MATLAB®: Graphics

Bruce Beckles, Bob Dowling[1]

This is a self-paced course. The demonstrator will get you going and illustrate the various milestones you are expected to reach, but mostly you will be able to work at your own pace. You can ask questions or seek help at any point; simply ask one of the demonstrators.

You are expected to type the MATLAB[2] instructions that appear next to the graphs or screen shots.

Note that to improve readability we have artificially tweaked the thickness of the curves and lines in the pictures (we'll see how to do that later on in the course). We've also increased the font size of the axis labels, which means that the pictures here may show fewer "tick marks" on their axes than what you see on your screen after you've typed in the MATLAB instructions next to the pictures on your computer – if this happens, don't worry about it; unless we explicitly told you to set the tick marks then it doesn't mean you've done anything wrong.

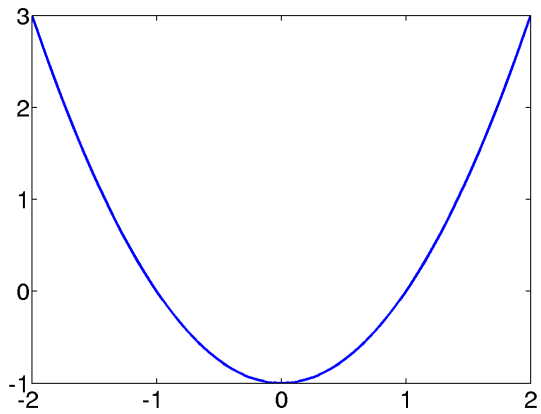
Setting up

If you are doing this course as part of a UCS course then your home directory will have already been set up with various files.

If you are doing this on your own computer you should create a directory to work in. The data file used in the example below can be downloaded from
`http://www-uxsup.csx.cam.ac.uk/courses/MATLAB_GX/data1.dat`.

Recap

We will start with a quick recap of what we think you already ought to be able to do from previous use of MATLAB. The UCS course "MATLAB®: Introduction for Absolute Beginners"[3] teaches you how to launch MATLAB and to create a simple graph of a polynomial function over a specified range, $y=x^2-1$ for $-2 \le x \le +2$, for example.




```
x = linspace(-2, 2, 100);
y = x.^2 - 1;
plot(x, y)
```

Note how we use *array* exponentiation (`.^`) to calculate `y` rather than normal exponentiation (`^`). Since `x` is a vector, we **must** tell MATLAB to use an array operation and so act on each element of the vector. If `x` is a vector, then "`y = x^2 - 1`" will not work as expected.

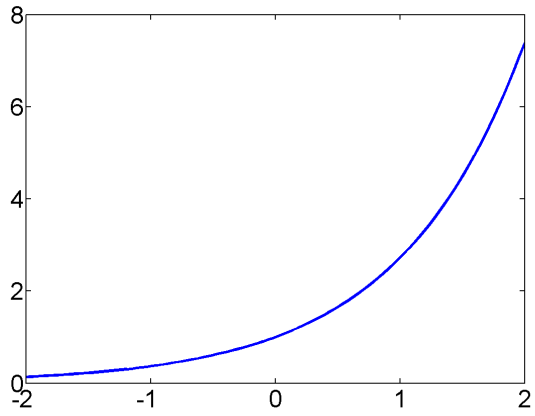
You might also want to plot other functions. If the function you wish to plot will accept a *vector* of points as *input* and return a vector of the function applied to each of those input points (as most of MATLAB's built-in scalar functions do), then this is very simple, as in the example overleaf which uses the `exp` built-in MATLAB function (`exp(x)` calculates e^x).

---

1 This course is based on Bob Dowling's "Mathematica: Graphics" course. Any mistakes, inadequacies, etc. in this course, however, are all my (Bruce Beckles') responsibility.

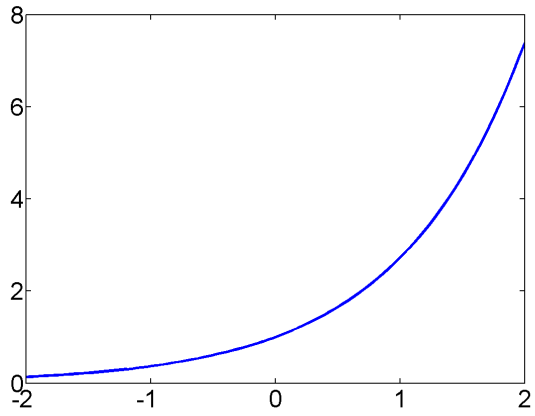
2 MATLAB® is a registered trademark of The MathWorks, Inc.

3 For further details of this course, see:
`http://www.training.cam.ac.uk/ucs/course/ucs-matlab`




```
x = linspace(-2, 2, 100);
y = exp(x);
plot(x, y)
```

Of course, the function you want to plot may not be that well behaved – in this case you can use MATLAB's built-in **`arrayfun`** function to help you. You can use `arrayfun` with any function that accepts scalar input and returns scalar output, whether or not it also accepts vector input. `arrayfun` takes a function and applies it to each element of the input matrix it has been given. It returns an output matrix of the same size as the input matrix whose elements are the result of applying the function to the corresponding element of the input matrix. Recall that we must precede the function's name with "@" to make it into a *function handle* so that we can pass it to `arrayfun` as input.




```
x = linspace(-2, 2, 100);
y = arrayfun(@exp, x);
plot(x, y)
```

Finally, you might want to repeat a sequence of commands a certain number of times. You can do this with a **`for`** loop, as shown below:

```
for n = 5:0.1:5.5
            disp(n)
end
```

Note the syntax for the **`for`** loop (shown overleaf):

```
for "loop variable" = start:step-size:finish
    MATLAB command
    another MATLAB command
    …and so on
end
```

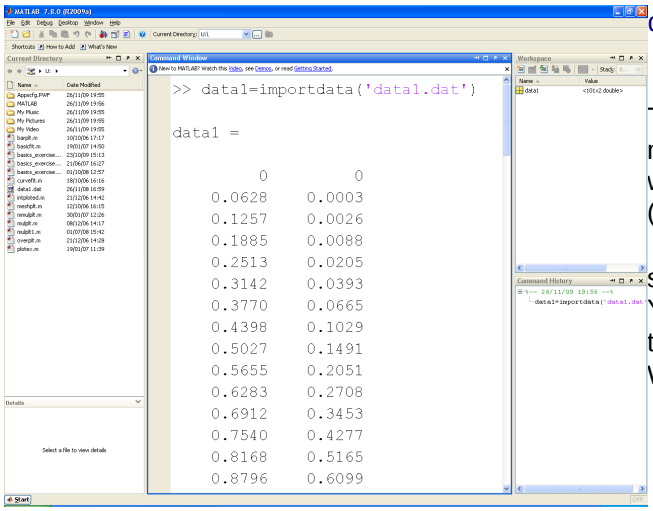
The "***loop variable***" is the name of a variable MATLAB should use as a counter for the loop. On the first iteration of the loop, it will be set equal to ***start***, and then gradually increase by ***step-size*** on each iteration until it reaches ***finish***. The loop stops as soon as the counter is greater than *finish*. (Note that *step-size* can be negative.) The keyword **end** indicates that we have finished the commands that should be executed on each iteration of the loop. You can get help on `for` loops by typing "`help for`" in MATLAB's Command Window.

These four simple procedures are all you need to start on the course.

Plotting external data

Of course, you may not want to plot a simple function. Your data may be imported from some other source. MATLAB can import matrices or vectors of data points and plot them.

