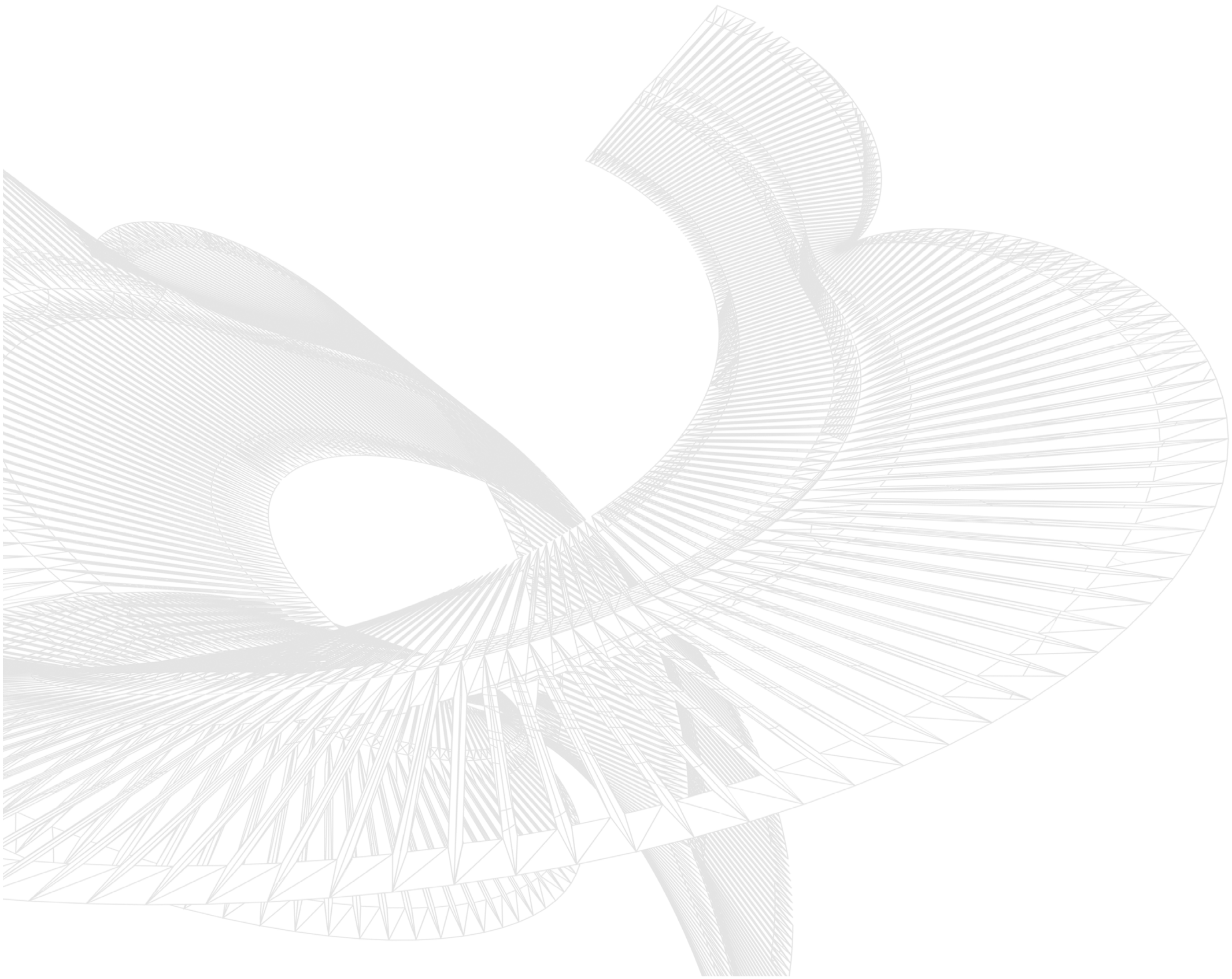
Wolfram *Mathematica* Tutorial Collection

VISUALIZATION AND GRAPHICS



For use with Wolfram *Mathematica*® 7.0 and later.

For the latest updates and corrections to this manual:
visit reference.wolfram.com

For information on additional copies of this documentation:
visit the Customer Service website at www.wolfram.com/services/customerservice
or email Customer Service at info@wolfram.com

Comments on this manual are welcomed at:
comments@wolfram.com

Printed in the United States of America.

15 14 13 12 11 10 9 8 7 6 5 4 3 2

Contents

Graphics and Sound

Basic Plotting

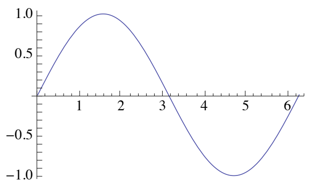
`Plot`$[f, \{x, x_{min}, x_{max}\}]$	plot f as a function of x from x_{min} to x_{max}
`Plot`$[f, \{x, x_0, x_1, \ldots, x_k\}]$	plot f over a series of segments, potentially breaking the curve at each of the x_i
`Plot`$[\{f_1, f_2, \ldots\}, \{x, x_{min}, x_{max}\}]$	plot several functions together

Basic plotting functions.

This plots a graph of $\sin(x)$ as a function of x from 0 to 2π.

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

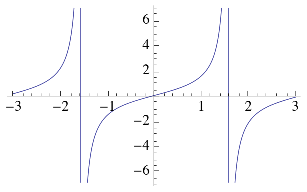
Out[1]=



You can plot functions that have singularities. *Mathematica* will try to choose appropriate scales.

```
In[2]:= Plot[Tan[x], {x, -3, 3}]
```

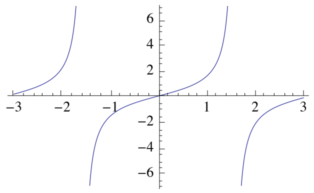
Out[2]=



The singularities can be omitted from the plot by specifying them in the plot's range.

```
In[3]:= Plot[Tan[x], {x, -3, -Pi / 2, Pi / 2, 3}]
```

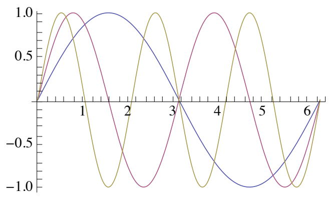
Out[3]=



You can give a list of functions to plot. A different color will automatically be used for each function.

```
In[4]:=  Plot[{Sin[x], Sin[2 x], Sin[3 x]}, {x, 0, 2 Pi}]
```

Out[4]=



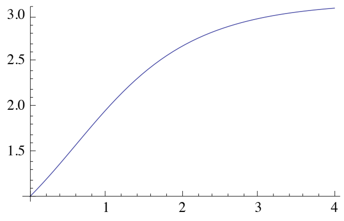
This finds the numerical solution to a differential equation, as discussed in "Numerical Differential Equations".

```
In[5]:=  NDSolve[{y'[x] == Sin[y[x]], y[0] == 1}, y, {x, 0, 4}]
Out[5]=  {{y → InterpolatingFunction[{{0., 4.}}, <>]}}
```

Here is a plot of the solution.

```
In[6]:=  Plot[y[x] /. %, {x, 0, 4}]
```

Out[6]=



Options for Graphics

When *Mathematica* plots a graph for you, it has to make many choices. It has to work out what the scales should be, where the function should be sampled, how the axes should be drawn, and so on. Most of the time, *Mathematica* will probably make pretty good choices. However, if you want to get the very best possible pictures for your particular purposes, you may have to help *Mathematica* in making some of its choices.

