross	Cross product (vector product)	cross(a,b)

Armed with these functions we can easily solve a range of linear algebra problems.

Find the angle between two vectors

Find the angle between the two vectors: $\mathbf{a} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ $\mathbf{b} = \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix}$

Recall that $\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}||\mathbf{b}|\cos\theta$

Hence that angle theta is given by: $\theta = \arccos \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}||\mathbf{b}|}$ where arccos is the inverse cosine function

Using MATLAB we can find theta as follows:

```
a = [1; 2; 3];  
b = [3; 2; 1];  
theta = acos( dot(a,b) / (norm(a)*norm(b)) )
```

```
theta =
```

```
0.7752
```

Find the projection of one vector onto another

Find the projection of vector \mathbf{u} onto vector \mathbf{v} : $\mathbf{u} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ $\mathbf{v} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$

Recall that $\mathbf{p} = \frac{\mathbf{u} \cdot \mathbf{v}}{\mathbf{v} \cdot \mathbf{v}} \mathbf{v}$

Using MATLAB we can find the projection as follows:

```
u = [1; 2; 3];  
v = [1; 0; 1];  
p = ( dot(u,v)/dot(v,v) ) * v
```

p =

```
2  
0  
2
```

Find the area of a triangle defined by two vectors

Find the area of the triangle defined by the two vectors: $\mathbf{a} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ $\mathbf{b} = \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix}$

Recall that $\mathbf{a} \times \mathbf{b} = |\mathbf{a}||\mathbf{b}|\sin\theta\hat{\mathbf{n}}$

The cross product creates a vector that is normal to both \mathbf{a} and \mathbf{b} , with magnitude $|\mathbf{a}||\mathbf{b}|\sin\theta$, ie the area of a parallelogram with sides \mathbf{a} and \mathbf{b} . The area of a triangle with sides \mathbf{a} and \mathbf{b} is half that of the parallelogram. Hence:

$$Area = \frac{|\mathbf{a} \times \mathbf{b}|}{2}$$

```
a = [1; 2; 3];  
b = [3; 2; 1];  
area = norm(cross(a,b))/2
```

area =

```
4.8990
```

Find the equation of a plane containing three points

Find the plane that passes through the three points: $p = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ $q = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$ $r = \begin{bmatrix} 0 \\ 2 \\ 1 \end{bmatrix}$

The lines \overline{pq} and \overline{pr} define two vectors in the plane.

We can use the cross product of these two vectors to create a normal for the plane.

The equation of a plane with normal \mathbf{n} , through a point \mathbf{p} is given by

$$\mathbf{x} \bullet \mathbf{n} = \mathbf{p} \bullet \mathbf{n}$$

We can find \mathbf{n} and $\mathbf{p} \bullet \mathbf{n}$ using MATLAB

```
p = [1; 2; 3];  
q = [1; 0; 1];  
r = [0; 2; 1];  
pq = q - p;  
pr = r - p;  
n = cross(pq, pr)
```

n =

```
    4  
    2  
   -2
```

```
rhs = dot(p, n)
```

rhs =

```
    2
```

The equation of the plane is:

$$\mathbf{x} \bullet \begin{bmatrix} 4 \\ 2 \\ -1 \end{bmatrix} = 2 \quad \text{or} \quad 4x + 2y - z = 2$$

Matrix Transformations

A point (or points) can be transformed by using a square matrix. It is possible to create matrices representing stretches, enlargements, reflections and rotation.

To transform a point we perform a matrix multiplication with the transformation matrix and the point.

This is easily done in MATLAB.

For example, we can easily transform the unit square as follows:

Stretch the x direction by 3, reflect in the y axis and then rotate by 45 degrees.