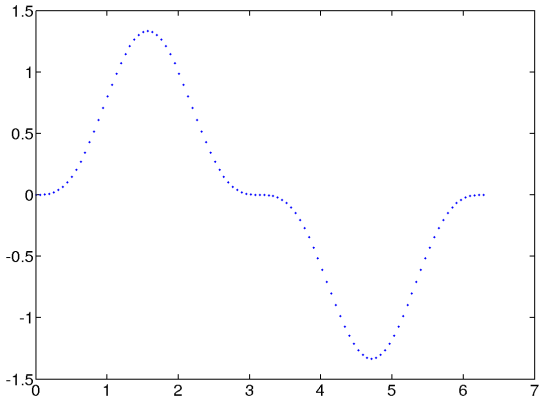
For the purposes of this course we have created some data files which you unpacked as part of the setting up. The one we want for now is called "`data1.dat`". The function to import the data is called "**`importdata`**":




```
data1 = importdata('data1.dat')
```

The `importdata` function understands the format of many different file types, and it primarily works out what type of file it is importing by the file's extension ("`.dat`" in this case). MATLAB expects files ending in `.dat` to be text files containing delimited (i.e. separated by a given sequence of characters) data. You can get help on the `importdata` function by typing "`help importdata`" in MATLAB's Command Window.

`data1` is a MATLAB matrix. It is a collection of (x,y) points, with the x points in the first column of the matrix, and the corresponding y points in the second column. We can tell MATLAB to plot the points on a graph with the `plot` function:

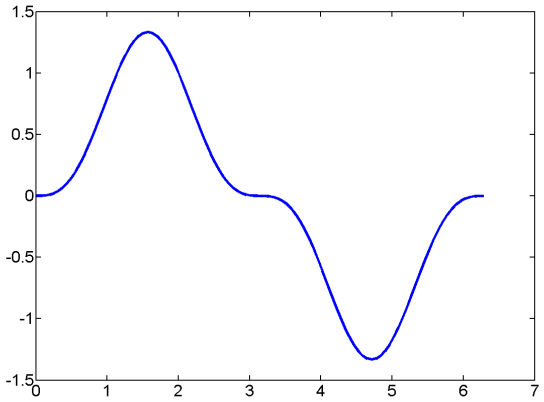



```
plot(data1(:,1), data1(:,2), '.')
```

Note how we give `plot` the x points as a column vector (the first column of `data1`) and the y points as another column vector (the second column of `data1`).

We also tell `plot` that instead of joining the points up as it normally does, it should just plot each point. We do this by giving an optional *line specification*, as a string. This particular string, `'.'`, tells `plot` to just plot the points using a "point" as a *marker* for each point, but not to join them up. We will meet the full range of line specifications shortly.

Contrast this with using `plot` on `data1` without specifying a line specification:



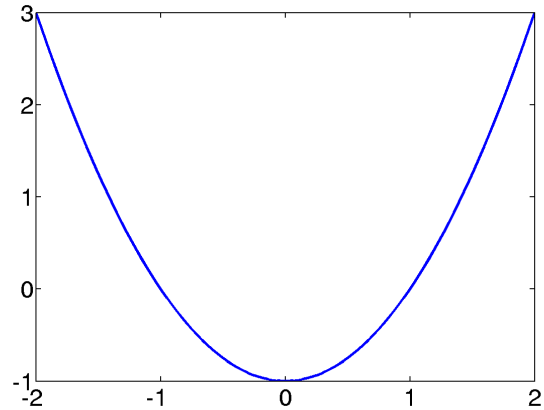

```
plot(data1(:,1), data1(:,2))
```

Once again, note how we give `plot` the x points as a column vector (the first column of `data1`) and the y points as another column vector (the second column of `data1`).

Exporting graphs

As well as importing data to create graphs from we may want to export our graph for use elsewhere. We can do this with the **`print`** command. The easiest way to do this is to first create the graph in a Figure window and then export it using **`print`**.

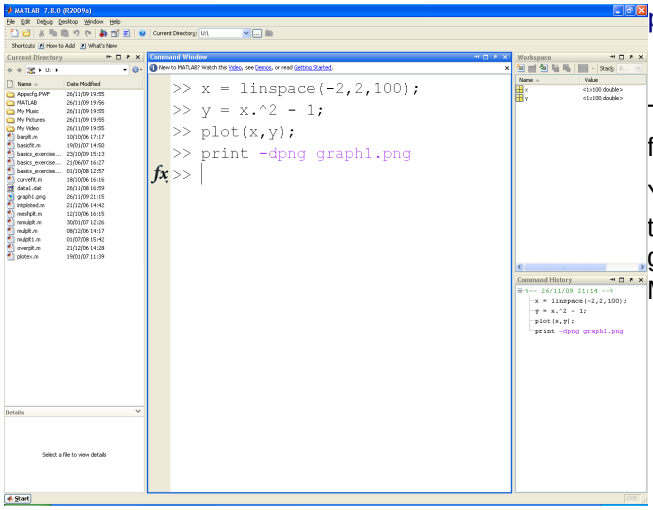
So first we create the graph:




```
x = linspace(-2, 2, 100);
y = x.^2 - 1;
plot(x, y);
```

This uses the function $y=x^2-1$ we plotted previously.

and then we export it as a Portable Network Graphics (PNG) graphics file[4]:

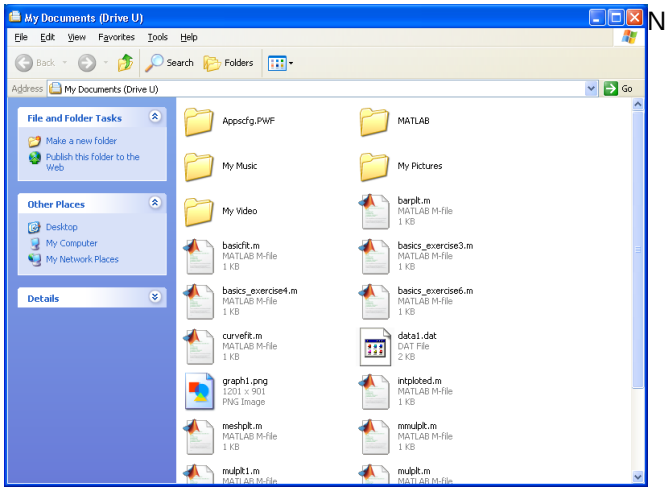



```
print -dpng graph1.png
```

The `-dpng` tells `print` to export the figure as a PNG file.

You can find out more about the `print` command and the different graphic formats to which it can export graphical objects by typing "`help print`" in MATLAB's Command Window.

You should now have the corresponding graphics file in your current directory.



Note the existence of "`graph1.png`".

All of this course's practicals will consist of creating a graph of some form or another and then exporting it as a file. To make sure you can export a graphics file, our first practical will be to do just that.

Practical 1

1. In MATLAB create a graph of $y=x^3$, $-1 \le x \le 1$.
2. Export it to a file called "`practical1.png`" in your current directory.
3. To check that it has worked, ***minimise*** MATLAB and then navigate to the directory in the file system and check you have a file called "`practical1.png`". Double-click on it to see the graph in all its glory.

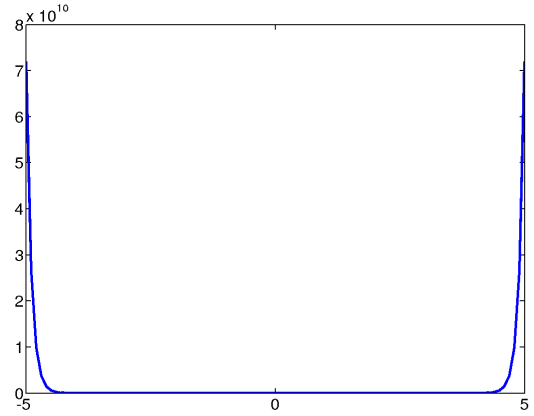
Taking control of the graph

Next we will fine-tune exactly what the graph looks like. The `plot` function as we have used it thus far takes two arguments, the x- and y-values to be plotted (although we have seen there can be an optional third string argument giving the line specification). MATLAB then derives the x- and y-ranges that it thinks are appropriate and selects its own tick marks, colour scheme, etc. In this section we will see how to take explicit control of these settings.

