Reprint from
The *Mathematica* Conference

June, 1992 Boston, MA

619.12.10.1992

Introduction to *Mathematica* Graphics

Tom Wickham-Jones

1 Introduction

Mathematica is a computer system which integrates symbolic and numerical mathematics with powerful computer graphics. These are supported by a concise and flexible programming language. These notes are intended to form an introduction and an overview to *Mathematica* graphics. This first section includes a very brief description of *Mathematica* as a system and how to use it. Subsequent sections will review some of the different ways to obtain graphical images. These notes only describe how to use the functionality provided by the system. The use of the *Mathematica* programming language to extend the system is beyond the scope of these notes.

■ 1.1 Introduction to *Mathematica*

Before we start to look at *Mathematica* graphics we should first review some basic *Mathematica* concepts.

Mathematica can be split into two parts: the kernel, which actually performs computations, and the front end, which handles interaction with the user. The kernel contains the same functionality on all the computers that run *Mathematica*. The front end, on the other hand, is optimized for particular computers and is concerned with system-specific tasks such as on-line editing, graphical display and printing. On many computers, the front end for *Mathematica* supports sophisticated interactive documents called Notebooks. These consist of an interactive mixture of computations, text and graphics. However the same input commands work on all versions to produce equivalent results.

When you start up a *Mathematica* kernel you see an input prompt `In[`*num*`]:=`, where *num* is a positive integer. Your documentation will tell you how to do this, the actual operations may vary from one system to another. With an input prompt you may enter something and ask *Mathematica* to evaluate it. This is how *Mathematica* works, you give it input, this is evaluated and the result is returned. We shall now examine a sample session. Enter the text which follows the input prompt.

`In[` *num* `]`	*Mathematica* is waiting for an input
`Out[` *num* `]`	The result from *Mathematica*

This arithmetic operation evaluates to a result different from the input expression.

```
In[1]:= 2 + 5

Out[1]= 7
```

There are many built-in mathematical functions.

```
In[2]:= Sin[ Sqrt[ 10.5]]

Out[2]= -0.0986171
```

The ? is a simple and quick way to get help on a function.

```
In[3]:= ?Sin

Sin[z] gives the sine of z.
```

The result is a rational number which can not simplify further.

```
In[3]:= 100 50/ 89
```

$$Out[3]= \frac{5000}{89}$$

`N` converts its argument to a real number. The `%` means the previous result.

```
In[4]:= N[ %]

Out[4]= 56.1798
```

You can also refer to a result further back by using the number from `Out`.

```
In[5]:= N[ %1]^2

Out[5]= 49.
```

Mathematica Expressions

We have now looked at various forms of arithmetic. The main type of input we have seen is for numbers. *Mathematica* can of course do much more.

- Numbers
- Symbols
- Strings
- Compound expressions

Some of the different types of *Mathematica* expressions.

Assign the number 42 to the symbol a.

```
In[6]:= a = 42

Out[6]= 42
```

The symbol a evaluates as it was set in the previous input.

```
In[7]:= a

Out[7]= 42
```

A string is put inside double quotes " ". Notice how the result drops these.

```
In[8]:= "This is a string"

Out[8]= This is a string
```

`InputForm` shows what we would have to enter into *Mathematica* to get the same result.

```
In[9]:= InputForm[ %]

Out[9]//InputForm= "This is a string"
```

This is a `List`. It is a fundamental way to hold a group of objects.

```
In[10]:= expr = {a,b,c,d}

Out[10]= {42, b, c, d}
```

The command `Table` is another way to generate a `List`. Here `i` is set to go from 1 to 5 in steps of 1.

```
In[11]:= Table[ {i, i^2, i^3}, {i,5}]

Out[11]= {{1, 1, 1}, {2, 4, 8}, {3, 9, 27}, {4, 16, 64},
  {5, 25, 125}}
```

`FullForm` shows how *Mathematica* actually stores *expr*.

```
In[12]:= FullForm[ expr]

Out[12]//FullForm= List[42, b, c, d]
```

The `Head` of *expr* is `List`. The `Head` of an expression is a way to label it.

```
In[13]:= Head[ expr]

Out[13]= List
```

The number of arguments of an expression is given by the `Length` function.

```
In[14]:= Length[ expr]

Out[14]= 4
```

`Part` returns that component of an expression.

```
In[15]:= Part[ expr, 3]

Out[15]= c
```

This is identical to the previous input. This is an abbreviated way to refer to `Part`. A number of *Mathematica* commands have similar abbreviations.

```
In[16]:= expr[[3]]

Out[16]= c
```

Mathematica Options

Many *Mathematica* commands can be altered by the use of options. All of the graphics commands have a number of options which alter the way that they behave. To see the options which are available for a *Mathematica* command one can use `Options`.

This shows the options for `Plot`.

```
In[17]:= Options[ Plot]

Out[17]= {AspectRatio -> 1/GoldenRatio, 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}
```

If there are no options an empty list is returned.