There is a general mechanism for specifying "options" in *Mathematica* functions. Each option has a definite name. As the last arguments to a function like `Plot`, you can include a sequence of rules of the form *name* −> *value*, to specify the values for various options. Any option for which you do not give an explicit rule is taken to have its "default" value.

`Plot[`f`,{`x`,`x_{min}`,`x_{max}`},`*option*`->`*value*`]`	make a plot, specifying a particular value for an option

Choosing an option for a plot.

A function like `Plot` has many options that you can set. Usually you will need to use at most a few of them at a time. If you want to optimize a particular plot, you will probably do best to experiment, trying a sequence of different settings for various options.

Each time you produce a plot, you can specify options for it. "Redrawing and Combining Plots" will also discuss how you can change some of the options, even after you have produced the plot.

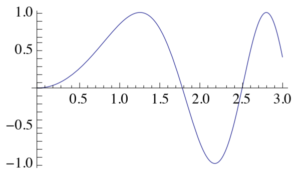
option name	*default value*	
`AspectRatio`	`1/GoldenRatio`	the height-to-width ratio for the plot; `Automatic` sets it from the absolute x and y coordinates
`Axes`	`True`	whether to include axes
`AxesLabel`	`None`	labels to be put on the axes; *ylabel* specifies a label for the y axis, {*xlabel*, *ylabel*} for both axes
`AxesOrigin`	`Automatic`	the point at which axes cross
`BaseStyle`	`{}`	the default style to use for the plot
`FormatType`	`TraditionalForm`	the default format type to use for text in the plot
`Frame`	`False`	whether to draw a frame around the plot
`FrameLabel`	`None`	labels to be put around the frame; give a list in clockwise order starting with the lower x axis
`FrameTicks`	`Automatic`	what tick marks to draw if there is a frame; `None` gives no tick marks
`GridLines`	`None`	what grid lines to include; `Automatic` includes a grid line for every major tick mark
`PlotLabel`	`None`	an expression to be printed as a label for the plot
`PlotRange`	`Automatic`	the range of coordinates to include in the plot; `All` includes all points
`Ticks`	`Automatic`	what tick marks to draw if there are axes; `None` gives no tick marks

Some of the options for `Plot`. These can also be used in `Show`.

Here is a plot with all options having their default values.

```
In[1]:= Plot[Sin[x^2], {x, 0, 3}]
```

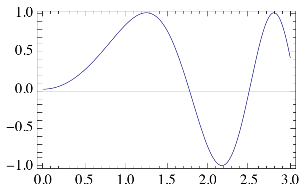
Out[1]=



This draws axes on a frame around the plot.

```
In[2]:= Plot[Sin[x^2], {x, 0, 3}, Frame -> True]
```

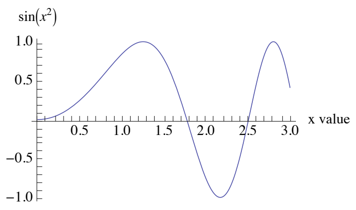
Out[2]=



This specifies labels for the x and y axes. The expressions you give as labels are printed just as they would be if they appeared as `TraditionalForm` *Mathematica* output. You can give any piece of text by putting it inside a pair of double quotes.

```
In[3]:= Plot[Sin[x^2], {x, 0, 3}, AxesLabel -> {"x value", Sin[x^2]}]
```

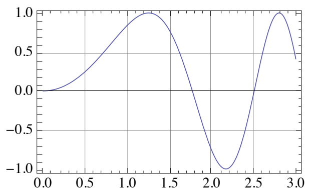
Out[3]=



You can give several options at the same time, in any order.

```
In[4]:= Plot[Sin[x^2], {x, 0, 3}, Frame -> True, GridLines -> Automatic]
```

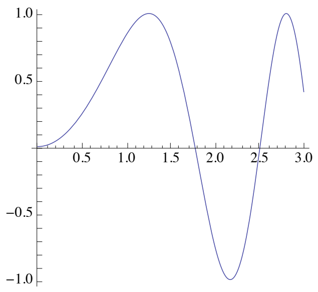
Out[4]=



Setting the `AspectRatio` option changes the whole shape of your plot. `AspectRatio` gives the ratio of width to height. Its default value is the inverse of the Golden Ratio—supposedly the most pleasing shape for a rectangle.

In[5]:=
```
Plot[Sin[x^2], {x, 0, 3}, AspectRatio -> 1]
```

Out[5]=



`Automatic`	use internal algorithms
`None`	do not include this
`All`	include everything
`True`	do this
`False`	do not do this

Some common settings for various options.

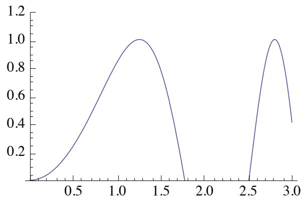
When *Mathematica* makes a plot, it tries to set the x and y scales to include only the “interesting” parts of the plot. If your function increases very rapidly, or has singularities, the parts where it gets too large will be cut off. By specifying the option `PlotRange`, you can control exactly what ranges of x and y coordinates are included in your plot.

`Automatic`	show at least a large fraction of the points, including the “interesting” region (the default setting)
`All`	show all points
$\{y_{min}, y_{max}\}$	show a specific range of y values
$\{xrange, yrange\}$	show the specified ranges of x and y values

Settings for the option `PlotRange`.

The setting for the option `PlotRange` gives explicit y limits for the graph. With the y limits specified here, the bottom of the curve is cut off.

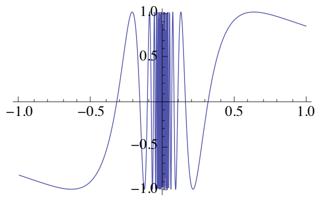
```
In[6]:= Plot[Sin[x^2], {x, 0, 3}, PlotRange -> {0, 1.2}]
```

Out[6]=


Mathematica always tries to plot functions as smooth curves. As a result, in places where your function wiggles a lot, *Mathematica* will use more points. In general, *Mathematica* tries to *adapt* its sampling of your function to the form of the function. There is, however, a limit, which you can set, to how finely *Mathematica* will ever sample a function.