The transformation matrices required are:

x stretch by 3

$$\mathbf{X} = \begin{bmatrix} 3 & 0 \\ 0 & 1 \end{bmatrix}$$

reflection in y axis

$$\mathbf{Y} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

rotation by 45 degrees

$$\mathbf{R} = \begin{bmatrix} \cos(\pi/4) & -\sin(\pi/4) \\ \sin(\pi/4) & \cos(\pi/4) \end{bmatrix}$$

The MATLAB code to transform the unit square is as follows:

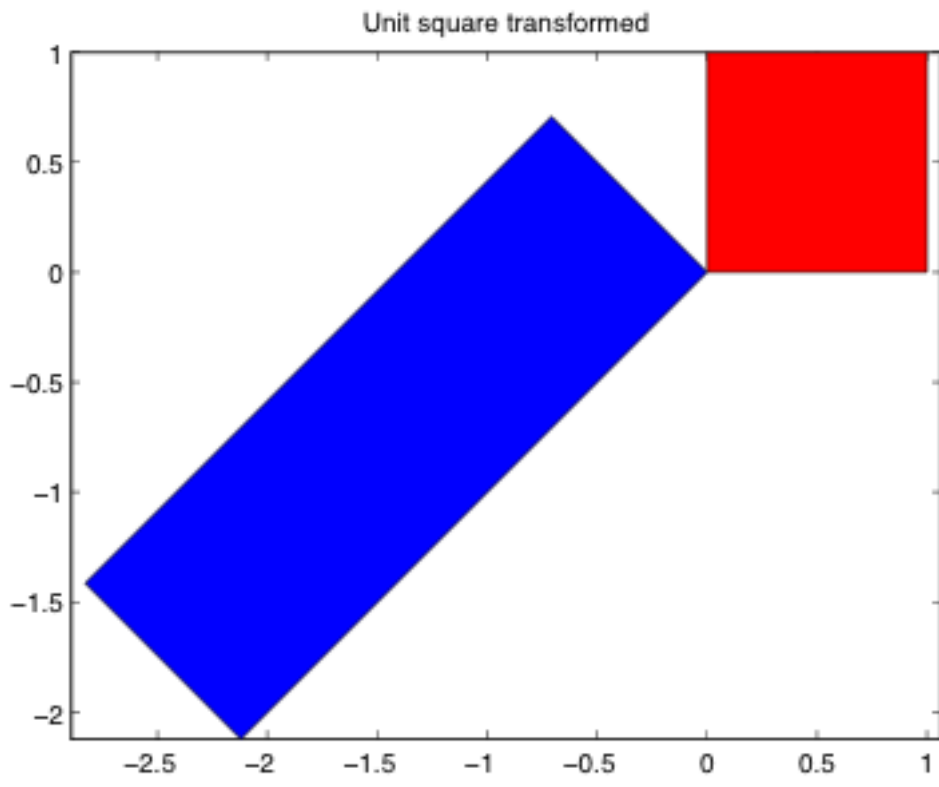
```
% matrix representing the unit square
S = [0 1 1 0; 0 0 1 1]

% plot square
fill( S(1,:), S(2,:), 'r');
title('Unit square transformed');
axis equal

% matrix to stretch x axis by a factor of 3
X = [3 0; 0 1]
% matrix to reflect in y axis
Y = [-1 0; 0 1]
% matrix to rotate by pi/4 radians (45 degrees)
R = [cos(pi/4) -sin(pi/4); sin(pi/4) cos(pi/4)]

% perform transformations
T = R * Y * X * S

% plot transformed square
hold on
fill( T(1,:), T(2,:), 'b');
```

Transition Matrices

MATLAB is particularly useful when working with transition matrices.

Consider the following problem. A taxi company has 200 taxis and studies them travelling between Auckland airport and Auckland city centre. At the start of the week (Monday 9am) it has 60 taxis at the airport and 140 in the city. Over each hour it observes that 20% of the airport taxis come into the city, and 30% of the city taxis move out to the airport.

1. How many taxis are there at the airport after one hour?
2. How many taxis are there at the airport after five hours?
3. What is the steady state distribution of taxis?