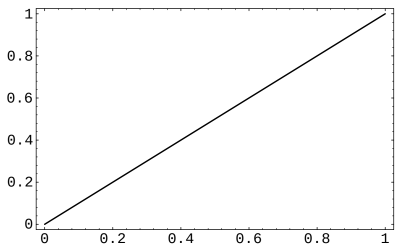
```
In[18]:= Options[ Integrate]

Out[18]= {}
```

This shows the default setting for an option.

```
In[19]:= Options[ Plot, Frame]

Out[19]= {Frame -> False}
```

This changes the option `Frame` for `Plot`. The actual meaning of these options is discussed later.

```
In[20]:= Plot[ x, {x,0,1}, Frame -> True]
```



This changes the default value for an option. The semi colon at the end of the input line suppresses the printing of the result.

```
In[21]:= SetOptions[ Plot, Frame -> True];
```

The default setting has now changed.

```
In[22]:= Options[ Plot, Frame]

Out[22]= {Frame -> True}
```

We should change the default back.

```
In[23]:= SetOptions[ Plot, Frame -> False];
```

Miscellaneous Useful Features

Mathematica runs on many different types of computers. These computers have a number of differences in the way in which they refer to and manipulate files. *Mathematica* provides commands to work with files in a way that is the same for every computer on which it runs.

This shows the files which end with the letters `dat`.

```
In[24]:= FileNames[ "*dat"]

Out[24]= {file.dat, file1.dat, file2.dat}
```

This shows the contents of the file `file.dat`. If this does not work probably the file does not exist on your system. Use a text editor to make it and insert these numbers. Then try again.

```
In[25]:= !!file.dat

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

This will reenter input line 19.

```
In[25]:= In[ 19]

Out[25]= {Frame -> False}
```

This retrieves output 19.

```
In[26]:= %19

Out[26]= {Frame -> False}
```

This is an equivalent way to retrieve output.

```
In[27]:= Out[19]

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

■ 1.2 More Information on *Mathematica*

If you wish to learn more about using *Mathematica* you should read one of the many publications and tutorials to the system which are available. Wolfram Research maintains a list of such publications which is available on request. In addition Wolfram Research publishes a number of documents such as collections of notes from *Mathematica* conferences. These cover a wide variety of topics from the general introductory to very detailed complex aspects of the system.

The standard reference to the system is the book *Mathematica: A System for Doing Mathematics by Computer*, Second Edition, by Stephen Wolfram, which I shall refer to as The *Mathematica* Book. It is distributed with every copy of the software and is also available in many technical bookstores. The *Mathematica* Book contains a reference guide which lists in alphabetical order every symbol defined by the system such as commands and options. In the reference guide each of these symbols is described in some detail. Additionally the body of the book gives further information and examples of use.

2 Graphics Commands

■ 2.1 Function Plotting

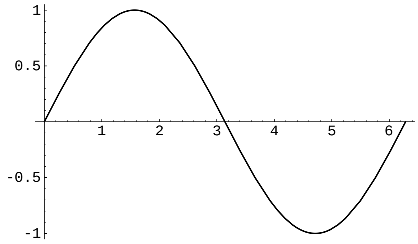
Mathematica has a number of built-in commands which plot functions. The term function is used here to mean some *Mathematica* expression which evaluates to a real number or a list of real numbers. The function plotting commands produce a wide range of images.

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

Mathematica commands which graph functions.

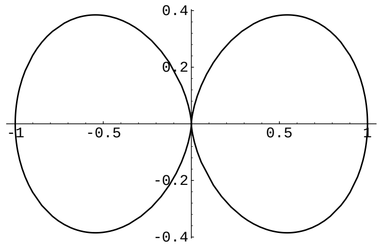
This generates a plot of `Sin[x]` for x from 0 to 2Pi.

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



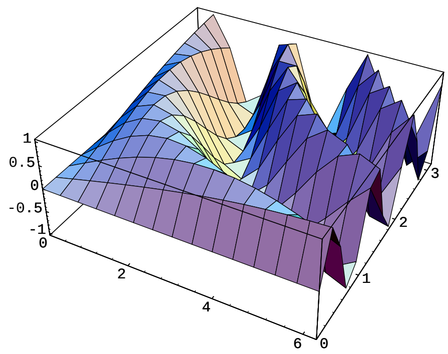
Sometimes we know how x, y depends on some parameter. For this use `ParametricPlot`.

```
In[29]:= 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]`.

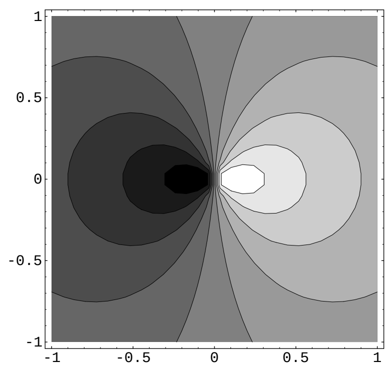
```
In[30]:= Plot3D[ Sin[x y], {x,0,2Pi}, {y,0,Pi}]
```



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

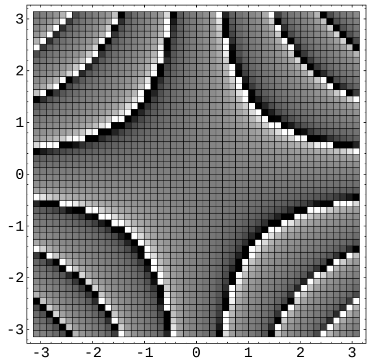
The option `PlotPoints` has been increased to give the plot a better appearance. Options are described in the next section.

```
In[31]:= ContourPlot[ Exp[ -Sqrt[ x^2 + y^2]] Cos[ ArcTan[x,y]],
                  {x,-1,1}, {y,-1,1}, PlotPoints -> 30]
```



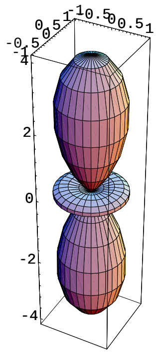
A density plot shows the value of the function at a regular array of points. Lighter regions are higher.

