Reprint from
The *Mathematica* Conference

June, 1992   Boston, MA

873.21.10.1992

Chapter 2 Basic Graphics

This chapter looks at some of the simplest ways to obtain graphical output from *Mathematica*. If you are new to the system this is a good place to learn about *Mathematica* graphics. We first consider graphing of a function and move on to data plotting. This shows the basic types of images that *Mathematica* can create. Next we see how to change plots and work with the graphics options. This is followed by a look at some of the *Mathematica* graphics packages and the types of plots they provide. The chapter concludes with a brief look at animation.

2.1 Graphics Commands

■ 2.1.1 Function Plotting

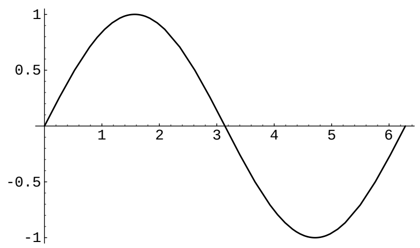
Mathematica has a number of built-in commands which plot functions. The term function is used here to mean some *Mathematica* expression which evaluates to a real number or a list of real numbers. The function plotting commands produce a wide range of images.

`Plot`	`ParametricPlot`
`Plot3D`	`ContourPlot`
`DensityPlot`	`ParametricPlot3D`
`Play`	

Mathematica commands which graph functions.

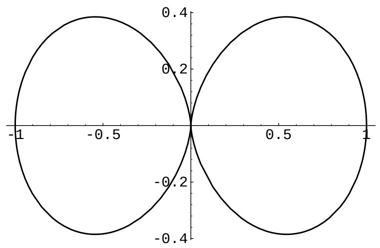
This generates a plot of `Sin[x]` for `x` from `0` to `2Pi`.

In[1]:= **`Plot[ Sin[x], {x,0,2Pi}]`**



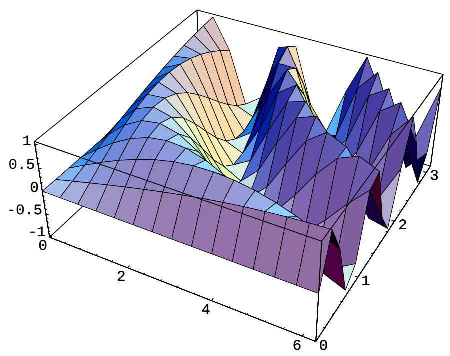
Sometimes we know how x, y depends on some parameter. For this use `ParametricPlot`.

In[2]:= **`ParametricPlot[ {Cos[t]^3, Cos[t]^2 Sin[t]}, {t,0,2Pi}]`**



This is a three-dimensional surface where the height is given by `Sin[x y]`.

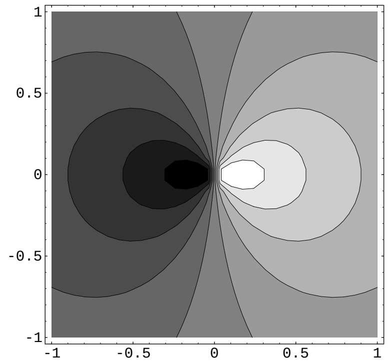
In[3]:= **`Plot3D[ Sin[x y], {x,0,2Pi}, {y,0,Pi}]`**



Other typical graphical representations such as contour plots and density plots can be generated.

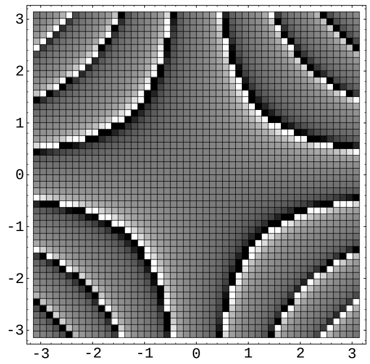
The option `PlotPoints` has been increased to give the plot a better appearance. Options are described in the next section.

```
In[4]:= ContourPlot[ Exp[ -Sqrt[ x^2 + y^2]] Cos[ ArcTan[x,y]],
             {x,-1,1}, {y,-1,1}, PlotPoints -> 30]
```



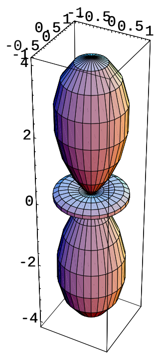
A density plot shows the value of the function at a regular array of points. Lighter regions are higher.

```
In[5]:= DensityPlot[ Tan[x y], {x,-Pi, Pi}, {y,-Pi, Pi},
        PlotPoints -> 50]
```



This is the three-dimensional analogue of `ParametricPlot`. Many of the typical *Mathematica* graphics are generated with `ParametricPlot3D`.

```
In[6]:= ParametricPlot3D[
        (3 Cos[t]^2 -1)^2 { Sin[t] Cos[p], Sin[t] Sin[p],
        Cos[t]},
        {t,0,Pi}, {p,0,2Pi}, PlotPoints -> 30]
```



`Play`, as it's name suggests, will produce an auditory representation of a function.