The function $\sin\left(\frac{1}{x}\right)$ wiggles infinitely often when $x \simeq 0$. *Mathematica* tries to sample more points in the region where the function wiggles a lot, but it can never sample the infinite number that you would need to reproduce the function exactly. As a result, there are slight glitches in the plot.

```
In[7]:= Plot[Sin[1 / x], {x, -1, 1}]
```

Out[7]=


It is important to realize that since *Mathematica* can only sample your function at a limited number of points, it can always miss features of the function. *Mathematica* adaptively samples the functions, increasing the number of samples near interesting features, but it is still possible to miss something. By increasing `PlotPoints`, you can make *Mathematica* sample your function at a larger number of points. Of course, the larger you set `PlotPoints` to be, the longer it will take *Mathematica* to plot *any* function, even a smooth one.

option name	*default value*	
`PlotStyle`	`Automatic`	a list of lists of graphics primitives to use for each curve (see "Graphics Directives and Options")
`ClippingStyle`	`None`	what to draw when curves are clipped

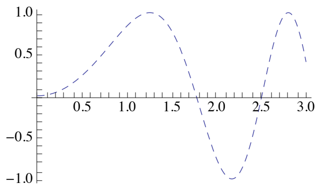
`Filling`	`None`	filling to insert under each curve
`FillingStyle`	`Automatic`	style to use for filling
`PlotPoints`	`50`	the initial number of points at which to sample the function
`MaxRecursion`	`Automatic`	the maximum number of recursive subdivisions allowed

More options for `Plot`. These cannot be used in `Show`.

This uses `PlotStyle` to specify a dashed curve.

```
In[8]:= Plot[Sin[x^2], {x, 0, 3}, PlotStyle → Dashed]
```

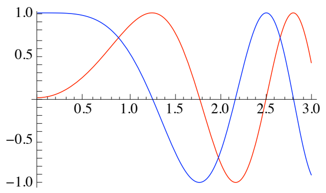
Out[8]=



When plotting multiple functions, `PlotStyle` settings in a list are used sequentially for each function.

```
In[9]:= Plot[{Sin[x^2], Cos[x^2]}, {x, 0, 3}, PlotStyle → {Red, Blue}]
```

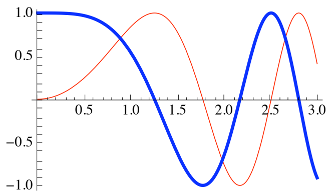
Out[9]=



When a `PlotStyle` contains a sublist, the settings are combined.

```
In[10]:= Plot[{Sin[x^2], Cos[x^2]}, {x, 0, 3}, PlotStyle → {Red, {Blue, Thick}}]
```

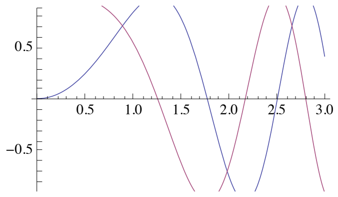
Out[10]=



By default nothing is indicated when the `PlotRange` is set, so that it cuts off curves.

```
In[11]:= Plot[{Sin[x^2], Cos[x^2]}, {x, 0, 3}, PlotRange → 0.9]
```

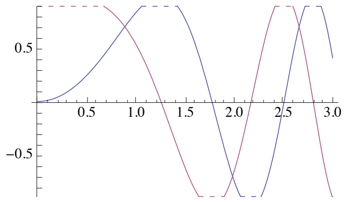
Out[11]=



Setting `ClippingStyle` to `Automatic` draws a dashed line where a curve is cut off.

```
In[12]:= Plot[{Sin[x^2], Cos[x^2]}, {x, 0, 3}, PlotRange → 0.9, ClippingStyle → Automatic]
```

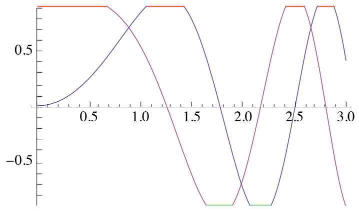
Out[12]=



Setting `ClippingStyle` to a list defines the style for the parts cut off at the bottom and top.

```
In[13]:= Plot[{Sin[x^2], Cos[x^2]}, {x, 0, 3},
          PlotRange → 0.9, ClippingStyle → {Green, Red}]
```

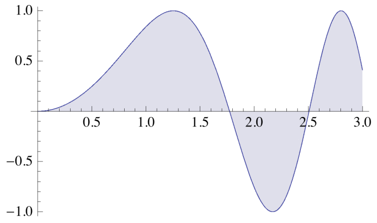
Out[13]=



This specifies filling between the curve and the x axis.

```
In[14]:= Plot[Sin[x^2], {x, 0, 3}, Filling → Axis]
```

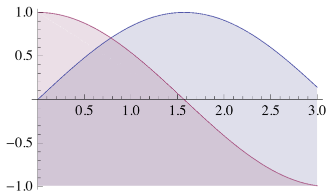
Out[14]=



The filling can be specified to extend to an arbitrary height, such as the bottom of the graphic. Filling colors are automatically blended where they overlap.

```
In[15]:= Plot[{Sin[x], Cos[x]}, {x, 0, 3}, Filling → Bottom]
```

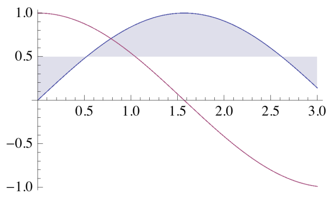
Out[15]=



This specifies a specific filling to be used only for the first curve.

```
In[16]:= Plot[{Sin[x], Cos[x]}, {x, 0, 3}, Filling → {1 → .5}]
```

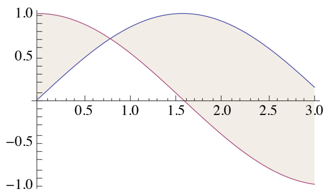
Out[16]=



This shows a filling from the first curve to the second using a nondefault filling style.

```
In[17]:= Plot[{Sin[x], Cos[x]}, {x, 0, 3}, Filling → {1 → {2}}, FillingStyle → LightBrown]
```

Out[17]=



Redrawing and Combining Plots

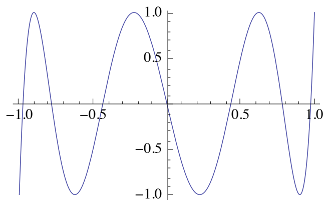
Mathematica saves information about every plot you produce, so that you can later redraw it. When you redraw plots, you can change some of the options you use.

`Show[`*plot*`,`*option*`->`*value*`]`	redraw a plot with options changed
`Show[`$plot_1$`,`$plot_2$`,`…`]`	combine several plots
`GraphicsGrid[{{`$plot_1$`,`$plot_2$`,`…`},`…`}]`	draw an array of plots
`InputForm[`*plot*`]`	show the underlying textual description of the plot

Functions for manipulating plots.

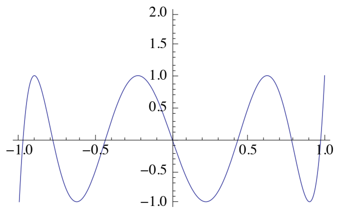
Here is a simple plot.

```
In[1]:= Plot[ChebyshevT[7, x], {x, -1, 1}]
```

Out[1]=


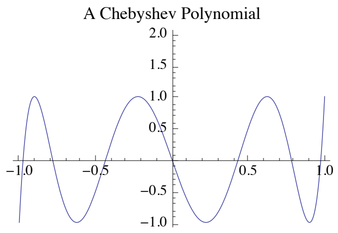
When you redraw the plot, you can change some of the options. This changes the choice of y scale.

```
In[2]:= Show[%, PlotRange -> {-1, 2}]
```

Out[2]=


This takes the plot from the previous line, and changes another option in it.

```
In[3]:= Show[%, PlotLabel -> "A Chebyshev Polynomial"]
```

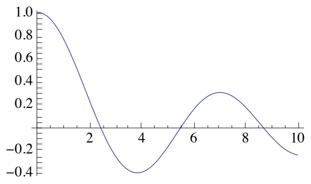
Out[3]=


By using `Show` with a sequence of different options, you can look at the same plot in many different ways. You may want to do this, for example, if you are trying to find the best possible setting of options.