```
In[32]:= 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[33]:= 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[34]:= Play[ Sin[ 2Pi 400 t], {t,0,2}]

Out[34]=-Sound-
```

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

■ 2.2 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.

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

Out[35]= file1.dat
```

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

```
In[36]:= !!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`.

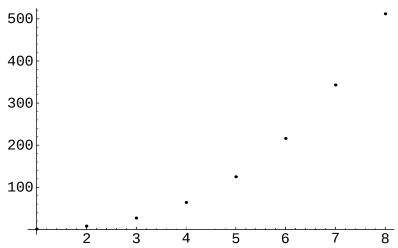
```
In[36]:= mydata = ReadList[ "file1.dat", {Number, Number}]

Out[36]= {{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[37]:= mydata

Out[37]= {{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[38]:= 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[39]:= (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"])

Out[39]= file2.dat
```

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

```
In[40]:= !!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[40]:= stm = OpenRead["file2.dat"];
```

This is the x-range.

```
In[41]:= {xmin, xmax} = Read[stm, {Number, Number}]

Out[41]= {-1, 1}
```

This is the y-range.

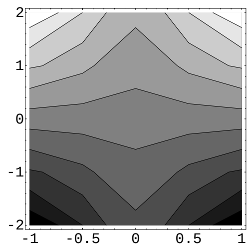
```
In[42]:= {ymin, ymax} = Read[stm, {Number, Number}]

Out[42]= {-2, 2}
```

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

```
In[43]:= (data1 = ReadList[stm, {Number, Number,
                                   Number, Number, Number}];
          Close[stm])

Out[43]= 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[44]:= ListContourPlot[ data1, MeshRange -> {{xmin, xmax},
         {ymin, ymax}}]
```



GraphicsArray

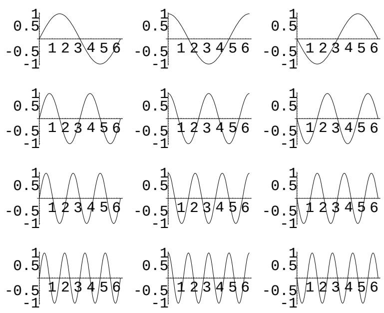
A last type of data plotting can be done with the command `GraphicsArray`. Here the data which are plotted in an array of the graphics.

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

```
In[45]:= Table[
           Plot[ Sin[ i x + j], {x,0,2Pi},
          DisplayFunction -> Identity],
           {i,4}, {j,0,Pi, Pi/2}]

Out[45]= {{-Graphics-, -Graphics-, -Graphics-},
  {-Graphics-, -Graphics-, -Graphics-},
  {-Graphics-, -Graphics-, -Graphics-},
  {-Graphics-, -Graphics-, -Graphics-}}
```

This is an array of the images.

```
In[46]:= Show[ GraphicsArray[ %]]
```



■ 2.3 More Information

To gain information on the range of commands which produce graphical output these notes are a good starting point. If you need more information on using a particular command you can look in the reference guide of The *Mathematica* Book and in the body of the book itself. Another possibility is the conference notes, "*Mathematica* Graphics", which complement these notes.

3 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`.

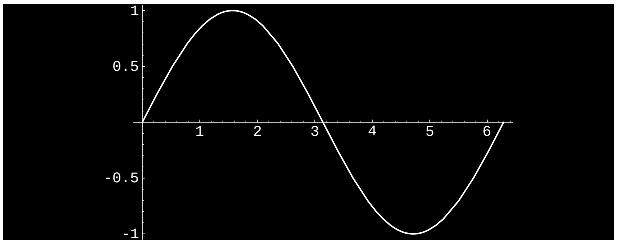
Redisplaying the image is simple. To alter the image we must examine the options which are available for graphics.

■ 3.1 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[47]:= a = Plot[ Sin[x], {x,0,2Pi}, Background -> GrayLevel[0]]
```

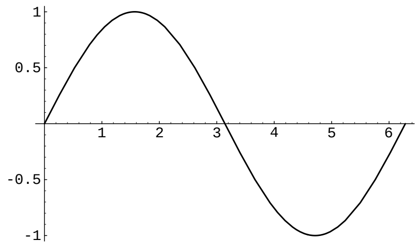


The options of Plot with their default values are shown.

```
In[48]:= Options[ Plot]

Out[48]= {AspectRatio -> 1/GoldenRatio, 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}
```

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



This shows the default setting of the option AspectRatio.

```
In[50]:= Options[ Plot, AspectRatio]

Out[50]= {AspectRatio -> 1/GoldenRatio}
```

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

```
In[51]:= SetOptions[ Plot, AspectRatio -> 1]

Out[51]= {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[52]:= 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[53]:= Options[ Plot, Background]

Out[53]= {Background -> Automatic}
```

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

```
In[54]:= Options[ Plot, Frame]

Out[54]= {Frame -> False}
```

■ 3.2 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`	notation for a graph
`PlotRange, AspectRatio`	scaling and range of coordinates
`PlotStyle`	line style

Useful graphics options.