We will start by adjusting the axes, as MATLAB's choices are not always ideal.

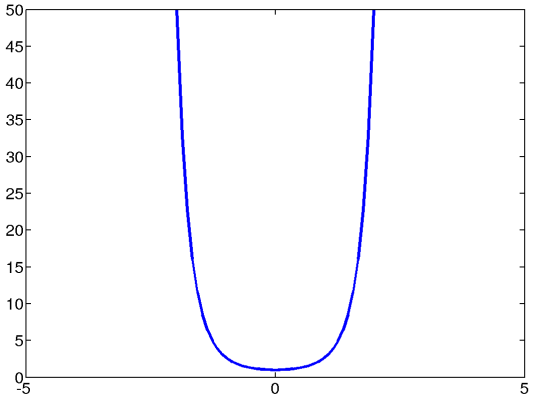
4 If you have not come across PNG files before, see the Wikipedia entry on them for more information:
`http://en.wikipedia.org/wiki/Portable_Network_Graphics`

For example, consider the function $y=e^{x^2}$ for $-5 \le x \le +5$:



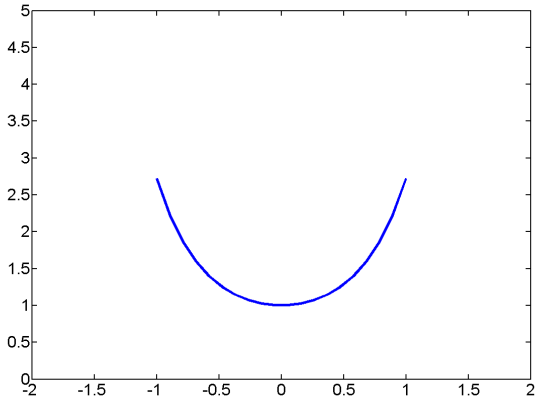

```
x = linspace(-5, 5, 100);
y = exp(x.^2);
plot(x, y)
```

We can take explicit control of the axes by defining the plotted x- and y-ranges with **`axis`**. For instance, we can give an exact range with **`axis([`x_{min} x_{max} y_{min} y_{max}`])`**, as in the following example:




```
x = linspace(-5, 5, 100);
y = exp(x.^2);
plot(x, y);
axis([-5 5 0 50])
```

What is the point of having to specify the x-range to be calculated ("`x = linspace(-5,5,`…") and the x-range to be plotted ("`axis([-5 5 …`")? We may want to float a graph for $-1 \le x \le 1$, say, in a set of axes running from $-2 \le x \le 2$ and by specifying the x-range in two separate places we can do this easily:




```
x = linspace(-1, 1, 20);
y = exp(x.^2);
plot(x, y);
axis([-2 2 0 5])
```

`axis off` turns off the axes altogether (**`axis on`** restores the axes). If you have manually set the x- and y-ranges, you can return to the default setting (MATLAB works it out for you) using **`axis auto`**. Also, if you want MATLAB to determine one of the limits of the x- or y-ranges you can just substitute **`Inf`** (for an upper

limit) or **-Inf** (for a lower limit), e.g. `axis([-2 2 0 Inf])` will use the specified x-range, start the y-range at 0, and allow MATLAB to determine the upper limit of the y-range. Try **`axis([-2 2 0 Inf])`** and see how the graph changes before continuing.

There are a few other commands and functions that can set properties of the graph, detailed below. You should try these out for yourself before moving on:

`box on` puts a box around the axes (the default), and **`box off`** removes it. (This only has any effect if the axes haven't been turned off.)

`set(gca,'XTick',[-1 0 1])` sets where the "tick marks" on the x-axis are. The vector you use to specify the tick marks must be in increasing order of value, but the values do not need to be equally spaced, so `set(gca,'XTick',[-1 0 0.5])` would work. If you want to go back to default setting (i.e. MATLAB works it out for you), use **`set(gca,'XTickMode','auto')`**.

`set(gca,'YTick',[1.0 1.5 2.0 2.5 3.0])` sets where the "tick marks" on the y-axis are. The vector you use to specify the tick marks must be in increasing order of value, but the values do not need to be equally spaced, so `set(gca,'YTick',[1.0 2.0 3.5])` would work. To return to the default setting, use **`set(gca,'YTickMode','auto')`**.

(The **`set`** function allows us to manipulate all the properties of the graphics objects that MATLAB creates. In order to use `set` we have to specify the object whose properties we wish to modify as the first argument we give to `set`. **`gca`** returns a special sort of handle (that `set` understands) that identifies the current axis in the current figure. Hence `set(gca,'XTick', ...)` modifies the **`XTick`** property of the current axis, which controls where the tick marks are placed on the x-axis.)

`set(gca,'XTickLabel',{'x1', 'x2', 'x3'})` specifies the text to be used as the *labels* for the tick marks on the x-axis. The labels are arranged starting from the lowest valued tick mark to the highest value tick mark. If you don't specify enough labels for all the tick marks then MATLAB starts from the first label again. (Note the use of curly braces **`{}`** to surround the collection of *text* labels.) To return to the default labels MATLAB gives you, use **`set(gca,'XTickLabelMode','auto')`**.

`set(gca,'YTickLabel',{'1.0', '1.5', '2.0', '2.5', 'almost pi'})` specifies the text to be used as the labels for the tick marks on the y-axis. The labels are arranged starting from the lowest valued tick mark to the highest value tick mark. If you don't specify enough labels for all the tick marks then MATLAB starts from the first label again. (Note again the use of curly braces `{}` to surround the collection of text labels.) To return to the default labels, use **`set(gca,'YTickLabelMode','auto')`**.

`grid on` displays a grid of lines across the graph (**`grid off`** removes it again). You can also make the grid more or less fine-grained with **`grid minor`** – be careful not to swamp the actual curve. (`grid minor` is a toggle setting, which only has any effect when the grid is displayed. The default is to show a more coarse-grained grid that only uses the major grid lines. `grid minor` switches between this default state and showing the finer-grained minor grid lines as well.) Note that the grid will not be shown if the axes have been turned off.

Next we turn our attention to the descriptive text and legend we might want for a graph. You should try these out for yourself before continuing.

`title('Title of graph')` specifies the text to be used as the title of the graph. It will be centred at the top of the current set of axes. You can find out more about what you can do with `title` by typing "`doc title`" in the Command Window.

`legend('Data series')` specifies the text to be used in the legend of the graph. It will be placed just inside the top right corner of the current set of axes. **`legend hide`** will hide the legend and **`legend show`** makes the legend visible again (if you haven't already created a legend `legend show` creates one for you using some default text). **`legend boxoff`** makes the background box around the legend disappear (when the legend is shown), whilst **`legend boxon`** makes the background box around the legend visible (when the legend is shown). If you want to completely destroy the legend (rather than just temporarily hiding it), use

`legend off`. You can find out more about what you can do with `legend` by typing "`doc legend`" in the Command Window.

`xlabel('x-axis label')` specifies the text to be used to label the x-axis. Note that this is the label of the x-axis itself, ***not*** the tick marks on the axis. It will be placed beside the x-axis. You can find out more about what you can do with `xlabel` by typing "`doc xlabel`" in the Command Window.