You can also use `Show` to combine plots. All of the options for the resulting graphic will be based upon the options of the first graphic in the `Show` expression.

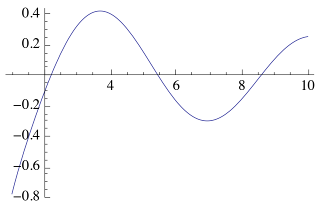
This sets `gj0` to be a plot of $J_0(x)$ from $x = 0$ to 10.

```
In[4]:=  gj0 = Plot[BesselJ[0, x], {x, 0, 10}]
```

Out[4]=


Here is a plot of $Y_1(x)$ from $x = 1$ to 10.

```
In[5]:=  gy1 = Plot[BesselY[1, x], {x, 1, 10}]
```

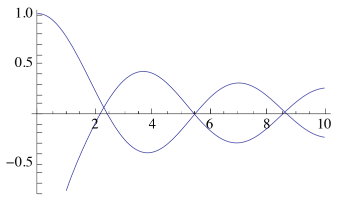
Out[5]=


`Plot` specifies an explicit `PlotRange` for each graphic.

```
In[6]:=  Options[gj0, PlotRange]
Out[6]=  {PlotRange → {{0, 10}, {-0.402759, 1.}}}
```

This uses `PlotRange` to override the explicit value set for `gj0`.

```
In[7]:=  gjy = Show[gj0, gy1, PlotRange → Automatic]
```

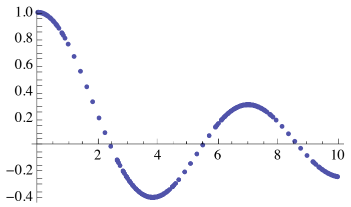
Out[7]=


All *Mathematica* graphics are expressions and can be manipulated in the same way as any other expression. Doing these kinds of manipulations does not require the use of `Show`.

This replaces all instances of the symbol `Line` with the symbol `Point` in the graphics expression represented by `gj0`.

```
In[8]:= gj0 /. Line → Point
```

Out[8]=



Using `Show[`*plot*$_1$`,` *plot*$_2$`,` ...`]` you can combine several plots into one. `GraphicsGrid` allows you to draw several plots in an array.

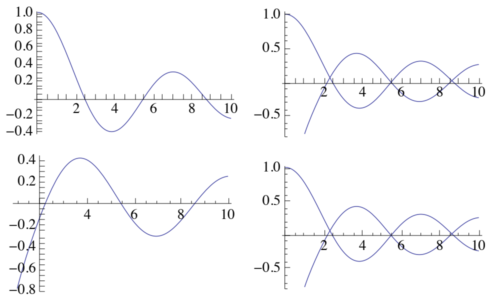
`GraphicsGrid[{{`*plot*$_{11}$`,`*plot*$_{12}$`,`...},...}]	draw a rectangular array of plots
`GraphicsRow[{`*plot*$_1$`,`*plot*$_2$`,`...`}]`	draw several plots side by side
`GraphicsColumn[{`*plot*$_1$`,`*plot*$_2$`,`...`}]`	draw a column of plots
`GraphicsGrid[`*plots*`,Spacings->{`*h*`,`*v*`}]`	put the specified horizontal and vertical spacing between the plots

Drawing arrays of plots.

This shows the previous plots in an array.

```
In[9]:= GraphicsGrid[{{gj0, gjy}, {gy1, gjy}}]
```

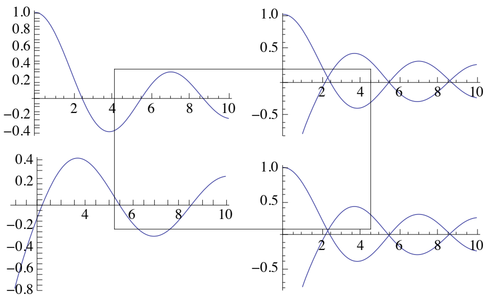
Out[9]=



If you redisplay an array of plots using `Show`, any options you specify will be used for the whole array, rather than for individual plots.

```
In[10]:= Show[%, Frame -> True, FrameTicks -> None]
```

Out[10]=

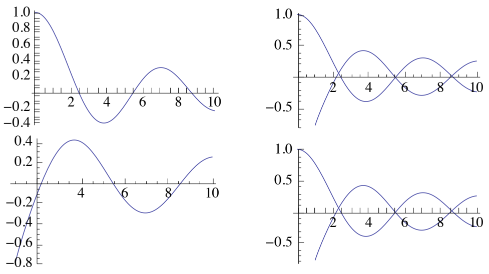


`GraphicsGrid` by default puts a narrow border around each of the plots in the array it gives. You can change the size of this border by setting the option `Spacings -> {`h`, `v`}`. The parameters h and v give the horizontal and vertical spacings to be used. The `Spacings` option uses the width and height of characters in the default font to scale the h and v parameters by default, but it is generally more useful in graphics to use `Scaled` coordinates. `Scaled` scales widths and heights so that a value of `1` represents the width and height of one element of the grid.

This increases the horizontal spacing, but decreases the vertical spacing between the plots in the array.

```
In[11]:= GraphicsGrid[{{gj0, gjy}, {gy1, gjy}}, Spacings -> {Scaled[.3], Scaled[0]}]
```

Out[11]=

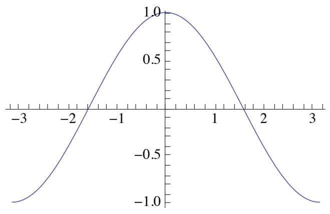


When you make a plot, *Mathematica* saves the list of points it used, together with some other information. Using what is saved, you can redraw plots in many different ways with `Show`. However, you should realize that no matter what options you specify, `Show` still has the same basic set of points to work with. So, for example, if you set the options so that *Mathematica* displays a small portion of your original plot magnified, you will probably be able to see the individual sample points that `Plot` used. Options like `PlotPoints` can only be set in the original `Plot` command itself. (*Mathematica* always plots the actual points it has; it avoids using smoothed or splined curves, which can give misleading results in mathematical graphics.)

Here is a simple plot.

```
In[12]:= Plot[Cos[x], {x, -Pi, Pi}]
```

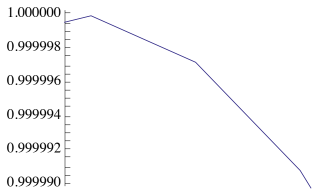
Out[12]=



This shows a small region of the plot in a magnified form. At this resolution, you can see the individual line segments that were produced by the original `Plot` command.

```
In[13]:= Show[%, PlotRange -> {{0, .005}, {.99999, 1}}]
```

Out[13]=



Manipulating Options

There are a number of functions built into *Mathematica* which, like `Plot`, have various options you can set. *Mathematica* provides some general mechanisms for handling such options.