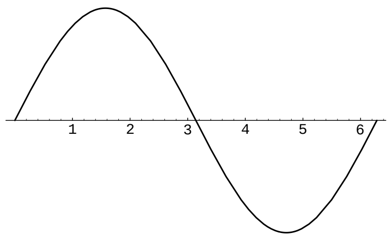
Axes and Frame

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

```
In[55]:= Options[ Plot, Axes]

Out[55]= {Axes -> Automatic}
```

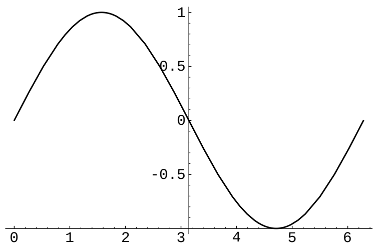
This draws the x-axis but not the y-axis.

```
In[56]:= Plot[ Sin[x], {x,0,2Pi}, Axes -> {True, False}]
```



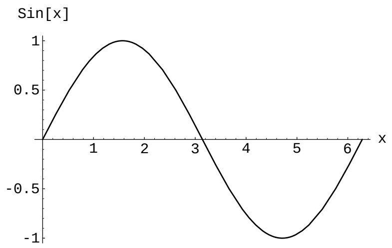
The position of the axes can be controlled with the option `AxesOrigin`.

```
In[57]:= Plot[ Sin[x], {x,0,2Pi}, AxesOrigin -> {Pi,-1}]
```



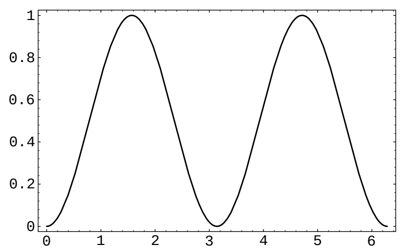
The axes can be labeled.

```
In[58]:= Plot[ Sin[x], {x,0,2Pi},
           AxesLabel -> {"x", "Sin[x]"} ]
```



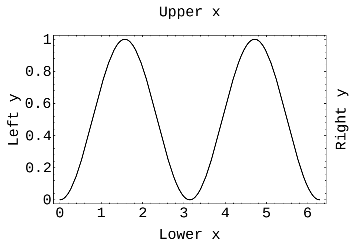
The option `Frame` produces different axes which frame the graphic.

```
In[59]:= Plot[ Sin[x]^2,  {x,0,2Pi}, Frame -> True]
```



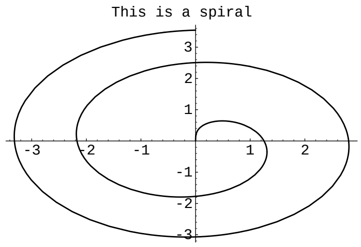
The frame can be labeled.

```
In[60]:= Show[%, FrameLabel ->
          {"Lower x", "Left y", "Upper x", "Right y"}]
```



PlotLabel will give a label to the whole graphic.

```
In[61]:= ParametricPlot[ {Sqrt[t] Sin[t], Sqrt[t] Cos[t]},
                    {t,0,4Pi},
                    PlotLabel -> "This is a spiral"]
```



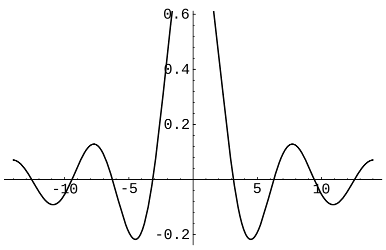
PlotRange

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

```
In[62]:= Options[ Plot, PlotRange]

Out[62]= {PlotRange -> Automatic}
```

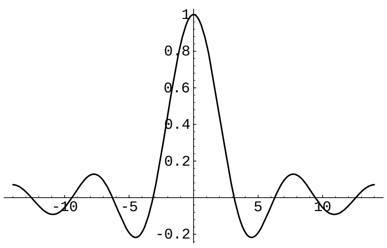
The default value of PlotRange is used to omit points which are scattered away from the group of the majority of the points.

```
In[63]:= Plot[ Sin[x]/x, {x,-14, 14}]
```



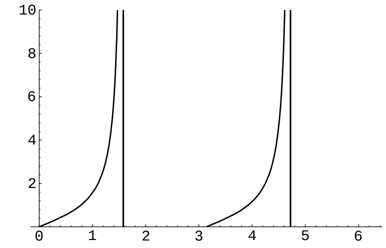
To see all the points specified in the primitives use PlotRange -> All.

```
In[64]:= Show[%, PlotRange -> All]
```



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

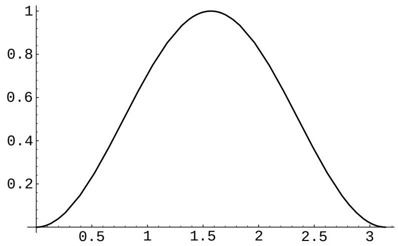
```
In[65]:= Plot[ Tan[x], {x,0,2Pi}, PlotRange -> {0,10}]
```



AspectRatio

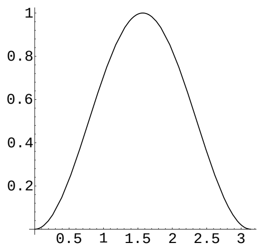
`AspectRatio` sets the shape of the output image. The default value uses the `GoldenMean`.

```
In[66]:= Plot[ Sin[x]^2, {x,0,Pi}]
```



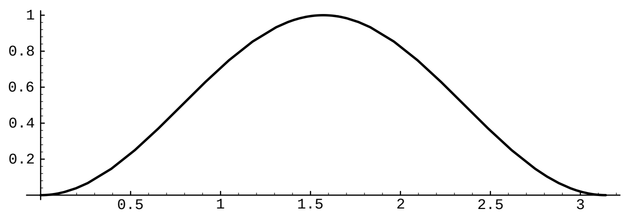
This shows a square image.

```
In[67]:= Show[%, AspectRatio -> 1]
```



This uses the real ratio of x and y.