`ylabel('y-axis label')` specifies the text to be used to label the y-axis. Note that this is the label of the y-axis itself, ***not*** the tick marks on the axis. It will be placed beside the y-axis. `ylabel` takes all the same options as `xlabel`, as can be seen by typing "`doc ylabel`" in the Command Window.

(Similarly, **`zlabel`** is used to label the z-axis for 3 dimensional graphs (which we haven't met yet). It takes the same options as `xlabel` and `ylabel`, as can be seen by typing "`doc zlabel`" in the Command Window.)

`text(0.0, 1.0, 'some text')` specifies that some text be placed at the specified x- and y co-ordinates of the current graph, (0.0, 1.0) in this case. You can find out more about what you can do with `text` by typing "`doc text`" in the Command Window.

Next we turn our attention to the curve itself. We will want to modify the nature of the curve itself especially when we have multiple curves on the same axes but for now we will focus on a single curve.

The nature of the curve is most usefully controlled by the optional line specification argument to the `plot` function. (You can read about the values this line specification argument can take in MATLAB's Help browser by typing "`doc LineSpec`" in the Command Window, but for convenience the values are summarised below.)

The line specification argument is a string made up of three parts, any of which can be omitted, and which can be given in any order. Each part controls a different aspect of the curve as follows:

Line Style:

Specifier	Line Style
`-`	solid line (the default)
`--`	dashed line
`:`	dotted line
`-.`	dash-dot line

Marker Specifiers:

The marker specifiers control what sort of *markers*, if any, are used to indicate the points of the curve on the plot. The default is not to indicate the individual points on the curve but instead to join them using a solid line. You can use both a marker specifier and a line specifier together which will produce a curve whose individual points are indicated by the specified marker and are joined together by a line of the specified style. If a marker specifier is used **without** a line specifier then the points on the curve will **not** be joined by a line.

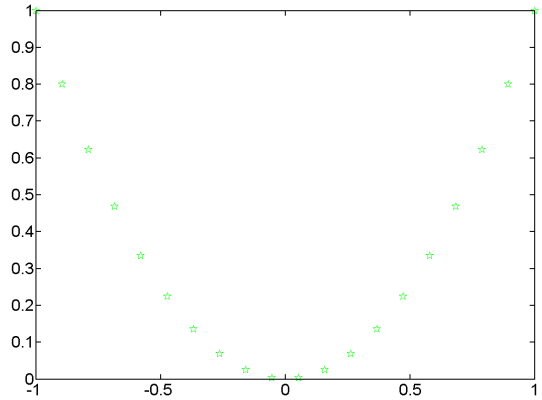
Specifier	Marker Type
`+`	plus sign (+)
`o`	circle
`*`	asterisk (*)
`.`	point
`x`	cross
`s` (*or* `square`)	square
`d` (*or* `diamond`)	diamond

Specifier	Marker Type
^	upward-pointing triangle
v	downward-pointing triangle
>	right-pointing triangle
<	left-pointing triangle
p (*or* `pentagram`)	5-pointed star (pentagram)
h (*or* `hexagram`)	6-pointed star (hexagram)

Colour:

Specifier	Colour
r	red
g	green
b	blue
c	cyan
m	magenta
y	yellow
k	black
w	white

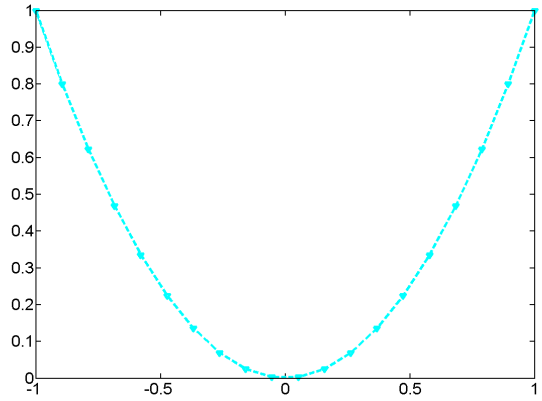
At most one specifier from each of the three types of specifier above can be mixed in any combination to produce truly bewildering styles of curves. For example:



```
x = linspace(-1, 1, 20);
y = x.^2;
plot(x, y, 'gpentagram')
```

Note that exactly the same plot is produced if we use `plot(x, y, 'pg')` instead – the order in which the specifiers are listed (and whether the long or short form is used for those specifiers, like `'pentagram'`, that have long and short forms) is irrelevant.

Or:



```
x = linspace(-1 , 1, 20);
y = x.^2;
plot(x, y, 'vc--')
```

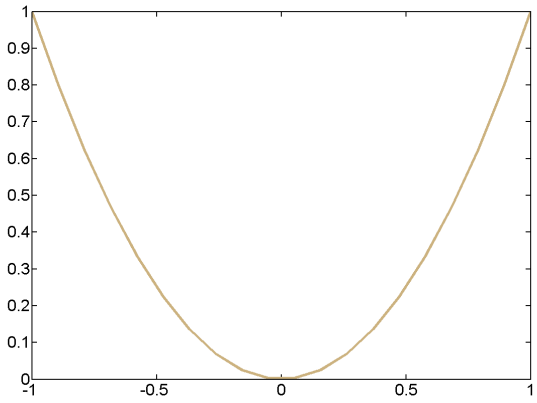
You should experiment with different combinations of these specifiers before moving on; they are easily the most commonly used way of controlling MATLAB's graphs.

What if we want to specify a colour that does not have a colour specifier? How can we do this? The `plot` function can take additional optional arguments that specify other properties of the plot (such as the colour of the line). These additional arguments are specified after the line specification argument, if this is being given. These arguments are specified in pairs: the first argument is the name of the property (as a string), the second argument is its value).

The colour of a line is controlled by the **`Color`** property, whose value is a 3 item row vector that specifies the RGB value of the colour. The first element of the row vector specifies the intensity of the red component of the colour, the second the intensity of the green component, and the third the intensity of the blue components. The intensity of each component must be a number between 0 and 1. (Some people may be more familiar with the computing habit of using three ranges of integers from 0 to 255 for this purpose.) Below is a table listing the RGB values of the colours for which MATLAB has colour specifiers:

RGB vector	Colour	MATLAB short name	MATLAB long name
`[1 0 0]`	red	`r`	`red`
`[0 1 0]`	green	`g`	`green`
`[0 0 1]`	blue	`b`	`blue`
`[0 1 1]`	cyan	`c`	`cyan`
`[1 0 1]`	magenta	`m`	`magenta`
`[1 1 0]`	yellow	`y`	`yellow`
`[0 0 0]`	black	`k`	`black`
`[1 1 1]`	white	`w`	`white`

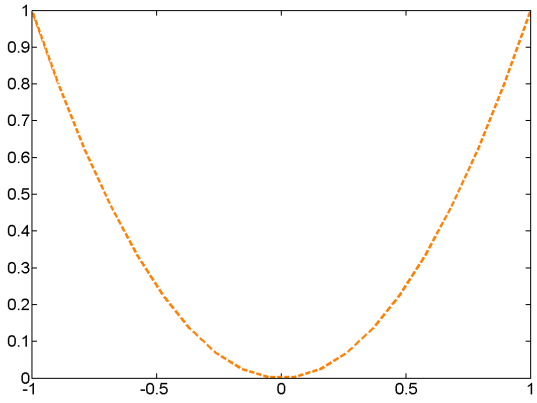
So, for example:




```
x = linspace(-1, 1, 20);
y = x.^2;
plot(x, y, 'Color', [0.8 0.7 0.5])
```

(Apparently, this colour is some sort of tan.)