If you do not give a specific setting for an option to a function like `Plot`, then *Mathematica* will automatically use a default value for the option. The function `Options[`*function*`,` *option*`]` allows you to find out the default value for a particular option. You can reset the default using `SetOptions[`*function*`,` *option* `->` *value*`]`. Note that if you do this, the default value you have given will stay until you explicitly change it.

`Options[`*function*`]`	give a list of the current default settings for all options
`Options[`*function*`,`*option*`]`	give the default setting for a particular option
`SetOptions[`*function*`,`*option*`->`*value*`,`…`]`	reset defaults

Manipulating default settings for options.

Here is the default setting for the `PlotRange` option of `Plot`.

```
In[1]:= Options[Plot, PlotRange]
Out[1]= {PlotRange → {Full, Automatic}}
```

This resets the default for the `PlotRange` option. The semicolon stops *Mathematica* from printing out the rather long list of options for `Plot`.

```
In[2]:= SetOptions[Plot, PlotRange -> All];
```

Until you explicitly reset it, the default for the `PlotRange` option will now be `All`.

```
In[3]:= Options[Plot, PlotRange]
Out[3]= {PlotRange → All}
```

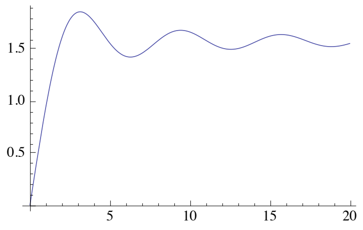
The graphics objects that you get from `Plot` or `Show` store information on the options they use. You can get this information by applying the `Options` function to these graphics objects.

`Options`[*plot*]	show all the options used for a particular plot
`Options`[*plot*,*option*]	show the setting for a specific option
`AbsoluteOptions`[*plot*,*option*]	show the absolute form used for a specific option, even if the setting for the option is `Automatic` or `All`

Getting information on options used in plots.

Here is a plot, with default settings for all options.

```
In[4]:= g = Plot[SinIntegral[x], {x, 0, 20}]
Out[4]=
```



The setting used for the `PlotRange` option was `All`.

```
In[5]:= Options[g, PlotRange]
Out[5]= {PlotRange → {All, All}}
```

`AbsoluteOptions` gives the *absolute* automatically chosen values used for `PlotRange`.

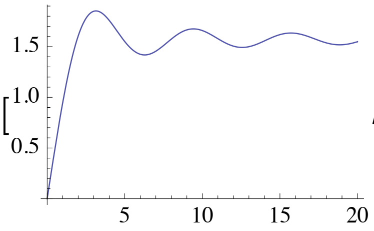
```
In[6]:= AbsoluteOptions[g, PlotRange]
```

Out[6]= $\{\texttt{PlotRange} \to \{\{4.08163\times10^{-7}, 20.\}, \{4.08163\times10^{-7}, 1.85194\}\}\}$

While it is often convenient to use a variable to represent a graphic as in the above examples, the graphic itself can be evaluated directly. The typical ways to do this in the notebook interface are to copy and paste the graphic or to simply begin typing in the graphical output cell, at which point the output cell will be converted into a new input cell.

When a plot created with no explicit `ImageSize` is placed into an input cell, it will automatically shrink to more easily accommodate input.

The following input cell was created by copying and pasting the graphical output created in the previous example.

In[7]:= **`AbsoluteOptions[`**  **`, PlotRange]`**

Out[7]=
```
{PlotRange → {{4.08163×10^-7, 20.}, {4.08163×10^-7, 1.85194}}}
```

Three-Dimensional Surface Plots

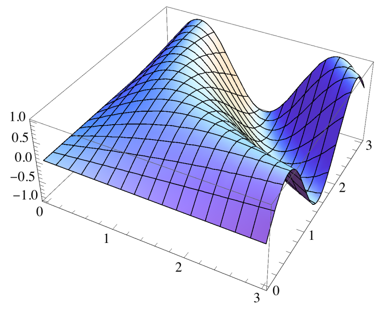
`Plot3D[`f`,{`x`,`x_{min}`,`x_{max}`},{`y`,`y_{min}`,`y_{max}`}]`	
	make a three-dimensional plot of f as a function of the variables x and y

Basic 3D plotting function.

This makes a three-dimensional plot of the function $\sin(x y)$.

In[1]:= **`Plot3D[Sin[x y], {x, 0, 3}, {y, 0, 3}]`**

Out[1]=



Three-dimensional graphics can be rotated in place by dragging the mouse inside of the graphic. Dragging inside of the graphic causes the graphic to tumble in a direction that follows the mouse, and dragging around the borders of the graphic causes the graphic to spin in the plane of the screen. Dragging the graphic while holding down the Shift key causes the graphic to pan. Use the Ctrl key (Cmd key on Macintosh) to zoom.

There are many options for three-dimensional plots in *Mathematica*. Some are discussed here; others are described in "The Structure of Graphics and Sound".

The first set of options for three-dimensional plots is largely analogous to those provided in the two-dimensional case.

option name	*default value*	
`Axes`	`True`	whether to include axes
`AxesLabel`	`None`	labels to be put on the axes: *zlabel* specifies a label for the *z* axis, {*xlabel*, *ylabel*, *zlabel*} for all axes
`BaseStyle`	`{}`	the default style to use for the plot
`Boxed`	`True`	whether to draw a three-dimensional box around the surface
`FaceGrids`	`None`	how to draw grids on faces of the bounding box; `All` draws a grid on every face
`LabelStyle`	`{}`	style specification for labels
`Lighting`	`Automatic`	simulated light sources to use
`Mesh`	`Automatic`	whether an *xy* mesh should be drawn on the surface
`PlotRange`	`{Full,Full, Automatic}`	the range of *z* or other values to include
`SphericalRegion`	`False`	whether to make the circumscribing sphere fit in the final display area
`ViewAngle`	`All`	angle of the field of view
`ViewCenter`	`{1,1,1}/2`	point to display at the center
`ViewPoint`	`{1.3,-2.4,2}`	the point in space from which to look at the surface
`ViewVector`	`Automatic`	position and direction of a simulated camera
`ViewVertical`	`{0,0,1}`	direction to make vertical
`BoundaryStyle`	`Automatic`	how to draw boundary lines for surfaces
`ClippingStyle`	`Automatic`	how to draw clipped parts of surfaces
`ColorFunction`	`Automatic`	how to determine the color of the surfaces
`Filling`	`None`	filling under each surface

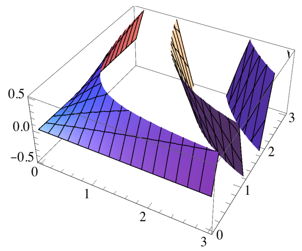
`FillingStyle`	`Opacity[.5]`	style to use for filling
`PlotPoints`	`25`	the number of points in each direction at which to sample the function; $\{n_x, n_y\}$ specifies different numbers in the x and y directions
`PlotStyle`	`Automatic`	graphics directives for the style of each surface

Some options for `Plot3D`. The first set can also be used in `Show`.