First we need the transition matrix for the Markov chain:

$$\begin{array}{l} \text{to airport} \\ \text{to city} \end{array} \quad \begin{array}{cc} \text{from airport} & \text{from city} \\ \mathbf{T} = \begin{bmatrix} 0.8 & 0.3 \\ 0.2 & 0.7 \end{bmatrix} \end{array}$$

How many taxis are there at the airport after one hour?

We need to perform the calculation $\begin{bmatrix} 0.8 & 0.3 \\ 0.2 & 0.7 \end{bmatrix} \begin{bmatrix} 60 \\ 140 \end{bmatrix}$

```
T = [0.8 0.3; 0.2 0.7];  
xinit = [60; 140];  
afterOneHourState = T*xinit
```

```
afterOneHourState =
```

```
90  
110
```

This means that 90 taxis are at the airport and 110 are at the city.

How many taxis are there at the airport after five hours?

We need to perform the calculation $\begin{bmatrix} 0.8 & 0.3 \\ 0.2 & 0.7 \end{bmatrix}^5 \begin{bmatrix} 60 \\ 140 \end{bmatrix}$

```
T = [0.8 0.3; 0.2 0.7];  
xinit = [60; 140];  
afterFiveHoursState = T^5*xinit
```

```
afterFiveHoursState =
```

```
118.1250  
81.8750
```

This means that approximately 118 taxis are at the airport and 82 are at the city.

What is the steady state distribution of taxis?

We need to solve $\begin{bmatrix} 0.8 & 0.3 \\ 0.2 & 0.7 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ ie $\mathbf{T}\mathbf{x} = \mathbf{x}$

$$\mathbf{T}\mathbf{x} = \mathbf{x}$$

$$\mathbf{T}\mathbf{x} = \mathbf{I}\mathbf{x}$$

$$\mathbf{T}\mathbf{x} - \mathbf{I}\mathbf{x} = \mathbf{0}$$

$$(\mathbf{T} - \mathbf{I})\mathbf{x} = \mathbf{0}$$

$$\left(\begin{bmatrix} 0.8 & 0.3 \\ 0.2 & 0.7 \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right) \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} -0.2 & 0.3 \\ 0.2 & -0.3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Note that two equations represented in the matrix are essentially identical (one is a scalar multiple of the other), so we can remove one of them. We also have the constraint that

$$x_1 + x_2 = 200$$

Hence we wish to solve:

$$\begin{bmatrix} -0.2 & 0.3 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 200 \end{bmatrix}$$

$$\mathbf{T} = [-0.2 \ 0.3; \ 1 \ 1];$$

$$\mathbf{b} = [0; \ 200];$$

$$\text{steadyState} = \mathbf{T} \setminus \mathbf{b}$$

$$\text{steadyState} =$$

$$\begin{array}{l} 120 \\ 80 \end{array}$$

Note that another alternative to this method is to simply see what happens in the long term, as markov chains tend to settle down to a steady state fairly quickly. eg we could see what happens after one day:

$$\mathbf{T} = [0.8 \ 0.3; \ 0.2 \ 0.7];$$

$$\mathbf{x}_{\text{init}} = [60; \ 140];$$

$$\text{afterOneDayState} = \mathbf{T}^{24} * \mathbf{x}_{\text{init}}$$

$$\text{afterOneDayState} =$$

$$\begin{array}{l} 120.0000 \\ 80.0000 \end{array}$$

Chapter 10 Summary Programs

We can now write programs to solve linear algebra problems.

For example, the following program finds the intersection of two lines:

$$\begin{aligned}x_1 + 2x_2 &= 1 \\ 2x_1 + 5x_2 &= 0\end{aligned}$$

```
% find the intersection of two lines:
% x_1 + 2x_2 = 1 and 2x_1 + 5x_2 = 0

A = [1 2; 2 5];
b = [1; 0];

if (det(A) == 0)
    message = 'No solution';
else
    x = A\b;
    message = sprintf('Intersection is (%g,%g)',x(1),x(2));
end;

disp(message);
```

An example of this program running is given below:

```
Intersection is (5,-2)
```