This will produce a 400-Hz sound.

```
In[7]:= Play[ Sin[ 2Pi 400 t], {t,0,2}]

Out[7]=-Sound-
```

Mathematica treats sounds as an extension to graphics and allows primitives which render sounds to be added to graphics objects.

■ 2.1.2 Data Plotting

We have now seen how to graph a *Mathematica* function and in certain situations this is useful. However you may have data in a file and these are what you wish to plot. In this section we shall see how to plot data. Each image shown for function plotting can be made for data plotting.

Function plotting	List plotting
`Plot`	`ListPlot`
`Plot3D`	`ListPlot3D`
`ContourPlot`	`ListContourPlot`
`DensityPlot`	`ListDensityPlot`
`Play`	`ListPlay`

Mathematica commands which plot functions and the corresponding commands to plot lists of data.

However before we look at plotting data we should see how to import data into *Mathematica*. These data may have been written by another application such as a spreadsheet or by some custom program which you have written. If the data are written in an ASCII form, that is, if they can be looked at with a text editor, they can easily be read into *Mathematica*. For the examples given here to work the data files which are loaded must exist. To prevent this being a problem I have included the *Mathematica* commands which will create the files used in these examples.

This will write the file so we can use it.

```
In[8]:= (Do[ WriteString[ "file1.dat",
                           i, "   ",
                           i^3,"\n"], {i,8}];
        Close["file1.dat"])

General::openx: file1.dat is not open.

Out[8]= Close[file1.dat]
```

This shows the contents of a data file. Notice that each line consists of two numbers.

```
In[9]:= !!file1.dat

1   1
2   8
3   27
4   64
5   125
6   216
7   343
8   512
```

This reads the numbers in the file and builds a list. The list is set to be the value of the symbol `mydata`.

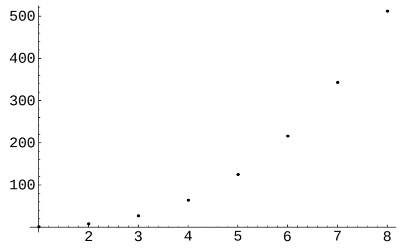
```
In[9]:= mydata = ReadList[ "file1.dat", {Number, Number}]

Out[9]= {{1, 1}, {2, 8}, {3, 27}, {4, 64}, {5, 125}, {6, 216},
  {7, 343}, {8, 512}}
```

The symbol `mydata` evaluates to the matrix of numbers.

```
In[10]:= mydata

Out[10]= {{1, 1}, {2, 8}, {3, 27}, {4, 64}, {5, 125}, {6, 216},
  {7, 343}, {8, 512}}
```

We can now plot `mydata` with `ListPlot`, the analogue of `Plot`. It takes lists of numbers instead of a function.

```
In[11]:= ListPlot[ mydata]
```



To read in a set of data and carry out a command such as `ListPlot3D`, `ListContourPlot` or `ListDensityPlot` is a little more complex. The data points must be regularly spaced over a grid. In addition you must specify the bounds of the x and y coordinates. First we shall make the file.

This writes the file we want to use for this example. This file is a little more complex than the previous example. I close the file for neatness.

```
In[12]:= (WriteString["file2.dat", -1, "   ", 1,"\n"];
          WriteString["file2.dat", -2, "   ", 2,"\n"];
          Do[ WriteString[ "file2.dat",
                          3i," ",2i," ",i," ",
                          2i," ",3i,"\n"],  {i,-2,2}];
          Close["file2.dat"])

General::openx: file2.dat is not open.

Out[12]= Close[file2.dat]
```

The first two lines contain the x and y range. The rest is the array of heights.

```
In[13]:= !!file2.dat

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

We open the file so we can read in the different parts.

```
In[13]:= stm = OpenRead["file2.dat"];
```

This is the x range.

```
In[14]:= {xmin, xmax} = Read[stm, {Number, Number}]

Out[14]= {-1, 1}
```

This is the y range.

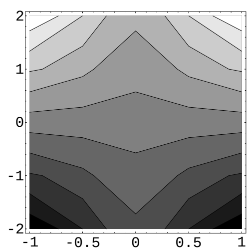
```
In[15]:= {ymin, ymax} = Read[stm, {Number, Number}]

Out[15]= {-2, 2}
```

The rest of the file contains the data. Then we close the file.

```
In[16]:= (data1 = ReadList[stm, {Number, Number,
                                  Number, Number, Number}];
          Close[stm])

Out[16]= file2.dat
```

This makes a contour plot. We could have used `ListPlot3D` or `ListDensityPlot` here. Note that the axes use the values for the mesh.