This redraws the previous plot with options changed. With this setting for `PlotRange`, only the part of the surface in the range $-0.5 \leq z \leq 0.5$ is shown.

```
In[2]:= Show[%, PlotRange -> {-0.5, 0.5}]
```

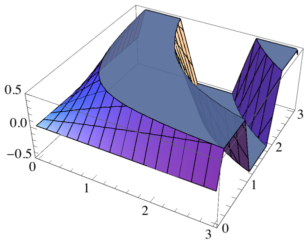
Out[2]=



The `ClippingStyle` option of `Plot3D` can be used to fill in the clipped regions.

```
In[3]:= Plot3D[Sin[x y], {x, 0, 3}, {y, 0, 3},
         PlotRange -> {-0.5, 0.5}, ClippingStyle → {Opacity[.9, Gray]}]
```

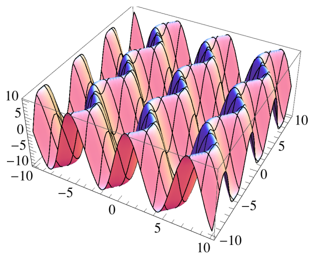
Out[3]=



When you make the original plot, you can choose to sample more points. *Mathematica* adaptively samples the plot, adding points for large variations, but occasionally you may still need to specify a greater number of points.

```
In[4]:= Plot3D[10 Sin[x + Sin[y]], {x, -10, 10}, {y, -10, 10}, PlotPoints -> 50]
```

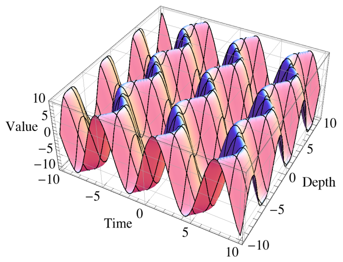
Out[4]=



Here is the same plot, with labels for the axes, and grids added to each face.

```
In[5]:= Show[%, AxesLabel -> {"Time", "Depth", "Value"}, FaceGrids -> All]
```

Out[5]=

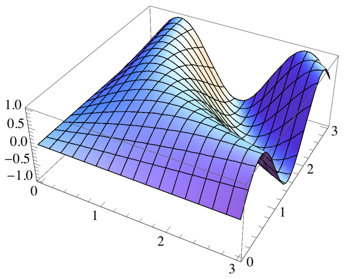


Probably the single most important issue in plotting a three-dimensional surface is specifying where you want to look at the surface from. The `ViewPoint` option for `Plot3D` and `Show` allows you to specify the point $\{x, y, z\}$ in space from which you view a surface. The details of how the coordinates for this point are defined are discussed in "Coordinate Systems for Three-Dimensional Graphics". When rotating a graphic using the mouse, you are adjusting the `ViewPoint` value.

Here is a surface, viewed from the default view point `{1.3, -2.4, 2}`. This view point is chosen to be "generic", so that visually confusing coincidental alignments between different parts of your object are unlikely.

```
In[6]:= Plot3D[Sin[x y], {x, 0, 3}, {y, 0, 3}]
```

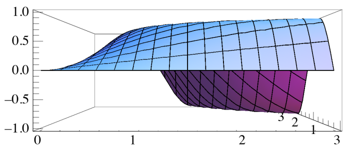
Out[6]=



This redraws the picture, with the view point directly in front. Notice the perspective effect that makes the back of the box look much smaller than the front.

```
In[7]:= Show[%, ViewPoint -> {0, -2, 0}]
```

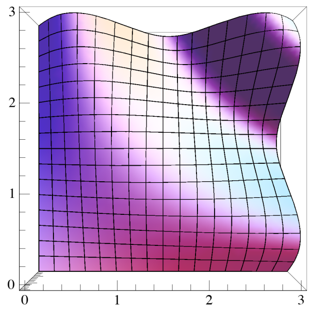
Out[7]=



The `ViewPoint` option also accepts various symbolic values which represent common view points.

```
In[8]:= Show[%, ViewPoint → Above]
```

Out[8]=



`{1.3,-2.4,2}`	default view point
`Front`	in front, along the negative y direction
`Back`	in back, along the positive y direction
`Above`	above, along the positive z direction
`Below`	below, along the negative z direction
`Left`	left, along the negative x direction
`Right`	right, along the positive x direction

Typical choices for the `ViewPoint` option.

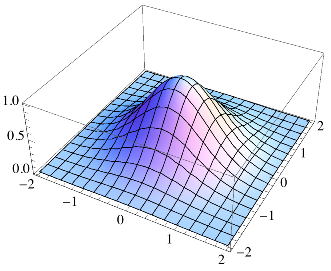
The human visual system is not particularly good at understanding complicated mathematical surfaces. As a result, you need to generate pictures that contain as many clues as possible about the form of the surface.

View points slightly above the surface usually work best. It is generally a good idea to keep the view point close enough to the surface that there is some perspective effect. Having a box explicitly drawn around the surface is helpful in recognizing the orientation of the surface.

Here is a plot with the default settings for surface rendering options.

```
In[9]:= Plot3D[Exp[-(x^2 + y^2)], {x, -2, 2}, {y, -2, 2}]
```

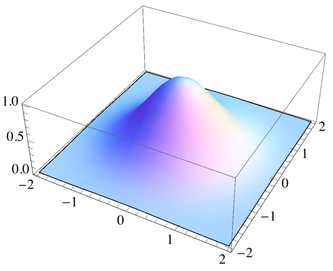
Out[9]=



This shows the surface without the mesh drawn. It is usually much harder to see the form of the surface if the mesh is not there.

```
In[10]:= Plot3D[Exp[-(x^2 + y^2)], {x, -2, 2}, {y, -2, 2}, Mesh → None]
```

Out[10]=

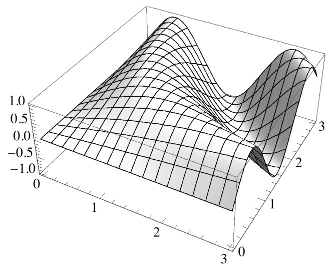


To add an extra element of realism to three-dimensional graphics, *Mathematica* by default colors three-dimensional surfaces using a simulated lighting model. In the default case, *Mathematica* assumes that there are four point light sources plus ambient lighting shining on the object. "Lighting and Surface Properties" describes how you can set up other light sources, and how you can specify the reflection properties of an object.

Lighting can also be specified using a string which represents a collection of lighting properties. For example, the option setting `Lighting -> "Neutral"` uses a set of white lights, and so can be faithfully reproduced on a black and white output device such as a printer.

```
In[11]:= Plot3D[{Sin[x y]}, {x, 0, 3}, {y, 0, 3}, Lighting → "Neutral"]
```

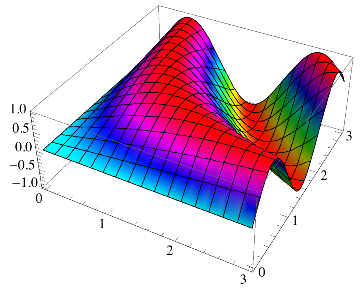
Out[11]=



The `ColorFunction` option by default uses `Lighting -> "Neutral"` so that the surface colors are not distorted by colored lights.

```
In[12]:= Plot3D[{Sin[x y]}, {x, 0, 3}, {y, 0, 3}, ColorFunction → Hue]
```

Out[12]=



Plotting Lists of Data

Mathematica can be used to make plots of *functions*. You give *Mathematica* a function, and it builds up a curve or surface by evaluating the function at many different points.