The following program finds the area enclosed by three points p, q, r and the angle at point q.

```
% find the area enclosed by three points p, q, r

% define three points in space
p = [1; 2; 2];
q = [1; 2; 3];
r = [0; 0; 3];

% construct a vector from p to q
pq = q - p;
% construct a vector from p to r
pr = r - p;

% angle between pq and pr
theta = acos( dot(pq,pr) / (norm(pq)*norm(pr)) );

% area of triangle
area = norm(cross(pq,pr))/2;

message = sprintf('Angle is %f and area is %f',theta,area);
disp(message);
```

An example of this program running is given below:

```
Angle is 1.15026 and area is 1.11803
```

Chapter 11: Differential Equations and "Function" Functions

Ordinary differential equations arise in many branches of engineering. MATLAB can be used to find numerical solutions to differential equations. This is very useful when you meet a differential equation which is difficult or even impossible to solve analytically. To solve a differential equation we use one of MATLAB's "solver" functions (eg ode45 or ode23). These functions are a little unusual in that one of the inputs into the function is itself a function.

There are several other useful functions which also take a function as an input.

Learning outcomes

After working through this chapter, you should be able to:

- Explain what a "function" function is
- Use fzero to find the root of a function
- Use ode45 to solve a first order differential equation
- Use feval to write your own "function" functions

What is a "function" function?

Until now we have dealt with functions that take one or more variables as inputs. Some special functions take a function as an input (so that they can use that function). There are quite a number of scenarios where a "function" function is useful.

Consider a function that models the velocity of a rocket at any given time. To find the distance travelled by our rocket we would like to be able to integrate our function. Rather than having to write specific code to numerically integrate our function it would be great if we had a function called "Integrate" which could be passed the function to integrate and the limits of integration.

Another common problem is that of finding the roots of a function (ie the input values for the function that give zero). A MATLAB function called fzero can be used to do this.

Finding roots with fzero

The fzero function allows us to find x values for any function $f(x)$, such that $f(x)=0$. These x values are called roots.

Consider the problem of solving the equation: $x^2 - x - 12 = 0$

This can be done by hand but it is also easy to solve using MATLAB.

First we write a function to represent our polynomial

```
function px = MyPolynomial(x)
px = x.^2 - x - 12;
return
```

We can now use the `fzero` function to find the roots of our polynomial close to a given x value. The `fzero` function takes two inputs

- The "address" of the function, so that `fzero` knows where to find the function
- An x value around which we want to search for roots.

For example to look for roots near $x=5$ we use

```
root = fzero(@MyPolynomial, 5)
```

This will produce the following output:

```
root =
     4
```

To find the other root we need to start looking from a different value:

```
root = fzero(@MyPolynomial, -5)
```

This will produce the following output:

```
root =
    -3
```

Notice that our function name has been preceded by the `@` symbol. You will be familiar with the `@` symbol from email addresses. The `@` symbol is used in a similar way here to indicate we are passing the "address" of a function. As a result of passing in the address of `MyPolynomial` to `fzero` it knows where the function "lives" in memory and can call it. The `@` symbol is a good reminder that the input being passed in is another function, not a variable.

An alternative method of passing a function to another function is to pass in the function name as a string:

```
fzero('MyPolynomial', 5)
```

This approach is not quite as obvious as using the `@` symbol. Also if you forget the quotes MATLAB will assume `MyPolynomial` is a variable. It will then attempt to use the contents of the variable `MyPolynomial` as a function, which is likely to generate an error.

You may use either quotes or the `@` symbol when passing functions. For consistency with the MATLAB help files we will use the `@` notation to pass functions into other functions.

Solving ODEs in MATLAB

MATLAB has some very powerful tools for solving ODEs. With very little effort you can find a numerical solution to a complicated first order differential equation, even if it is impossible to solve analytically. In MM2 you will also learn how to use MATLAB to solve second order and higher order differential equations that you meet in engineering problems. It can also be used to solve systems of differential equations such as you will meet in MM3.