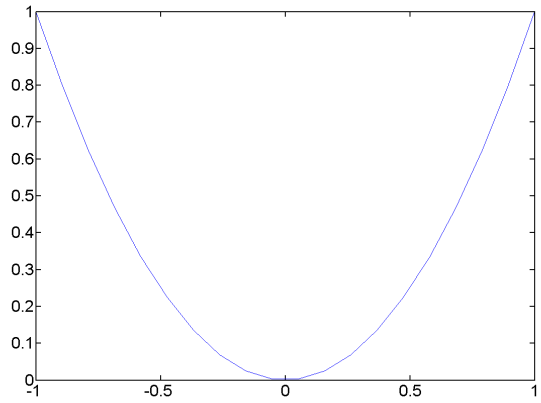
Or:




```
x = linspace(-1, 1, 20);
y = x.^2;
plot(x, y, '--', 'Color', [1 0.5 0])
```

(Apparently, this colour is coral.) Note how we use a line specification as well as setting the colour via the `Color` property.

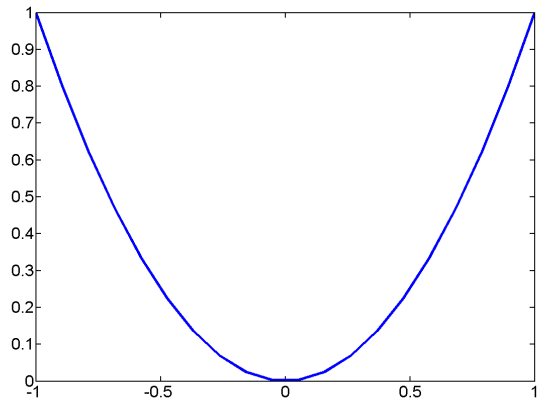
Suppose we now want to set the thickness of the line of the curve. (We have already been using this behind the scenes to give visible lines for this set of notes because whilst the default line thickness often looks fine on the screen it often doesn't look so good on the laser-printed page.) The thickness of the line drawn for a graph is controlled by the **`LineWidth`** property. The `LineWidth` gives the width or thickness of the line in points (1 point = $\frac{1}{72}$ inch). The default `LineWidth` is 0.5 points. Unless otherwise stated, the graphs shown in this handout all have a `LineWidth` of 4.0 points.




```
x = linspace(-1, 1, 20);
y = x.^2;
plot(x, y)
```

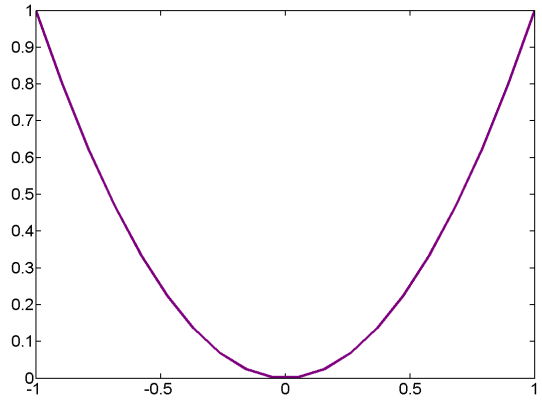
(This graph really was plotted with the default `LineWidth` of 0.5 points – you can see how difficult it is to see the line on the printed page.)

So here is a graph explicitly plotted with a `LineWidth` of 4.0 points:




```
x = linspace(-1, 1, 20);
y = x.^2;
plot(x, y, 'LineWidth', 4.0)
```

But we don't want the thickness setting to get in the way of the colour. How can we combine the results into a single graph? We do this by simply telling `plot` all the properties whose values we want to set (we list these arguments after we have given it the (x,y) values to plot and any line specification argument we may wish to use).



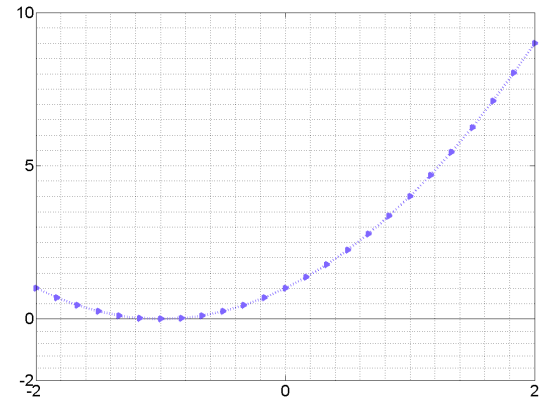

```
x = linspace(-1, 1, 20);
y = x.^2;
plot(x, y, 'Color', [0.5 0 0.5],
'LineWidth', 4.0)
```

(This colour is a light magenta.)

Note that it doesn't matter whether we specify the `Color` property and its value first or the `LineWidth` property and its value first; `plot` doesn't care.

Note that there are other properties which can be set that affect the graph which we have not mentioned, but the ones covered here are the most commonly useful.

Practical 2

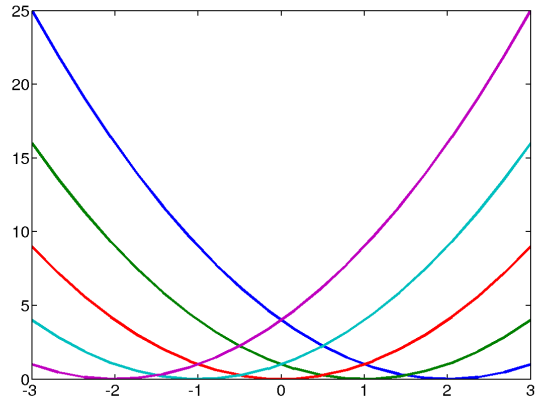


Work out the MATLAB commands necessary to create this graph. The curve is $y=(x+1)^2$. You don't need to get the colour exactly the same, although the colour of your graph should be similar.

(Note that if you are looking at a printed copy of these notes then depending on the printer used to print them out, it may appear that there is a **solid** horizontal line running across the graph around $y=0$. This is a printing artefact and should be ignored.)

Multiple graphs

Next we will plot multiple curves on the same set of axes. There are various ways we can do this. The first, and easiest, way to do this is to create a matrix of the y-values of the functions we want to plot and then plot the matrix with `plot`.



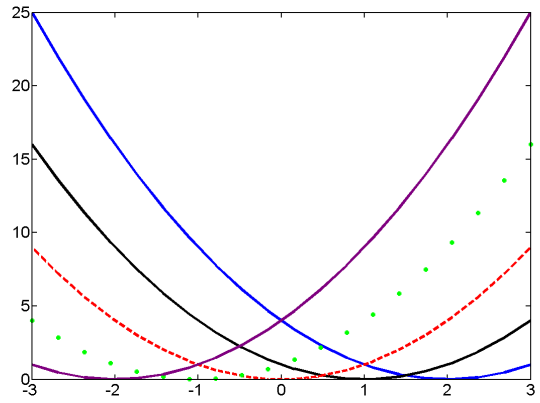

```
x = linspace(-3, 3, 20);
VALUES = [(x - 2).^2 ; (x - 1).^2 ; x.^2 ; (x + 1).^2 ; (x + 2).^2];
plot(x, VALUES)
```

Note that it is vital to separate the different functions in our matrix with a semi-colon (`;`), so that we end up with a matrix each of whose *columns* contains the y-values for a particular function.

MATLAB has default colours for each graph but we may want to change them, possibly use different line specifications for each graph, etc. There are various ways of doing this, but we're just going to look at the most versatile way of doing this here. This involves using multiple plotting functions and having them all display their output in a single figure.