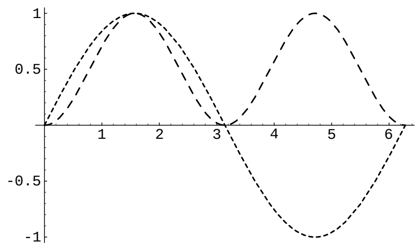
```
In[68]:= Show[%, AspectRatio -> Automatic]
```



PlotStyle

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

```
In[69]:= Plot[ {Sin[x], Sin[x]^2}, {x,0,2Pi},
          PlotStyle -> {Dashing[ {0.01}], Dashing[{0.02}]}]
```



■ 3.3 Three-Dimensional Graphics

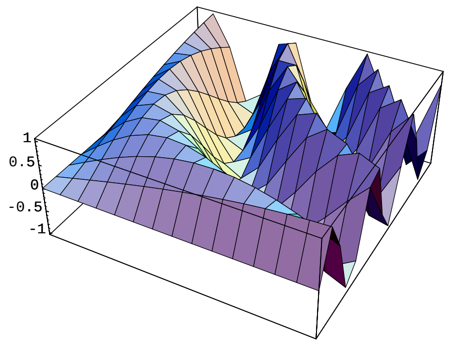
Three-dimensional graphics have similar options to two-dimensional images with axes and labels. In addition they have other options which draw boxes and grids around the image. One can also look at the picture from a different position.

`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.

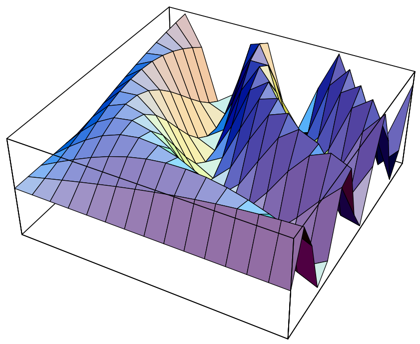
This gives a z-axis but no x or y axes.

```
In[70]:= Plot3D[ Sin[x y], {x,0,2Pi}, {y,0,Pi}, Axes ->
         {False,False,True}]
```



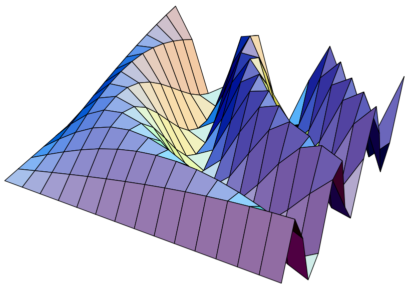
This still draws the box around the image.

```
In[71]:= Show[ %, Axes -> False]
```



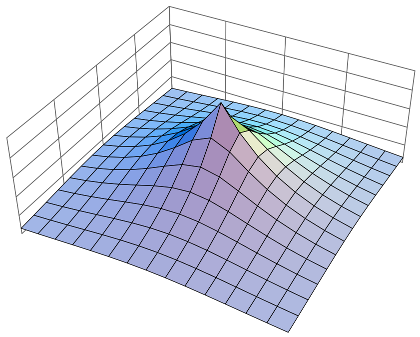
This just draws the image.

```
In[72]:= Show[ %, Boxed -> False]
```



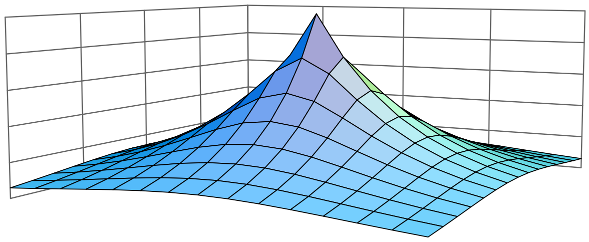
Here we draw grids behind the image.

```
In[73]:= 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[74]:= 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.

3.4 More Information

There are many more options which can be used with graphics commands than those described here. For information on these one can first look at all the options for a given graphics command with `Options`. Next it is possible to examine the usage message of an option with `?`. Lastly, The *Mathematica* Book contains a more detailed description of each option.

4 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 PieCharts, BarCharts and LogPlots. *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[75]:= <<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[76]:= $Packages

Out[76]= {Graphics`Polyhedra`, Global`, System`}
```

4.1 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[77]:= <<Graphics`MultipleListPlot`
```

This is one dataset.

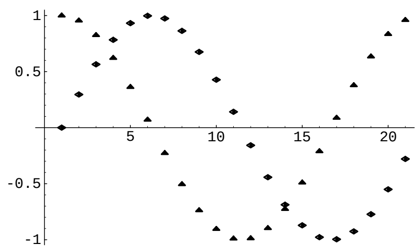
```
In[78]:= data1 = Table[ Sin[x], {x,0,2Pi,.3}];
```

This is an alternative set.

```
In[79]:= data2 = Table[ Cos[x], {x,0,2Pi,.3}];
```

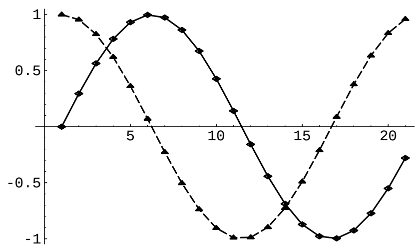
This plots the two datasets.

```
In[80]:= MultipleListPlot[ data1, data2]
```



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

```
In[81]:= 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.