```
In[17]:= ListContourPlot[ data1, MeshRange -> {{xmin, xmax},
         {ymin, ymax}}]
```



GraphicsArray

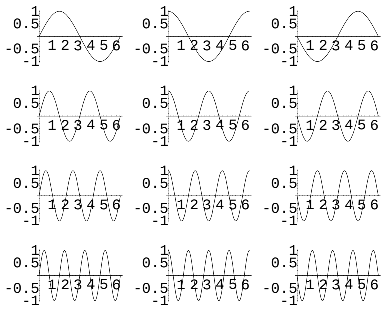
A last type of data plotting can be done with the command `GraphicsArray`. Here the data which are plotted are graphics objects.

We use `Table` to make an array of objects. `DisplayFunction -> Identity` is given to suppress the generation of graphical images at this point.

```
In[18]:= Table[
          Plot[ Sin[ i x + j], {x,0,2Pi},
         DisplayFunction -> Identity],
          {i,4}, {j,0,Pi, Pi/2}]

Out[18]= {{-Graphics-, -Graphics-, -Graphics-},
  {-Graphics-, -Graphics-, -Graphics-},
  {-Graphics-, -Graphics-, -Graphics-},
  {-Graphics-, -Graphics-, -Graphics-}}
```

This is an array of the images.

```
In[19]:= Show[ GraphicsArray[ %]]
```



2.2 Altering Pictures

Now we have seen how to make a picture. Often we wish to redisplay it or to change it in some way. This is done with the command `Show`.

`Show[` *image*`]`	redisplay *image*
`Show[` *image*`,` *option* `->` *setting*`]`	redisplay *image* with *option*
`Show[` *image1*`,` *image2*`]`	combine *image1* and *image2* and display the result

Using the command `Show`.

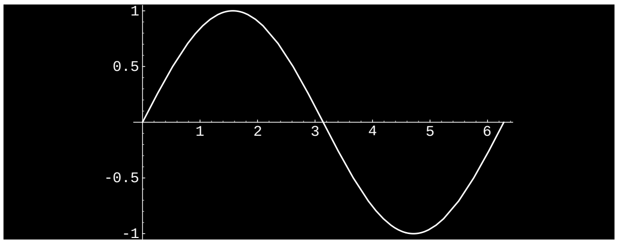
Redisplaying the image is simple. To alter the image we must examine the options which are available for graphics.

■ 2.2.1 Command Options

In *Mathematica* many built-in commands have options which alter their behavior. The options are given as rules and are given with "->". This description of passing options to a function is valid for any command in *Mathematica*, not just the graphics commands.

Here the background color of the plot is set to be a `GrayLevel` of 0, which is black. We save the plot as the symbol `a` to use it later.

```
In[20]:= a = Plot[ Sin[x], {x,0,2Pi}, Background -> GrayLevel[0]]
```

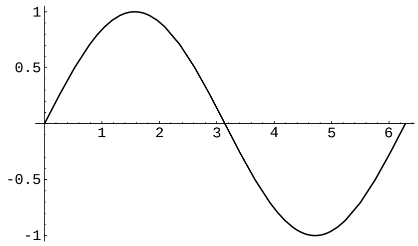


The options of `Plot` with their default values are shown.

```
In[21]:= Options[ Plot]

                               1
Out[21]= {AspectRatio -> -----------, Axes -> Automatic,
                          GoldenRatio

  AxesLabel -> None, AxesOrigin -> Automatic,
  AxesStyle -> Automatic, Background -> Automatic,
  ColorOutput -> Automatic, Compiled -> True,
  DefaultColor -> Automatic, Epilog -> {}, Frame -> False,
  FrameLabel -> None, FrameStyle -> Automatic,
  FrameTicks -> Automatic, GridLines -> None, MaxBend -> 10.,
  PlotDivision -> 20., PlotLabel -> None, PlotPoints -> 25,
  PlotRange -> Automatic, PlotRegion -> Automatic,
  PlotStyle -> Automatic, Prolog -> {}, RotateLabel -> True,
  Ticks -> Automatic, DefaultFont :> $DefaultFont,
  DisplayFunction :> $DisplayFunction}
```

This uses `Show` to change the option for background color to be white. The drawing color, which is set by the option `DefaultColor`, changes according to the background.

```
In[22]:= Show[ a, Background -> GrayLevel[1]]
```



This shows the default setting of the option `AspectRatio`.

```
In[23]:= Options[ Plot, AspectRatio]

                               1
Out[23]= {AspectRatio -> -----------}
                          GoldenRatio
```

To change a default setting one uses the command `SetOptions`.

```
In[24]:= SetOptions[ Plot, AspectRatio -> 1]

Out[24]= {AspectRatio -> 1, Axes -> Automatic,
  AxesLabel -> None, AxesOrigin -> Automatic,
  AxesStyle -> Automatic, Background -> Automatic,
  ColorOutput -> Automatic, Compiled -> True,
  DefaultColor -> Automatic, Epilog -> {}, Frame -> False,
  FrameLabel -> None, FrameStyle -> Automatic,
  FrameTicks -> Automatic, GridLines -> None, MaxBend -> 10.,
  PlotDivision -> 20., PlotLabel -> None, PlotPoints -> 25,
  PlotRange -> Automatic, PlotRegion -> Automatic,
  PlotStyle -> Automatic, Prolog -> {}, RotateLabel -> True,
  Ticks -> Automatic, DefaultFont :> $DefaultFont,
  DisplayFunction :> $DisplayFunction}
```

Let's change it back. The ";" at the end suppresses the result from printing out.