Here we describe how you can make plots from lists of data, instead of functions. ("Importing and Exporting Data" discusses how to read data from external files and programs.) The *Mathematica* commands for plotting lists of data are direct analogs of the ones for plotting functions.

`ListPlot[{`y_1`,`y_2`,…}]`	plot y_1, y_2, … at x values 1, 2, …
`ListPlot[{{`x_1`,`y_1`},{`x_2`,`y_2`},…}]`	plot points (x_1, y_1), …
`ListLinePlot[`*list*`]`	join the points with lines
`ListPlot3D[{{`z_{11}`,`z_{12}`,…},{`z_{21}`,`z_{22}`,…},…}]`	make a three-dimensional plot of the array of heights z_{yx}
`ListPlot3D[{{`x_1`,`y_1`,`z_1`},{`x_2`,`y_2`,`z_2`},…}]`	make a three-dimensional plot with heights z_i at positions {x_i, y_i}
`ListContourPlot[`*array*`]`	make a contour plot
`ListDensityPlot[`*array*`]`	make a density plot

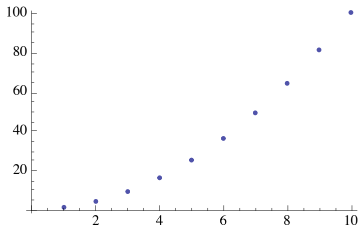
Functions for plotting lists of data.

Here is a list of values.

```
In[1]:= t = Table[i^2, {i, 10}]
Out[1]= {1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

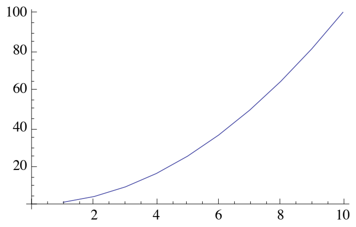
This plots the values.

```
In[2]:= ListPlot[t]
```

Out[2]=


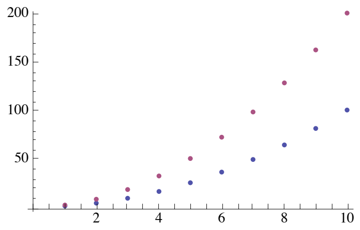
This joins the points with lines.

```
In[3]:= ListLinePlot[t]
```

Out[3]=


When plotting multiple datasets, *Mathematica* chooses a different color for each dataset automatically.

```
In[4]:= ListPlot[{t, 2 t}]
```

Out[4]=


This gives a list of x, y pairs.

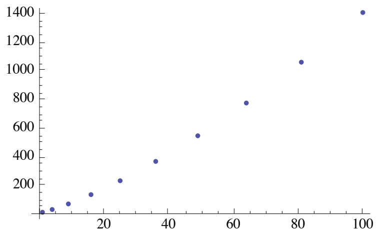
```
In[5]:= Table[{i^2, 4 i^2 + i^3}, {i, 10}]

Out[5]= {{1, 5}, {4, 24}, {9, 63}, {16, 128}, {25, 225},
         {36, 360}, {49, 539}, {64, 768}, {81, 1053}, {100, 1400}}
```

This plots the points.

```
In[6]:= ListPlot[%]
```

Out[6]=



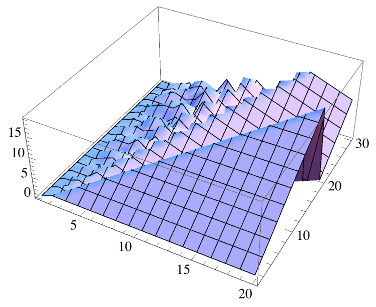
This gives a rectangular array of values. The array is quite large, so we end the input with a semicolon to stop the result from being printed out.

```
In[7]:= t3 = Table[Mod[x, y], {x, 30}, {y, 20}];
```

This makes a three-dimensional plot of the array of values.

```
In[8]:= ListPlot3D[t3]
```

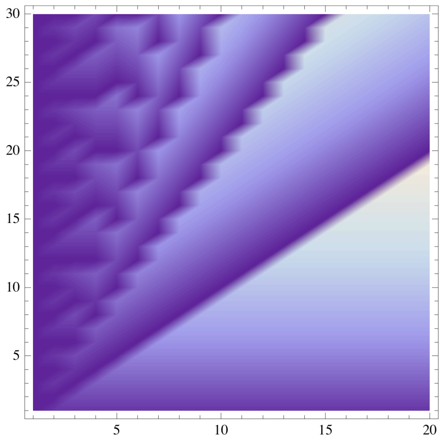
Out[8]=



This gives a density plot of the array of values.

```
In[9]:= ListDensityPlot[t3]
```

Out[9]=



Parametric Plots

"Basic Plotting" described how to plot curves in *Mathematica* in which you give the y coordinate of each point as a function of the x coordinate. You can also use *Mathematica* to make *parametric* plots. In a parametric plot, you give both the x and y coordinates of each point as a function of a third parameter, say t.

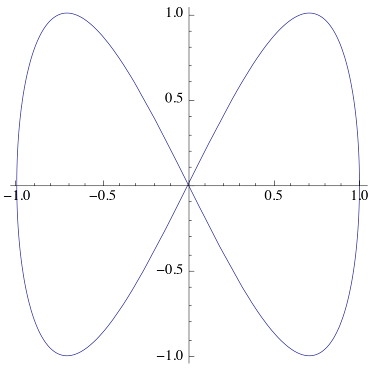
`ParametricPlot`$[\{f_x, f_y\}, \{t, t_{min}, t_{max}\}]$	make a parametric plot
`ParametricPlot`$[\{\{f_x, f_y\}, \{g_x, g_y\}, \ldots\}, \{t, t_{min}, t_{max}\}]$	plot several parametric curves together

Functions for generating parametric plots.

Here is the curve made by taking the x coordinate of each point to be `Sin[t]` and the y coordinate to be `Sin[2 t]`.

```
In[1]:= ParametricPlot[{Sin[t], Sin[2 t]}, {t, 0, 2 Pi}]
```

Out[1]=



`ParametricPlot3D`$[\{f_x, f_y, f_z\}, \{t, t_{min}, t_{max}\}]$	make a parametric plot of a three-dimensional curve
`ParametricPlot3D`$[\{f_x, f_y, f_z\}, \{t, t_{min}, t_{max}\}, \{u, u_{min}, u_{max}\}]$	make a parametric plot of a three-dimensional surface
`ParametricPlot3D`$[\{\{f_x, f_y, f_z\}, \{g_x, g_y, g_z\}, \ldots\}, \ldots]$	plot several objects together

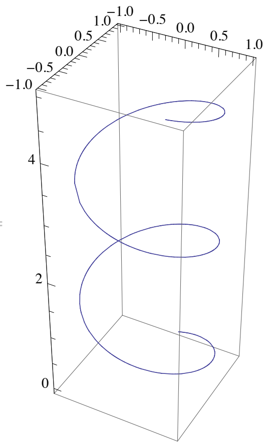
Three-dimensional parametric plots.

`ParametricPlot3D[{`f_x`, `f_y`, `f_z`}, {`t`, `t_{min}`, `t_{max}`}]` is the direct analog in three dimensions of `ParametricPlot[{`f_x`, `f_y`}, {`t`, `t_{min}`, `t_{max}`}]` in two dimensions. In both cases, *Mathematica* effectively generates a sequence of points by varying the parameter t, then forms a curve by joining these points. With `ParametricPlot`, the curve is in two dimensions; with `ParametricPlot3D`, it is in three dimensions.

This makes a parametric plot of a helical curve. Varying `t` produces circular motion in the x-y plane, and linear motion in the z direction.