In this course we introduce you to using MATLAB to find numerical solutions for ODEs that can be written in the following form:

$$\frac{dy}{dt} = f(t,y)$$

ie the derivative can be written as some function of the independent variable and dependent variable. In many cases the derivative will only depend on one of the variables but MATLAB can handle functions that depend on both.

We will demonstrate how to use MATLAB to solve a simple ODE:

Solving ODEs numerically

In MM1 you were introduced to Euler's method for finding a numerical solution to first order differential equations. In order to use Euler's method you needed three pieces of information:

- A formula for the derivative
- A time interval (start time and finish time)
- An initial value at the start time (initial condition)

The same three pieces of information are needed for any other numerical method. We will need to pass all three pieces of information to our MATLAB solver function. To do this we need:

- A MATLAB function that calculates the derivative for any given values of the independent and dependent variables
- A time span array containing two values (a start time and finish time)
- An initial value

We can then pass these three inputs into one of the MATLAB ODE solvers. The solver will call our derivative function many times to find a sequence of values for the given time span and initial conditions. The output from the solver will be two arrays, one containing values for the independent variable and the other containing the corresponding dependent variable values.

Note that the solvers require the derivative function to take **both** the independent and dependent variables as inputs (even if one or the other may not be used). *The independent variable must be the first input and the dependent the second.* The output for the function must be the derivative for the given inputs.

A simple ODE

Consider the problem of determining the volume of muddy water left in a 1000 litre tank, which began leaking at 1am. We know that the rate of flow out of the tank will be proportional to the volume left in the tank.

$$\frac{dv}{dt} \propto v$$

We also know that as time goes by the mud will gradually coat the hole, reducing the amount of fluid that can flow out. Hence the rate of flow is inversely proportional to time.

$$\frac{dv}{dt} \propto \frac{1}{t}$$

Putting this together we have the following model:

$$\frac{dv}{dt} = -\frac{kv}{t} \quad v(1) = 1000$$

Where k is determined by experiment to have the value $k = 3$

Note that our dependent variable is v as it depends on time. Our independent variable is t .

Recall that the MATLAB solver functions need:

- A MATLAB function that calculates the derivative for any given values of the independent and dependent variables
- A time span array containing two values (a start time and finish time)
- An initial value

It will produce two outputs:

- An array of time values (the independent variable)
- An array of corresponding solution values (the dependent variable)

Representing our ODE with a function

We can now write a function that represents our ODE. It will take as inputs our independent variable and our dependent variable. (**Note the order, the independent variable comes first**).

```
function dvdt = MuddyTankFlowRate(t,v)
% calculate the flow rate out of a muddy tank of water
% inputs: v    the volume of water
%         t    the time since the start of the flow
% output: dvdt the flow rate

k = 3;
dvdt = -k * v / t;
return
```

This function allows us to calculate the flow rate for any given time and volume.

Using a solver

There are quite a number of ODE solvers to choose from: `ode45`, `ode23`, `ode113`, `ode15s`, `ode23s`, `ode23t`, or `ode23tb`.

In general `ode45` is a good solver to try first. The others can be useful if `ode45` does not work well for your problem.