■ 4.2 PieCharts

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

Load the package.

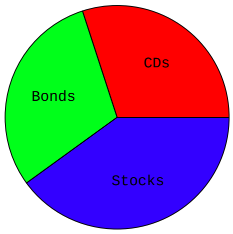
```
In[82]:= <<Graphics`Graphics`
```

```
In[83]:= ?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}]`.

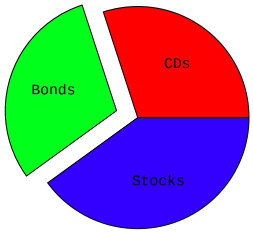
```
In[83]:= data = {{30, "CDs"}, {30, "Bonds"}, {40, "Stocks"}}

Out[83]= {{30, CDs}, {30, Bonds}, {40, Stocks}}
```

```
In[84]:= PieChart[ data]
```



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

```
In[85]:= PieChart[ data,  PieExploded -> {2, .2}]
```

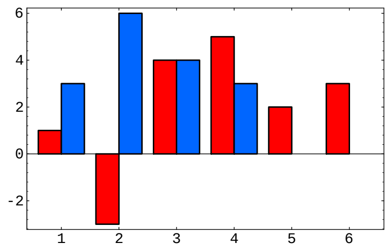


■ 4.3 BarChart

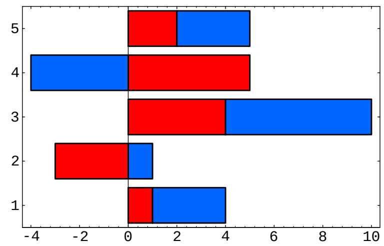
`BarChart` is defined in the package `Graphics`Graphics``. If this package is already loaded there is no need to load it again.

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



```
In[87]:= StackedBarChart[ {1, -3, 4, 5, 2}, {3, 1, 6, -4, 3},
           BarOrientation -> Horizontal, Frame -> True]
```



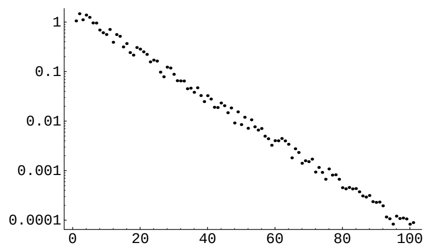
■ 4.4 LogPlot

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

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

```
In[89]:= LogListPlot[ data]
```



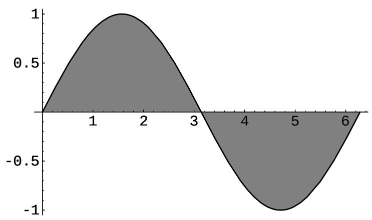
■ 4.5 FilledPlot

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[90]:= <<Graphics`FilledPlot`
```

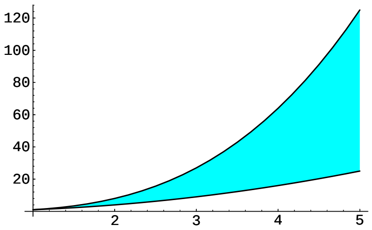
Giving one function fills between the curve and the axis.

```
In[91]:= 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[92]:= FilledPlot[ {x^2,x^3}, {x,1,5}]
```



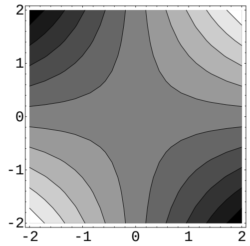
■ 4.6 PlotLegend

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[93]:= <<Graphics`Legend`
```

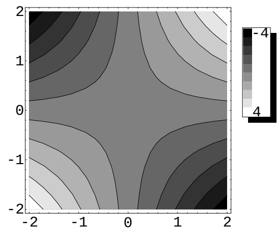
First we make a graphic.

```
In[94]:= cont = ContourPlot[ x y, {x,-2,2}, {y,-2,2}]
```



Now we can redisplay it adding the legend.

```
In[95]:= ShowLegend[ cont, {GrayLevel, 10, "-4", " 4",
           LegendPosition -> {1.1,0}}]
```



■ 4.7 More Information

The best reference for *Mathematica* packages is the Wolfram Research technical report *Guide to Standard Mathematica Packages* which is shipped with full versions of *Mathematica*. Extra copies are available from Wolfram Research at a nominal cost.

5 The Mechanics of a Plot

`Graphics`	two-dimensional pictures
`SurfaceGraphics`	simple three-dimensional surfaces
`Graphics3D`	general three-dimensional pictures
`ContourGraphics`	contour diagrams
`DensityGraphics`	bit maps
`GraphicsArray`	arrays of other pictures
`Sound`	sounds

Graphical objects in *Mathematica*.

In *Mathematica* there is a number of different types of graphical object. These types include two-dimensional graphics, three-dimensional graphics and special types which represent images such as contour diagrams and simple surfaces. The special types exist for efficiency reasons. Built-in commands for constructing these different objects exist.

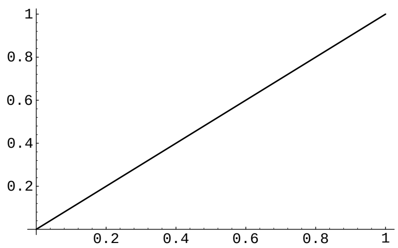
Command	Object
`Plot, ParametricPlot, ListPlot`	`Graphics`
`Plot3D, ListPlot3D`	`SurfaceGraphics`
`ContourPlot, ListContourPlot`	`ContourGraphics`
`DensityPlot, ListDensityPlot`	`DensityGraphics`
`ParametricPlot3D`	`Graphics3D`
`Play, ListPlay`	`Sound`

Graphics commands and the objects they produce.

Each command works by constructing the appropriate object. For example `Plot` builds a `Graphics` object. As an aside one should note the distinction between the generic use of the term graphics and the particular use of the *Mathematica* command `Graphics`. The latter is the *Mathematica* expression which holds two-dimensional graphics.

As can be seen there are many easy ways provided to generate all these different types of pictures. Thus one can just use built-in commands like `Plot` to make pictures. Here we shall look at the components of the command `Plot`. The actual result can be seen with `InputForm`; the drawing of the picture is a side effect.

This draws a picture. *In[96]:=* **`Plot[ x, {x,0,1}]`**



This is the result returned to *Mathematica* by the `Plot` command.