```
In[25]:= SetOptions[ Plot, AspectRatio -> GoldenRatio^(-1)];
```

Settings for Options

The permissible values for a given option vary according to the actual effect of the option. Some options take numbers to control the number of points at which a function is evaluated, some options control the appearance of axes and these take style directives and some options take a simple Boolean `True` or `False` to decide whether to do something or not.

`Automatic`	use an optimal internal algorithm
`All`	include everything
`None`	do not include this
`True`	do this
`False`	do not do this

Some common settings for options.

The value is `Automatic`. This means use an internal algorithm to choose some good value.

```
In[26]:= Options[ Plot, Background]

Out[26]= {Background -> Automatic}
```

The value is `False`. This means do not draw frame axes.

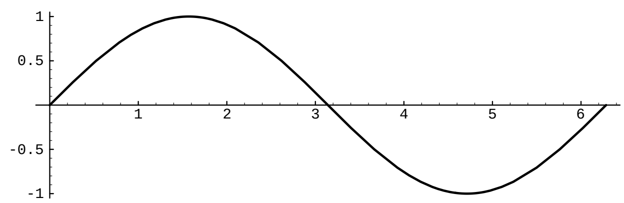
```
In[27]:= Options[ Plot, Frame]

Out[27]= {Frame -> False}
```

FullOptions

Often it is useful to use an internal algorithm to set an option, for example, setting `PlotRange` or `AspectRatio` to be `Automatic`. Sometimes you want to know the value which was actually used. This functionality is provided by `FullOptions`.

Make a plot and use `Automatic` for the `AspectRatio`.

```
In[28]:= Plot[ Sin[x], {x,0,2Pi},  AspectRatio -> Automatic]
```



FullOptions shows the literal value which was used.

```
In[29]:= FullOptions[ %, AspectRatio]

Out[29]= 0.31831
```

■ 2.2.2 Two-Dimensional Graphics

Having seen how to issue commands I shall review some ways to change two-dimensional graphics using options. This is not a detailed list of all the options of the graphics.

Axes, Frame	axes for a plot
PlotRange, AspectRatio	scaling and range of coordinates
PlotStyle	line style

Useful graphics options.

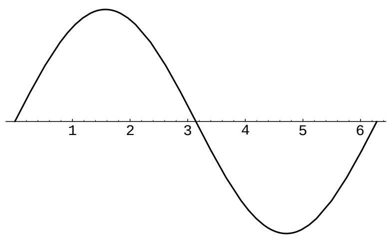
Axes and Frame

This is the default setting, it draws x and y axes.

```
In[30]:= Options[ Plot, Axes]

Out[30]= {Axes -> Automatic}
```

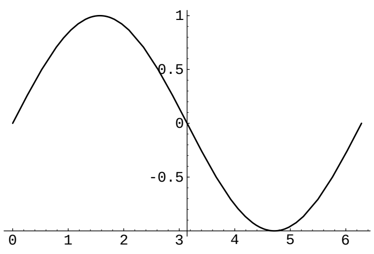
This draws only the horizontal axis.

```
In[31]:= Plot[ Sin[x], {x,0,2Pi}, Axes -> {True, False}]
```



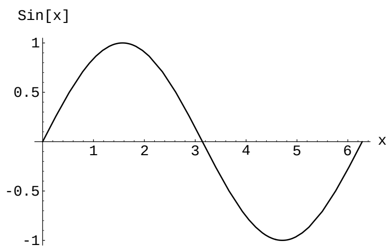
The position of the axes can be controlled with the option AxesOrigin.

```
In[32]:= Plot[ Sin[x], {x,0,2Pi}, AxesOrigin -> {Pi,-1}]
```



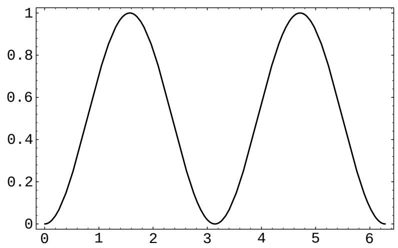
The axes can be labeled.

```
In[33]:= Plot[ Sin[x], {x,0,2Pi},
         AxesLabel -> {"x", "Sin[x]"} ]
```



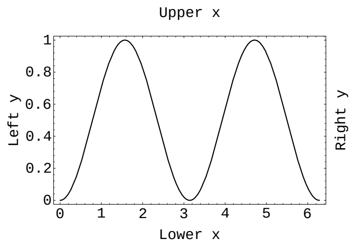
The option `Frame` produces different axes which frame the graphic.