We can now write a short script to solve our ODE

```
% Script to solve the volume of fluid left in a muddy tank
% that contains a hole

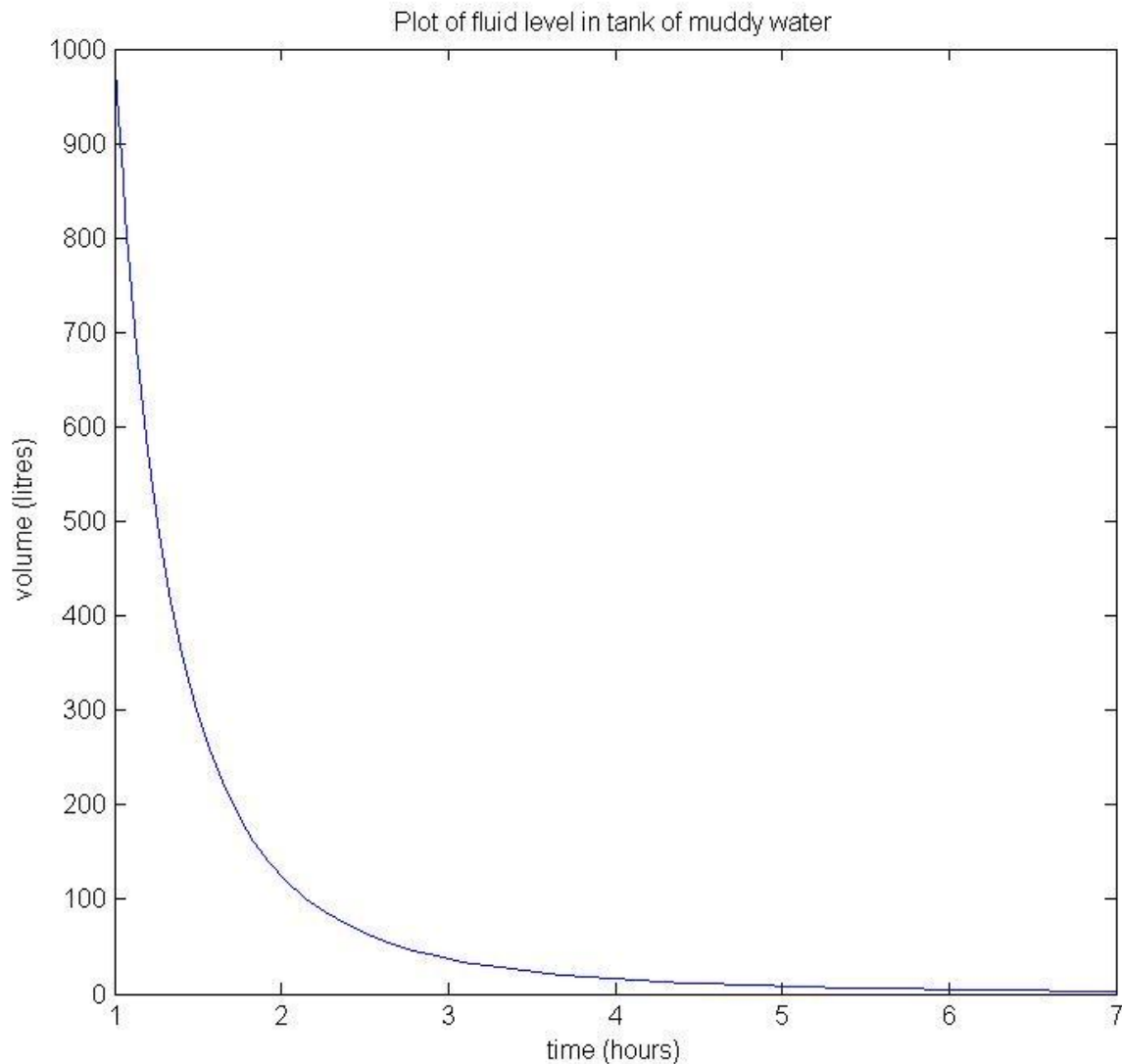
% set up a time span of 6 hours (t is measured in hours)
timeSpan = [1, 7]

% the initial volume at time t=0 is 1000 litres
vInit = 1000

% solve our ODE
[t,v] = ode45(@MuddyTankFlowRate, timeSpan, vInit);

% plot result
plot(t,v)
title('Plot of fluid level in tank of muddy water')
xlabel('time (hours)');
ylabel('volume (litres)')
```

Running our script will produce a plot of the solution



Using feval to write our own function functions

If we want to write our own "function" functions we need a way of being able to call a function that has been passed in as an input. The feval function allows us to do this.