```
In[97]:= InputForm[%]

Out[97]//InputForm=
  Graphics[{{Line[{{0., 0.},
       {0.0416666666666666, 0.0416666666666666},
       {0.0833333333333333, 0.0833333333333333},
       {0.125, 0.125}, {0.1666666666666666,
        0.1666666666666666},
       {0.2083333333333333, 0.2083333333333333}, {0.25, 0.25},
       {0.2916666666666666, 0.2916666666666666},
       {0.3333333333333333, 0.3333333333333333},
       {0.375, 0.375}, {0.4166666666666666,
        0.4166666666666666},
       {0.4583333333333333, 0.4583333333333333}, {0.5, 0.5},
       {0.5416666666666666, 0.5416666666666666},
       {0.5833333333333333, 0.5833333333333333},
       {0.625, 0.625}, {0.6666666666666666,
        0.6666666666666666},
       {0.7083333333333333, 0.7083333333333333}, {0.75, 0.75},
       {0.7916666666666666, 0.7916666666666666},
       {0.833333333333333, 0.833333333333333}, {0.875, 0.875},
       {0.916666666666667, 0.916666666666667},
       {0.958333333333333, 0.958333333333333}, {1., 1.}}]}},
    {PlotRange -> Automatic, AspectRatio -> GoldenRatio^(-1),
     DisplayFunction :> $DisplayFunction,
     ColorOutput -> Automatic, Axes -> Automatic,
     AxesOrigin -> Automatic, PlotLabel -> None,
     AxesLabel -> None, Ticks -> Automatic, GridLines -> None,
     Prolog -> {}, Epilog -> {}, AxesStyle -> Automatic,
     Background -> Automatic, DefaultColor -> Automatic,
     DefaultFont :> $DefaultFont, RotateLabel -> True,
     Frame -> False, FrameStyle -> Automatic,
     FrameTicks -> Automatic, FrameLabel -> None,
     PlotRegion -> Automatic}]
```

- `fun[x]` is evaluated for `x` varying from `xmin` to `xmax`.
- These points are placed in line primitives and given a head of `Graphics`.
- The `Graphics` object is passed to `Show`, which processes options and calls the `DisplayFunction` which typically causes PostScript to be emitted.
- The `Graphics` object is returned as the result.

The steps involved in evaluating `Plot[ fun[x], {x,xmin, xmax}]`.

In these notes we have taken the view that one can just use the tools which are built in to make pictures and we did not consider the actual form of the result. Here we see that this result is just a *Mathematica* expression. We could easily write our own code to generate it, or we could take the result and adapt it in some way. To actually do this is beyond the scope of these notes. This is one of the topics covered in the conference notes, "*Mathematica* Graphics".

6 Hardcopy Output

In this section we examine how *Mathematica* graphics are actually produced and some ways to obtain hardcopy output.

`Show[` *obj1*`..,` *obj2*`, ..` *opts*`..]`	combine pictures, process options and call `DisplayFunction`
`Display[` *stream*`,` *object*`]`	send PostScript representation of *object* to *stream*

Commands used to display *Mathematica* graphics.

All the graphics commands have an option `DisplayFunction`. This defaults to `$DisplayFunction` which is set to send the graphics to `$Display`. The latter will invoke the PostScript interpreter for the window system your computer is using.

Default setting of `DisplayFunction` option.

```
In[98]:= Options[ Plot, DisplayFunction]

Out[98]= {DisplayFunction :> $DisplayFunction}
```

Default value of `$DisplayFunction`.

```
In[99]:= $DisplayFunction

Out[99]= Display[$Display, #1] &
```

On your machine you may see something else. Here the PostScript interpreter is an executable called `motifps`.

```
In[100]:= InputForm[ $Display]

Out[100]//InputForm=
  {"!motifps  -title 'Mathematica Graphics:  Out[100]'"}
```

Generally one does not want to change the `DisplayFunction`. However, there are reasons why you may wish to do this. You maybe want to write your own renderer and use *MathLink*. Then if you made the correct assignment to `DisplayFunction` all the commands like `Plot` and `Show` would just work. Another use is to set `DisplayFunction` to be `Identity`. This suppresses the production of PostScript. In certain situations this is very useful.

This produces no pictures. The result can be used in `GraphicsArray`.

```
In[101]:= Table[
            Plot[ x^n, {x,0,2}, DisplayFunction -> Identity],
               {n,1,4,1}]

Out[101]= {-Graphics-, -Graphics-, -Graphics-, -Graphics-}
```

■ 6.1 Printers and PostScript Files

This sends PostScript into the file `tmp.ps`.