```
In[34]:= Plot[ Sin[x]^2,  {x,0,2Pi}, Frame -> True]
```



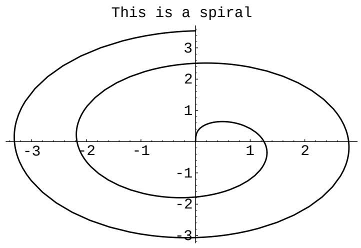
The frame can be labeled.

```
In[35]:= Show[%, FrameLabel ->
           {"Lower x", "Left y", "Upper x", "Right y"}]
```



`PlotLabel` will give a label to the whole graphic.

```
In[36]:= ParametricPlot[ {Sqrt[t] Sin[t], Sqrt[t] Cos[t]},
                         {t,0,4Pi},
                         PlotLabel -> "This is a spiral"]
```



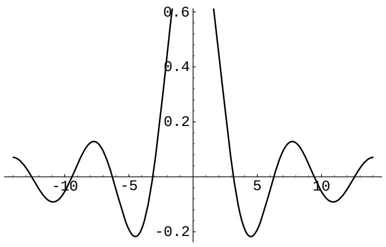
PlotRange

The `PlotRange` determines the region of user coordinates which are actually shown.

```
In[37]:= Options[ Plot, PlotRange]

Out[37]= {PlotRange -> Automatic}
```

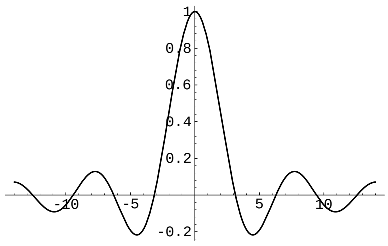
The default value of `PlotRange` is used to omit points which are scattered away from the group of the majority of the points.

```
In[38]:= Plot[ Sin[x]/x, {x,-14, 14}]
```



To see all the points specified in the primitives use `PlotRange -> All`.

```
In[39]:= Show[%, PlotRange -> All]
```

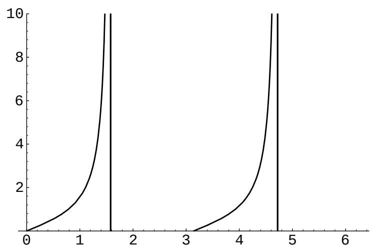


`FullOptions` shows the value of `PlotRange` which was used.

```
In[40]:= FullOptions[ %, PlotRange]

Out[40]= {{-14.7, 14.7}, {-0.247655, 1.03043}}
```

One can give explicit values to the `PlotRange` command.

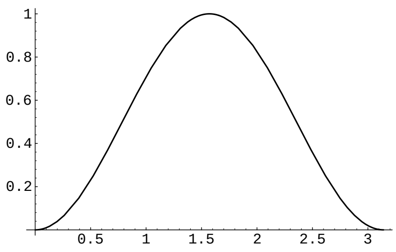
```
In[41]:= Plot[ Tan[x], {x,0,2Pi}, PlotRange -> {0,10}]
```



AspectRatio

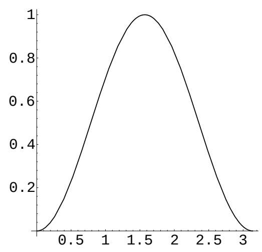
`AspectRatio` sets the shape of the output image. The default value uses the `GoldenMean`.

```
In[42]:= Plot[ Sin[x]^2, {x,0,Pi}]
```



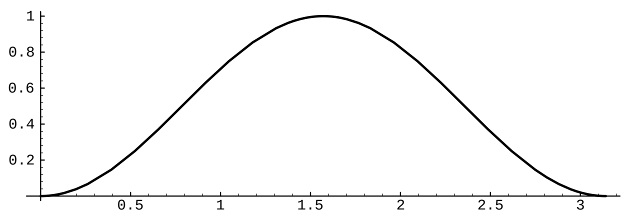
This shows a square image.

```
In[43]:= Show[%, AspectRatio -> 1]
```



This uses the real ratio of x and y.

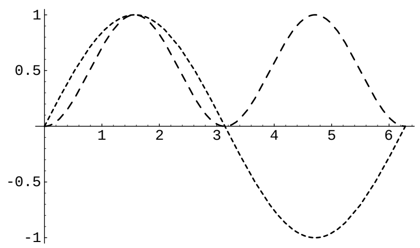
```
In[44]:= Show[%, AspectRatio -> Automatic]
```



PlotStyle

When making a number of superimposed plots, the `PlotStyle` command can be used to help distinguish between them.