Consider evaluating our polynomial function introduced earlier.

```
function px = MyPolynomial(x)
px = x.^2 - x - 12;
return
```

To find out the value of our polynomial at $x = 2$ we can type:

```
p = MyPolynomial(2)
```

```
p =
    -10
```

Another way of evaluating the function is by using feval

```
p = feval(@MyPolynomial,2)
```

```
p =  
    -10
```

If we want to get a little trickier we can even do the following

```
% create a variable to contain the address of our function  
myFunctionAddress = @MyPolynomial;
```

```
p = feval(myFunctionAddress,2)
```

This gives us a way of evaluating functions that have been stored in variables. With this ability we can now write our own "function" functions.

For example, the following function calculates the derivative of a given input function at a specified point.

```
function dfdx = NumericalDerivative(f,x)  
% calculate the derivative of a function f at a given value x  
% using a central difference to estimate the derivative value  
% ie df/dx is approximated by ( f(x+h) - f(x-h) ) / (2h)  
% inputs: f          the address of the function to differentiate  
%         x          the value to calculate the derivative at  
% output: dfdx the derivative of myFunction at x.  
  
% set h to be a very small value, using the special MATLAB variable  
% eps. We multiply by 1000 so that h is big enough that we get a  
% difference when evaluating to the left and right of x  
h = 1000*eps;  
  
dfdx = ( feval(f,x+h) - feval(f,x-h) ) / (2*h);  
  
return
```

We can use our NumericalDerivative function in the same way that we used fzero:

```
deriv = NumericalDerivative(@MyPolynomial,1)
```

```
deriv =
```

```
    1
```

Chapter 11 Summary Programs

We can now write programs that use functions which take other functions as inputs.

For example, the following function will classify a stationary point.

```
function pointType = ClassifyStationaryPoint(derivative,point)
% classify a stationary point for a function, given the
% derivative of the function
% inputs: derivative      The derivative function which must have
%                        only ONE input value
%           point        x value of the stationary point
% output: pointType      a string describing the type of point

% set up delta, a small positive number
% eps is a special variable which contains a very small
% positive number
delta = 1000 * eps;

% calculate the derivative just to the left of the point
leftDerivative = feval(derivative, point - delta);

% calculate the derivative just to the right of the point
rightDerivative = feval(derivative, point + delta);

if (leftDerivative > 0 & rightDerivative < 0)
    pointType = 'maximum';
elseif (leftDerivative < 0 & rightDerivative > 0)
    pointType = 'minimum';
else
    pointType = 'point of inflection';
end

return
```

The velocity in metres per second of an object is given by $\frac{dy}{dt} = 2t - 1$ $y(0) = 0$

Solve for the displacement for $0 \leq t \leq 2$ and also locate and classify the stationary points of y .

To solve an ODE we need a derivative function that takes TWO inputs (the independent and dependent variables).

```

function dydt = DisplacementDerivative(t,y)
% calculates the value of dy/dt = 2t - 1
% inputs: y      The dependent variable
%         t      The independent variable
% output: dydt  The rate of change dy/dt

dydt = 2 * t - 1;

return

```

To use `fzero` and our `ClassifyStationaryPoint` function we need a function that takes only ONE input

```

function v = Velocity(t)
% calculates the value of v(t) = 2t - 1
% input:  t      The independent variable
% output: v      The value of v
v = 2 * t - 1;
return

```

We can now solve our ODE and classify the stationary point.

```

% This script solves the equation dy/dt = 2t - 1 with initial
% condition y(0) = 0 for the range t = 0 to 2
% it also locates and classifies a stationary point

% start and end value for time interval
timeSpan = [0 2];

% at t = 0, y = 0
yInit = 0;

% solve the ode using the ode45 solver
[t,y] = ode45(@DisplacementDerivative,timeSpan,yInit);

% stationary points occur when the first derivative is zero
% use fzero to locate a stationary point near t=1
% Note we cannot use the DisplacementDerivative function as
% it has two inputs rather than one.
p = fzero(@Velocity,1);

% use our classification function to classify the point
pointType = ClassifyStationaryPoint(@Velocity,p);

% plot the solution
plot(t,y)
% create a nicely formatted title string for the plot
header = sprintf('Solution for y with a %s at %f',pointType,p);
title(header);
xlabel('t (seconds)');
ylabel('y (metres)');

```