```
In[102]:= Display[ "tmp.ps", Graphics[ Line[ {{0,0},{1,1}}]]]

Out[102]= -Graphics-
```

Here are the contents of the file which resulted. Note that it is not standard PostScript.

```
In[103]:= !!tmp.ps

%!
%%Creator: Mathematica
%%AspectRatio: .61803
MathPictureStart
%% Graphics
/Courier findfont 5.5  scalefont  setfont
% Scaling calculations
0.02381 0.952381 0.014715 0.588604 [
[ 0 0 0 0 ]
[ 1 .61803  0 0 ]
] MathScale
% Start of Graphics
1 setlinecap
1 setlinejoin
newpath
[ ] 0 setdash
0 g
p
P
0 0 m
1 0 L
1 .61803  L
0 .61803  L
closepath
clip
newpath
.004  w
.02381  .01472  m
.97619  .60332  L
s
% End of Graphics
MathPictureEnd
```

The command `Display` can be used to send a PostScript form of a *Mathematica* graphic to a file. However, the PostScript is not suitable to be sent to a printer. This is because various PostScript procedure definitions must be added. The methods which are available to work with *Mathematica* PostScript depend upon whether your front end supports Notebooks or not.

PostScript from Front Ends with Notebooks

On a Notebook front end it is very easy to print a picture. One simply pulls down the menu item for printing and everything works in the typical way for your particular type of computer. This means that on a NeXT you see a NeXT printer panel and options, on a Macintosh you see a Macintosh printer panel and so on. You set up the printer you want and you print. The PostScript header is added automatically.

Alternatively, you may wish to make a PostScript file that can be added into some other application program. This involves writing an encapsulated PostScript file, an EPSF. To do this, you select the image in question, copy it to the clipboard and choose the menu item for clipboard conversion. One of the options will be for an EPSF, selecting this will make the appropriate file.

PostScript from Front Ends without Notebooks

On front ends which do not support Notebooks the situation is more complex. One of the major reasons for this is that the operating system on Notebook front ends does a large amount of work. On front ends without Notebooks the operating system typically provides fewer facilities.

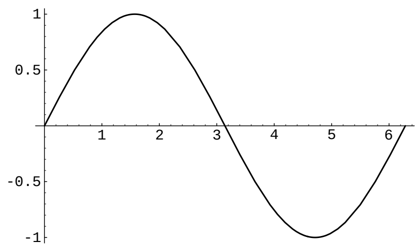
To print a graphic one uses the command `PSPrint`. This will add header information and send the result to your default printer. On Unix things look like this.

This is the definition for `PSPrint`.

```
In[103]:= ??PSPrint

PSPrint[-graphics-] sends graphics to a printer.

Attributes[PSPrint] = {Protected}

PSPrint[System`Private`x_] :=
  (Display["!psfix | lpr", System`Private`x]; System`Private`x)
```

This makes a plot.

```
In[103]:= Plot[ Sin[x], {x,0,2Pi}]
```



`PSPrint` returns the graphics object.

```
In[104]:= PSPrint[ %]

Out[104]= -Graphics-
```

psfix

`PSPrint` calls `Display`, sends the result to an executable called `psfix` and pipes the result to the printer. The executable `psfix` is a shell script with a number of options. The result of applying psfix is standard PostScript and this can be sent to a printer or some other PostScript device.

-width x	width of image
-height y	height of image
-lmarg x	left margin
-rmarg x	right margin
-bmarg y	bottom margin
-tmarg y	top margin
-land	print in landscape mode (rotated)
-epsf	produce EPSF
-stretch	do not maintain aspect ratio
-stringfix	adjust non-fixed width fonts

Some options for `psfix`.

rasterps

It is possible that one requires functionality beyond that supplied by `psfix`, for example to support a non-PostScript printer or to make an EPSF which contains a special rendering of the picture, a so-called preview image. For this the binary `rasterps` is provided. It is a PostScript interpreter which can render into a number of different formats.

-w x	image width in pixels
-h y	image height in pixels
-xdpi nx	number of dots per inch horizontal
-ydpi ny	number of dots per inch vertical
-format device	format type (see next table)

Some options for `rasterps`.

ps	PostScript
eps	Encapsulated PostScript
psimask	PostScript imagemask format
hplj.75	HP Laserjet at 75dpi
hplj.100	HP Laserjet at 100dpi
hplj.150	HP Laserjet at 150dpi
hplj.300	HP Laserjet at 300dpi
epsonfx	Epson FX series printer at 72 dpi
epsonlq	Epson LQ series printer at 180 dpi
toshp3	Toshiba P3 series printer at 180 dpi
ibmpro	IBM ProPrinter series printer at 60x72 dpi
ibmpro24	IBM ProPrinter24 series printer at 180 dpi
xbitmap	X bitmap file

Some of the possible format types for `rasterps`.

We can use `rasterps` to support other types of printer from within *Mathematica*. This is a *Mathematica* function which uses `rasterps` to print to a HP LaserJet.

```
In[105]:= HPPrint[ g_] :=
        Display["!rasterps -format hplj.75 | lpr -Phpl", g]
```

If the type of printer which you have is not supported by `rasterps` you can use the `psimask` format option. This makes a bitmap of the picture encoded in a simple PostScript way. You will have to discover how to make your printer render a bitmap and write some translation routine for it. This is a simpler task than writing a PostScript interpreter for your printer. The technical support services at Wolfram Research can give some guidance on this.