Normally each time you call a function that plots something it clears the current figure. You can stop this behaviour with **`hold on`**. **`hold off`** will return to the default behaviour of clearing the current figure between plots.

By doing this we can completely customise each curve to our heart's content since we just plot the curve with whatever customisations we want as we normally would, and then go on to plot the next curve, and so on.

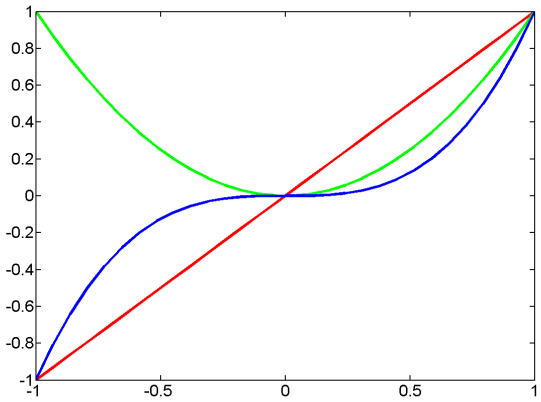



```
x = linspace(-3, 3, 20);
y1 = (x - 2).^2;
y2 = (x - 1).^2;
y3 = x.^2;
y4 = (x + 1).^2;
y5 = (x + 2).^2;
plot(x, y1);
hold on;
plot(x, y2, 'k');
plot(x, y3, 'r--');
plot(x, y4, '+g');
plot(x, y5, 'Color', [0.5 0 0.5])
hold off
```

Remember to use `hold off` when you've finished plotting your curves on the same graph.

Note that MATLAB adjusts the axes automatically for us so that all the curves can fit on the plot. If we don't want this, or if MATLAB's auto-scaling proves to be sub-optimal, then we should set the ranges ourselves with `axis`.

Practical 3



Reproduce this graph and save it as a file `practical3.png` in your current subdirectory.

The graph has curves for $y=x$ in red, $y=x^2$ in green and $y=x^3$ in blue.

Animation

Using `hold on` merges various graphs into a single plot. An alternative is to animate the sequence. We can do this by plotting each frame of the animation in turn and then capturing it with **`getframe`**. We store each frame in an array and then use **`movie`** to generate an animation of them in sequence.

When creating an animation it is even more important to set the plot ranges explicitly. If you don't then the axes of each graph will take effect as that graph is shown and they will wobble up and down as the animation proceeds. For example, consider the following animation:

```
frame = 1;
x = linspace(-3, 3, 20);
for n = -2:0.1:2
            y = (x - n).^2;
            hold off;
            plot(x, y);
            axis([-3 3 0 25]);
            M(frame)=getframe;
            frame = frame + 1;
end
movie(M)
```

Now try the above instructions again but *without* using the `axis` instruction and keep an eye on the y-axis.

`movie` takes an optional argument which tells it how many times to play the animation, e.g. in the above example `movie(M, 3)` would play the animation 3 times. If the number of times specified is *negative*, then each time the animation is played it will play once forward, and then once backward, so `movie(M, -2)` would play the animation four times in total: once forward, then once backward, then forward again, and finally backward.

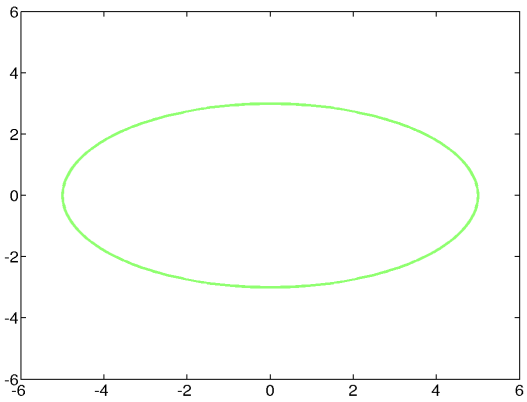
MATLAB supports exporting animations to the AVI file format[5], but unfortunately it doesn't support very many types of AVI file. On UNIX/Linux (including MacOS X), MATLAB only supports exporting to uncompressed AVI files which, as you might guess from the word "uncompressed", are extremely large. On Windows it supports a few more options, but exactly which ones depend on what video compressors (the jargon term for a video compressor is "*codec*") have been installed on the system. The command you use to export an animation is **`movie2avi`** – for further details type "`help movie2avi`" in the Command Window.

Note though, that the default setting for `movie2avi` for MATLAB under PWF Windows produces an AVI file that Windows Media Player doesn't seem able to play. You can produce an uncompressed AVI file of the animation above, in a file called `test.avi`, that Windows Media Player *can* play (but which is very large) using the following command:

```
movie2avi(M, 'test.avi', 'compression', 'None')
```

Implicit functions

Sometimes we do not have y as an explicit function of x, but rather an equation relating the two values. For example, suppose we wanted to plot the curve satisfying $x^2/25+y^2/9 = 1$. We will use a MATLAB command called **`contour`** to do this. Note that we first have to create a **`meshgrid`** from the x- and y-ranges. Note also how we specify that we want to plot the values that satisfy `z` = 1 by saying we want the *contour* for the level "1" (by specifying the row vector `[1 1]`).

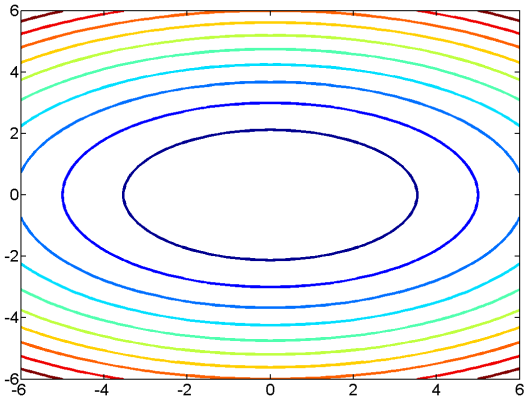



```
t = linspace(-6, 6, 100);
[x, y] = meshgrid(t, t);
z = (x.^2)/25 + (y.^2)/9;
contour(x ,y ,z, [1 1])
```

Note that we specify that we want the contour v by using the row vector $[v\ \ v]$. This is slightly odd, and we'll see the reason for it shortly. (If we want to specify more than one contour, the syntax is what you would expect: for the contours v_1 and v_2, we would use the row vector $[v_1\ \ v_2]$; for the contours v_1, v_2 and v_3 we would user the row vector $[v_1\ \ v_2\ \ v_3]$; and so on.)

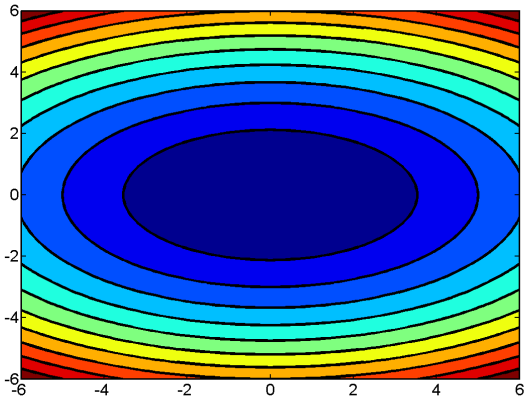
If we don't explicitly specify any contours then MATLAB will choose them for us, as shown overleaf:

5 If you have not come across AVI files before, see the Wikipedia entry on them for more information:
`http://en.wikipedia.org/wiki/Audio_Video_Interleave`




```
t = linspace(-6, 6, 100);
[x, y] = meshgrid(t, t);
z = (x.^2)/25 + (y.^2)/9;
contour(x ,y ,z)
```

If we want a filled contour plot, we need to use the function **`contourf`** instead of `contour`. Its syntax is the same as that of `contour`.