```
In[45]:= Plot[ {Sin[x], Sin[x]^2}, {x,0,2Pi},
          PlotStyle -> {Dashing[ {0.01}], Dashing[{0.02}]}]
```



2.2.3 Three-Dimensional Graphics

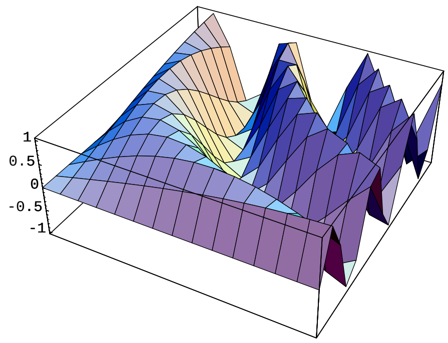
Three-dimensional graphics have similar options to two-dimensional images with axes and labels. In addition they have other options which draw boxes and grids around the image. One can also look at the picture from a different position.

`Axes`	axes for a plot
`Boxed`	a box around the plot
`FaceGrids`	grids on the faces of the box
`ViewPoint`	how the object is viewed

Some useful three-dimensional options.

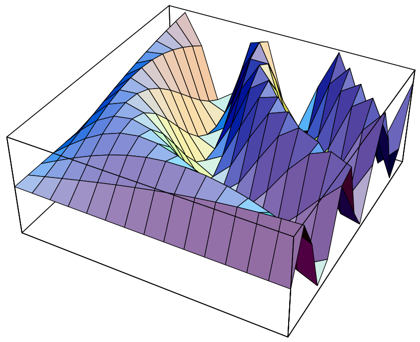
This gives a z axis but no x or y axes.

```
In[46]:= Plot3D[ Sin[x y], {x,0,2Pi}, {y,0,Pi}, Axes ->
         {False,False,True}]
```



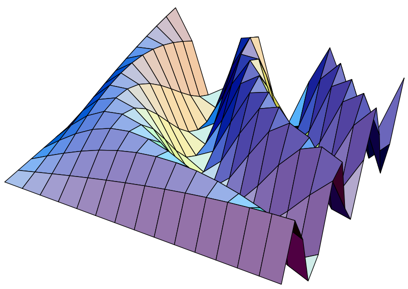
This still draws the box around the image.

```
In[47]:= Show[ %, Axes -> False]
```



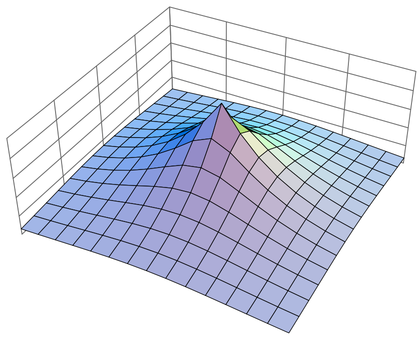
This just draws the image.

```
In[48]:= Show[ %, Boxed -> False]
```



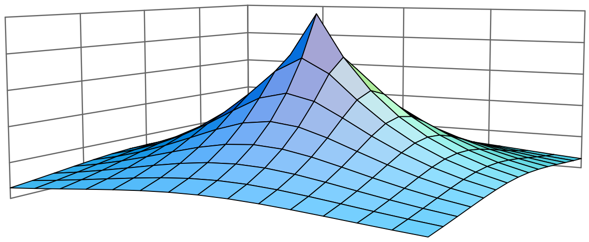
Here we draw grids behind the image.

```
In[49]:= Plot3D[
            Exp[ -Sqrt[ x^2 + y^2]], {x,-2,2}, {y,-2,2},
            Axes -> False, Boxed -> False,
            FaceGrids -> {{-1,0,0}, {0,1,0}},
            PlotRange -> {0,1}]
```



This shows a more end-on view of this surface.

```
In[50]:= Show[ %, ViewPoint -> {1.3, -2.4, .3}]
```



Often when viewing a three-dimensional surface it is necessary to experiment with a number of ways to view it to find the one which is most pleasing. Many of the *Mathematica* front ends have specialized tools to allow this to be done.

2.3 Graphics Packages

Mathematica contains a powerful programming language. Programs can be written in top-level *Mathematica* code which operate on graphics primitives and produce some particular type of graphic. There are many types of plot which are desirable, such as `PieChart`s, `BarChart`s and `LogPlot`s. *Mathematica* comes with code to produce these. This code is stored in what we call a *Mathematica* package.

The *Mathematica* packages are well described in the Wolfram Research technical report the *Guide to Standard Mathematica Packages* which is distributed with *Mathematica* and is also available from Wolfram Research. This describes each of the packages and gives examples of its functions. In this section I shall give a brief look at a few of them. One important thing to remember about these packages is that for the functions to be available to *Mathematica* the package must be loaded.

This loads the package which defines polyhedra.

```
In[51]:= <<Graphics`Polyhedra`
```

By using this notation, with a back quote `, *Mathematica* ensures that the same command will load the package on every computer system on which *Mathematica* runs, even when the file naming conventions for those systems are different.

These are the packages which have been loaded.

```
In[52]:= $Packages

Out[52]= {Graphics`Polyhedra`, Global`, System`}
```

2.3.1 General Data Plots

In the previous section the command `ListPlot` was suggested as a method to plot data. However, you may wish to go further and plot several sets of data or maybe use symbols to show the data points. This is done by the function `MultipleListPlot` which is defined in the package `Graphics`MultipleListPlot``.

This will load the package.

```
In[53]:= <<Graphics`MultipleListPlot`
```

This is one data set.

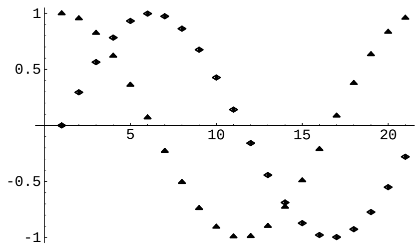
```
In[54]:= data1 = Table[ Sin[x], {x,0,2Pi,.3}];
```

This is an alternative set.

```
In[55]:= data2 = Table[ Cos[x], {x,0,2Pi,.3}];
```

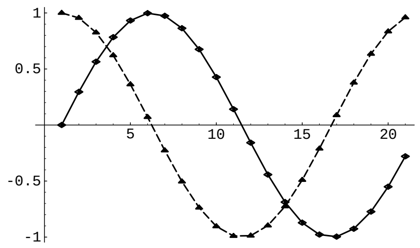
This plots the two data sets.

```
In[56]:= MultipleListPlot[ data1, data2]
```



This joins the points of each set of data and helps to distinguish them.

```
In[57]:= MultipleListPlot[ data1, data2, PlotJoined -> True]
```



In this package facilities are given to change the plotting symbols and line styles which are used. The Wolfram Research *Guide to Standard Mathematica Packages* describes this and all the other *Mathematica* packages in detail.

■ 2.3.2 PieCharts

These are defined in the package `Graphics`Graphics``.

Load the package.

```
In[58]:= <<Graphics`Graphics`
```

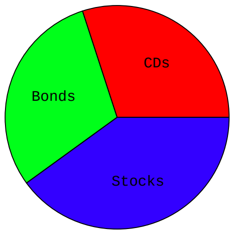
```
In[59]:= ?PieChart
```

```
PieChart[{y1, y2, ...}] generates a pie chart of the values yi.
   The values yi need to be positive. Several options
   (PieLabels, PieStyle, PieLineStyle, PieExploded) are
   available to modify the style of the pie.
```

This is data I want to plot. If this were in a file I could read it with a command like `ReadList["file", {Number, String}]`.

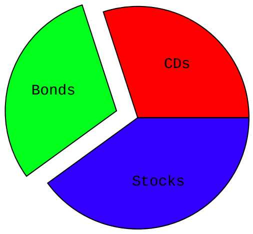
```
In[59]:= data = {{30, "CDs"}, {30, "Bonds"}, {40, "Stocks"}}

Out[59]= {{30, CDs}, {30, Bonds}, {40, Stocks}}
```

```
In[60]:= PieChart[ data]
```



One can use the options to alter the appearance of the result.

```
In[61]:= PieChart[ data,  PieExploded -> {2, .2}]
```

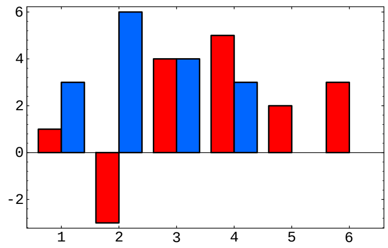


2.3.3 BarChart

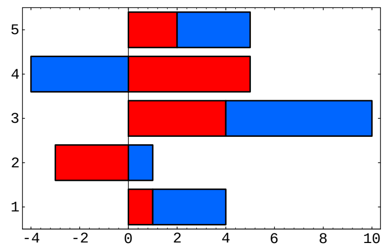
BarChart is defined in the package Graphics`Graphics`. If this package is already loaded there is no need to load it again.

The Frame option is useful in BarChart to keep the bars from obstructing the labeling. If Graphics`Graphics` has not been loaded you will have to load it first.

```
In[62]:= BarChart[ {1, -3, 4, 5, 2, 3}, {3, 6, 4, 3}, Frame ->
         True]
```



```
In[63]:= StackedBarChart[ {1, -3, 4, 5, 2}, {3, 1, 6, -4, 3},
         BarOrientation -> Horizontal, Frame -> True]
```



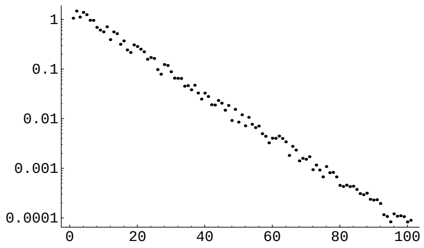
2.3.4 LogPlot

Sometimes it is useful to make a plot with a scaling other than linear. For example if some value is growing or falling very rapidly one may wish to plot the logarithm of the value. One could of course do Plot[Log[fun[x]], {x,0,t}]. This is unsatisfactory since it does not actually plot fun[x] and the axes will look strange. LogPlot[fun[x], {x,0,t}] would be much better. Support for many logarithmically scaled plots is defined in the package Graphics`Graphics`.