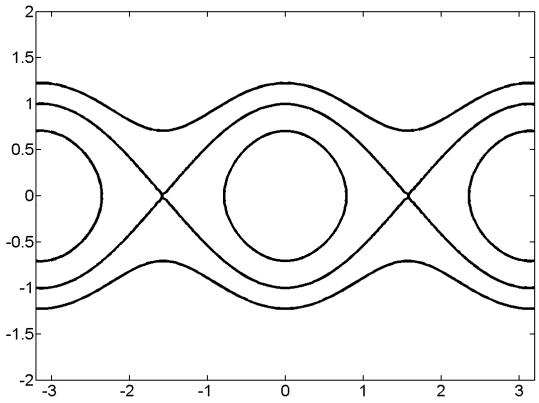

```
t = linspace(-6, 6, 100);
[x, y] = meshgrid(t, t);
z = (x.^2)/25 + (y.^2)/9;
contourf(x ,y ,z)
```

Note that MATLAB will select the values to plot contours for. `contour` (and `contourf`) take an optional argument to take explicit control of this in one of two ways. We've already seen that if we use a row vector of the form `[1 2 3]` then MATLAB will plot 3 contours matching the values 1, 2, and 3. If, instead of a row vector, we use a single positive integer then MATLAB will plot that many equally spaced contour lines (*try this out for yourself*). This is the reason that if we want to plot a single contour matching a specific value, v, we need to specify it as `[v v]`: MATLAB can't tell the difference between `[v]` and v so if v is an integer then it will treat `[v]` as the *number* of contour lines it should produce rather than the single *value* for which we want a contour.

We can also give `contour` and `contourf` the same line specification argument that we use with `plot`, and we can also set additional properties of the graph (`Color`, `LineWidth`, etc) in the same way. Note that the line specification argument and any additional property arguments come *after* the argument to control the values or numbers of contours plotted, if given.

Successive contour plots can be collected together and animated just as we did with the output of successive `plot` commands earlier.

Practical 4



Plot the graph shown and save it to a file `practical4.png` in your current directory.

The three curves are

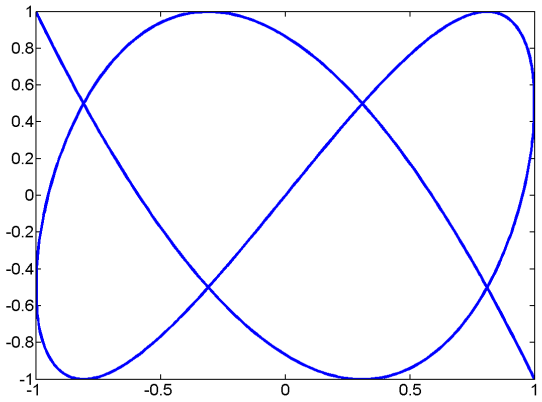
$y^2 + \sin^2 x = 1/2$
$y^2 + \sin^2 x = 1$
$y^2 + \sin^2 x = 3/2$

There is more than one way to do this. You only need one.

Parametric curves

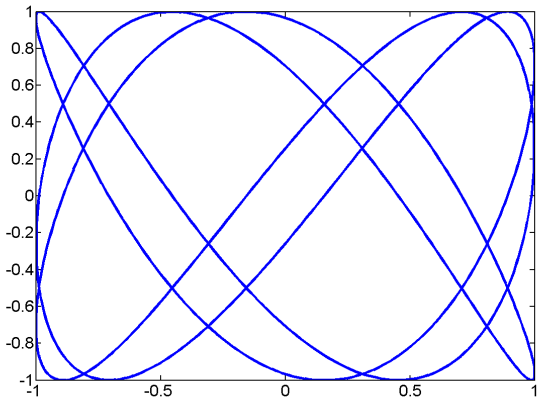
As well as implicit functions a very common way to plot a two-dimensional curve is parametrically. In this approach we define the x and y coordinates of points as explicit functions of a third variable whose value changes along the length of the curve.

Consider for example the classic Lissajou figure given by $x = \sin 3t$, $y = \sin 5t$, for $0 \le t \le 2\pi$. In MATLAB we plot this with the plot function exactly as we would a normal equation. (Note, though, that because this curve bends a lot more and crosses itself much more than other curves we have previously plotted that we need to plot many more points to get a smooth curve.)




```
t = linspace(0, 2*pi, 1000);
x = sin(3*t);
y = sin(5*t);
plot(x,y)
```

Practical 5



(a) Plot the parametric equation

$x = \sin(3t+\pi/20)$
$y = \sin(5t)$

for $0 \le t \le 2\pi$. Save the graph to `practical5.png` in your current directory.

(b) Use a `for` loop to create a list of forty plots for the parametric equations

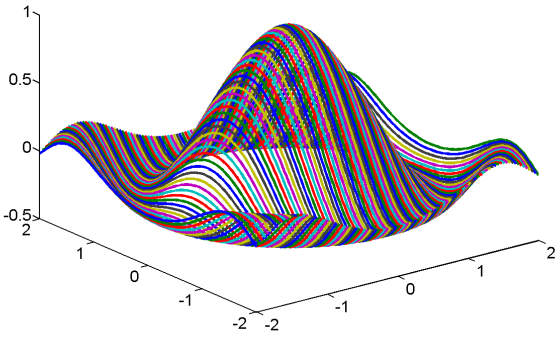
$x = \sin(3t+d)$
$y = \sin(5t)$

each plotted for $0 \le t \le 2\pi$ with d taking values {0, π/20, π/10, ... 39π/20} and animate them.

Explicit three dimensional plots

So far all our graphs have been two dimensional. Now we will move on to three dimensions. However, this mostly consists of learning the 3D versions of the 2D functions. The naming convention is simple: take the name of the 2D function and add "3" to the end of it. Also, whilst for the 2D functions we supplied the (x,y) points to plot, for the 3D functions we will supply (x,y,z) points instead. This is straightforward if we want to plot a *single line* in three dimensions (as we will see later). If, however, we want to plot in three dimensions a set of lines that describe a 2-dimensional *surface* (or if we want to plot the surface itself), then MATLAB needs us to first create a **`meshgrid`** of (x,y) points and then to calculate z at those points. Finally, a very important point before we go any further is that when we are plotting surfaces we must not try and plot *too many* points too close together or we will end up with a solid outline of the surface (often in black), which is probably not what we want!

The simplest 3D graphs are "height graphs" where we provide a function specifying z as an explicit function of x and y. Having first prepared our points with `meshgrid`, we can use the **`plot3`** function to draw these graphs (see overleaf).




```
t = linspace(-2, 2, 100);
[x, y] = meshgrid(t, t);
z = cos(x.^2 + y.^2) ./ (1.0 + x.^2 + y.^2);
plot3(x, y, z)
```

Note that unlike `plot`, the default for `plot3` is to have the box around the graph turned off. (Recall that you can turn the box around the graph on with **`box on`**.) Also note that we are only using a 100×100 grid of points.

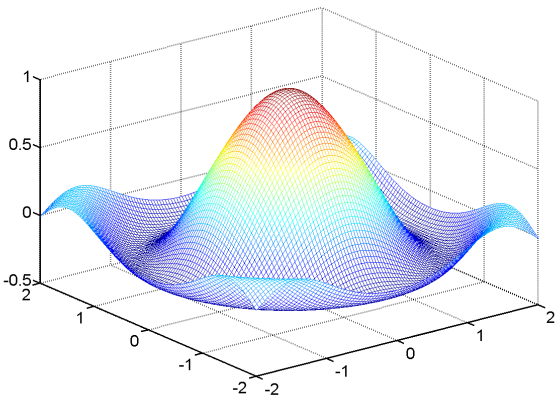
MATLAB sets the view point to see the 3D plot from to a default value. We can rotate the surface shown (equivalent to changing the view point) by choosing the Rotate 3D tool in the Figure window (in the Tools menu) and dragging the graph with the mouse. (This tool can also be turned on and off with **`rotate3d on`** and **`rotate3d off`**). The view point can be set explicitly on any 3D function with **`view`**. **`view([x y z])`** sets the view angle in Cartesian co-ordinates (note that the magnitude of the row vector `[x y z]` is ignored). **`view(3)`** sets the view point to be the default view for 3D plots.