```
In[64]:= data = Table[ Exp[ -x] (1+ Random[]), {x,0,10,.1}];
```

This probably contains more information than the result of `ListPlot[ data]`.

`In[65]:=` **`LogListPlot[ data]`**



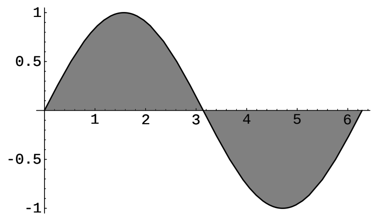
■ 2.3.5 FilledPlot

A good picture will maximize the amount of ink used to display information. If one is plotting two functions simultaneously one may be interested in the absolute values of the functions but also in the difference between them. One effective way to do this is shade in the area between the two plots. This is done with the command `FilledPlot` which is defined in the package `Graphics`FilledPlot``.

`In[66]:=` **`<<Graphics`FilledPlot``**

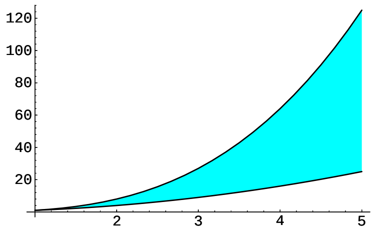
Giving one function fills between the curve and the axis.

`In[67]:=` **`FilledPlot[ Sin[x], {x,0,2Pi}]`**



With two functions the area between them is filled and we can see the relationship between the two functions more clearly.

`In[68]:=` **`FilledPlot[ {x^2,x^3}, {x,1,5}]`**



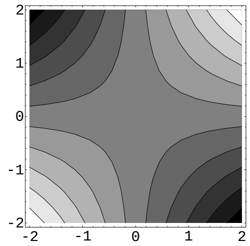
■ 2.3.6 PlotLegend

The various options give a number of methods for labeling a plot. However you may wish to give a more sophisticated label or legend to a graphic. This is provided with the function `ShowLegend` which is defined in the package `Graphics`Legend``.

`In[69]:=` **`<<Graphics`Legend``**

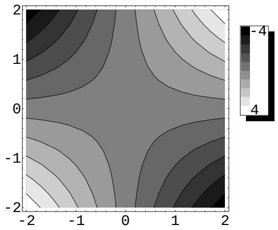
First we make a graphic.

```
In[70]:= cont = ContourPlot[ x y, {x,-2,2}, {y,-2,2}]
```



Now we can redisplay it adding the legend.

```
In[71]:= ShowLegend[ cont, {GrayLevel, 10, "-4", " 4",
         LegendPosition -> {1.1,0}}]
```



2.4 Animation

Animation of a sequence of graphical images gives an extra dimension to the data which is represented. Since this extra dimension is easily associated with time one way to use an animation is to demonstrate and investigate dynamical effects.

In *Mathematica* there is a number of ways to carry out animations. The simplest is provided by the Notebook front end to *Mathematica*. One generates a sequence of graphics, groups them and issues the appropriate command to start the animation. The other way, also available on front ends which do not support Notebooks, is to use the package `Graphics`Animation``. This package defines useful commands which work on Notebook and non-Notebook front ends. Here I demonstrate using the animation package since it is more general.

`<<Graphics`Animation``	load the animation package (it may be loaded)
`Animate[`*plot*`, {`*t*`, `*tmin*`, `*tmax*`}]`	execute the graphics command *plot* for a sequence of values of *t*, and animate the resulting sequence of frames
`ShowAnimation[{`g_1`, `g_2`, ... }]`	produce an animation from a sequence of graphics objects

Commands defined in the package `Graphics`Animation``.

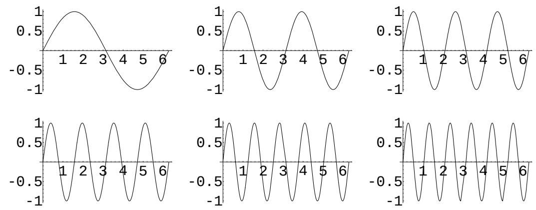
To demonstrate the effects of an animation in notes like these is quite difficult. One way to do this is to use the package `FlipBookAnimation`` which produces an array of images.

Do not load `FlipBookAnimation`` unless you wish to make a static sequence of pictures like these. If you wish an animated image your system is already configured to work.

```
In[72]:= <<FlipBookAnimation`
```

```
In[73]:= <<Graphics`Animation`
```

```
In[74]:= Animate[
           Plot[ Sin[ x n], {x,0,2Pi}],
           {n, 1,6,1}]
```



To make full use of animation commands one needs more understanding of programming ideas and of the use of graphics options than have been covered so far. For this reason we leave the subject and cover it in more detail in a later chapter.