If you choose the Pan tool (also in the Tools menu) instead then as you drag the graph with the mouse it pans rather than rotates. (This tool can also be turned on and off with **`pan on`** and **`pan off`**).

You can also zoom by choosing the Zoom In or Zoom Out tools in the Figure Window (in the Tools menu) and clicking on the graph.

As you probably noticed, though, `plot3` is not the greatest function for plotting 3-dimensional surfaces, since it represents them as a series of discrete curves. MATLAB has two functions that are much better for plotting 3-dimensional surfaces, **`surf`** and **`mesh`**, and we'll look at these next.

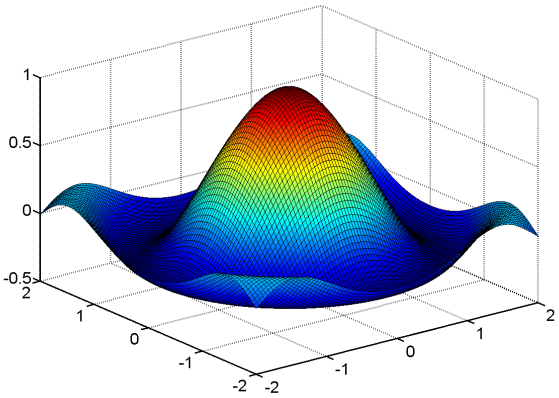
`mesh` plots a coloured 3D mesh surface:




```
t = linspace(-2, 2, 100);
[x, y] = meshgrid(t, t);
z = cos(x.^2 + y.^2) ./ (1.0 + x.^2 + y.^2);
mesh(x, y, z)
```

Note that we are only using a 100×100 grid of points. If you try this with a a 1000×1000 grid it takes noticeably longer and instead of a mesh you get a filled in surface.

`surf` plots a 3D coloured surface:

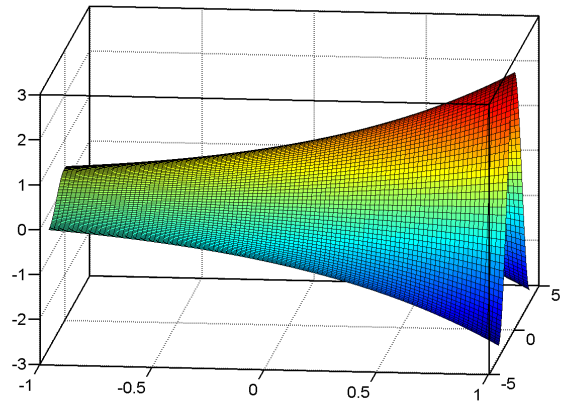



```
t = linspace(-2, 2, 100);
[x, y] = meshgrid(t, t);
z = cos(x.^2 + y.^2) ./ (1.0 + x.^2 + y.^2);
surf(x, y, z)
```

Again note that we are only using a 100×100 grid of points. If you try this with a a 1000×1000 grid not only does it take a lot longer, but you end up with a solid black outline of the surface.

`colorbar` will display a colour scale next to your graph that indicates the height represented by each colour on the surface. Try it out. **`colorbar('off')`** makes the colour scale disappear again.

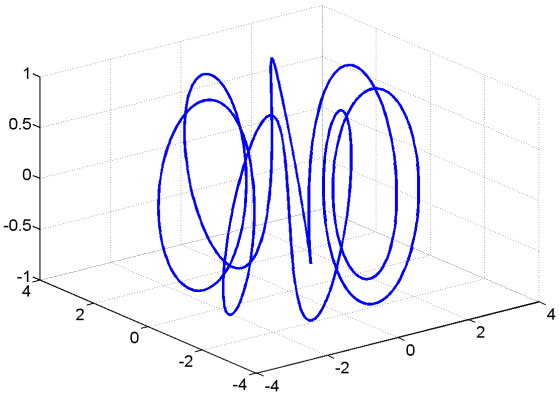
Practical 6



Plot the surface $z = e^x \cos(y)$ for $-1 \le x \le 1$, $-\pi \le y \le \pi$ and put a box around it. Save the graph in a file `practical6.png` in your current directory. Whilst you don't need to use the exact same view point as used for the graph to the left, you should use a similar one (i.e. your graph should look very similar to the one shown).

Parametric three dimensional line plots

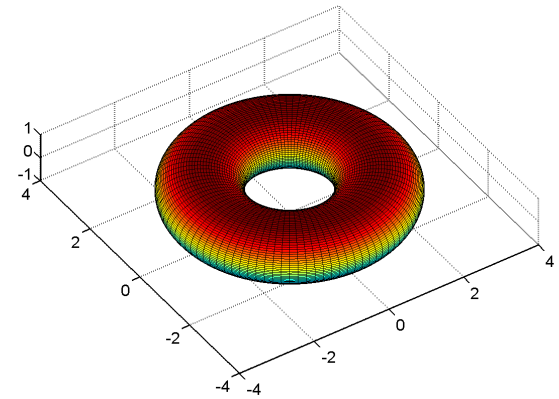
Just as we could define a 2D point as a function of a single parameter to plot a 2D line we can do the same with a 3D point to trace a line in 3D.




```
t = linspace(0, 2*pi, 1000);
x = cos(t).*(cos(7*t) + 2);
y = sin(t).*(cos(7*t) + 2);
z = sin(7*t);
plot3(x, y, z);
grid on
```

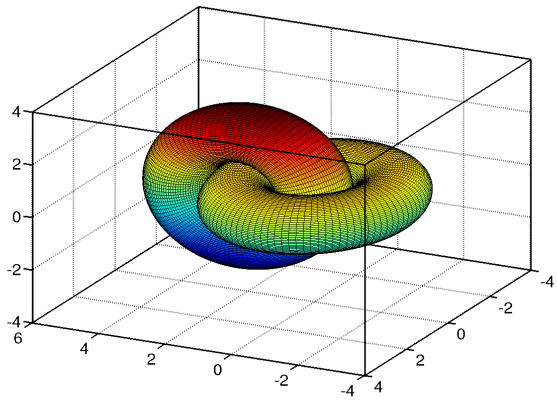
Parametric three dimensional surface plots

Just as we could trace a line with a single parameter we can define a surface with two parameters. We use the `surf` function with our surface expressed in terms of two parameters u and v.




```
[u, v] = meshgrid(linspace(0, 2*pi,
100), linspace(0, 2*pi, 100));
x = cos(u).*(cos(v) + 2);
y = sin(u).*(cos(v) + 2);
z = sin(v);
surf(x, y, z);
view([-0.1 -0.15 0.9])
```

Practical 7



Create two 3D parametric torus plots following the example above by plotting a pair of functions. Keep one exactly the same as the above example, and for the other swap the y and z definitions.

Then combine them in a single graph using successive plots with `surf` to nearly give the picture shown.

Tweak the definitions of the x-components of the two functions to give the diagram shown. (The diagram shown uses the view point given by **`view([-0.4 0.8 0.5])`**.)

Save the plot as `practical7.png`.