```
In[2]:= ParametricPlot3D[{Sin[t], Cos[t], t / 3}, {t, 0, 15}]
```

Out[2]=

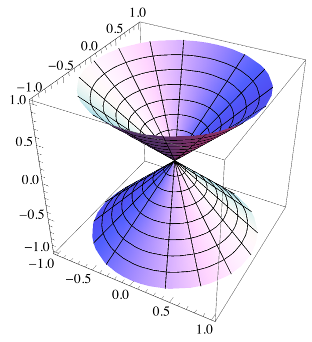


`ParametricPlot3D[{`f_x`, `f_y`, `f_z`}, {`t`, `t_{min}`, `t_{max}`}, {`u`, `u_{min}`, `u_{max}`}]` creates a surface, rather than a curve. The surface is formed from a collection of quadrilaterals. The corners of the quadrilaterals have coordinates corresponding to the values of the f_i when t and u take on values in a regular grid.

Here the x and y coordinates for the quadrilaterals are given simply by `t` and `u`. The result is a surface plot of the kind that can be produced by `Plot3D`.

```
In[3]:= ParametricPlot3D[{u Sin[t], u Cos[t], u}, {t, 0, 2 Pi}, {u, -1, 1}]
```

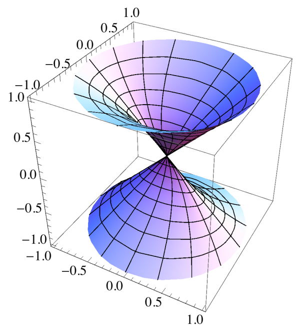
Out[3]=



This shows the same surface as before, but with the y coordinates distorted by a quadratic transformation.

```
In[4]:= ParametricPlot3D[{u Sin[t], u^2 Cos[t], u}, {t, 0, 2 Pi}, {u, -1, 1}]
```

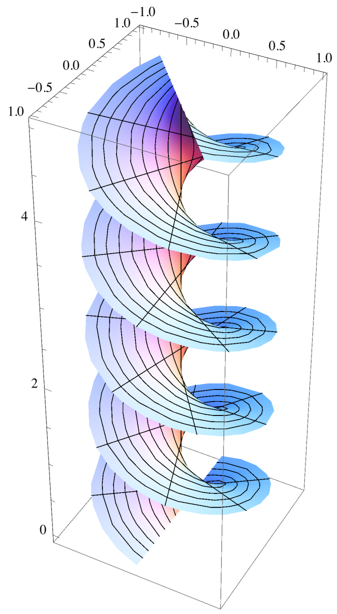
Out[4]=



This produces a helicoid surface by taking the helical curve shown above, and at each section of the curve drawing a quadrilateral.

```
In[5]:= ParametricPlot3D[{u Sin[t], u Cos[t], t / 3}, {t, 0, 15}, {u, -1, 1}]
```

Out[5]=

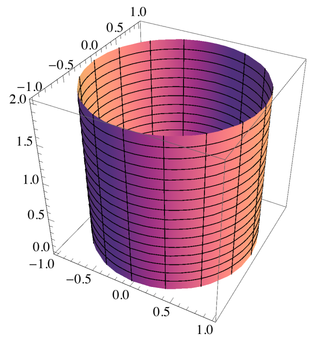


In general, it is possible to construct many complicated surfaces using `ParametricPlot3D`. In each case, you can think of the surfaces as being formed by "distorting" or "rolling up" the t-u coordinate grid in a certain way.

This produces a cylinder. Varying the `t` parameter yields a circle in the x-y plane, while varying `u` moves the circles in the z direction.

```
In[6]:= ParametricPlot3D[{Sin[t], Cos[t], u}, {t, 0, 2 Pi}, {u, 0, 2}]
```

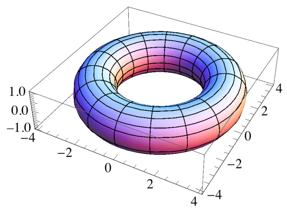
Out[6]=



This produces a torus. Varying `u` yields a circle, while varying `t` rotates the circle around the z axis to form the torus.

```
In[7]:= ParametricPlot3D[
         {Cos[t] (3 + Cos[u]), Sin[t] (3 + Cos[u]), Sin[u]}, {t, 0, 2 Pi}, {u, 0, 2 Pi}]
```

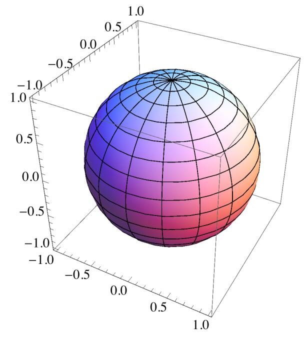
Out[7]=



This produces a sphere.

```
In[8]:= ParametricPlot3D[{Cos[t] Cos[u], Sin[t] Cos[u], Sin[u]},
         {t, 0, 2 Pi}, {u, -Pi / 2, Pi / 2}]
```

Out[8]=



You should realize that when you draw surfaces with `ParametricPlot3D`, the exact choice of parametrization is often crucial. You should be careful, for example, to avoid parametrizations in which all or part of your surface is covered more than once. Such multiple coverings often lead to discontinuities in the mesh drawn on the surface, and may make `ParametricPlot3D` take much longer to render the surface.

Some Special Plots

As discussed in "The Structure of Graphics and Sound", *Mathematica* includes a full graphics programming language. In this language, you can set up many different kinds of plots. A few of the common ones are included in standard *Mathematica* packages.

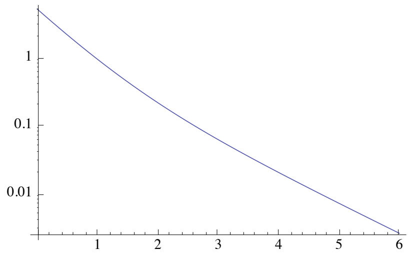
`LogPlot`[f, {x, x_{min}, x_{max}}]	generate a linear-log plot
`LogLinearPlot`[f, {x, x_{min}, x_{max}}]	generate a log-linear plot
`LogLogPlot`[f, {x, x_{min}, x_{max}}]	generate a log-log plot
`ListLogPlot`[*list*]	generate a linear-log plot from a list of data
`ListLogLinearPlot`[*list*]	generate a log-linear plot from a list of data
`ListLogLogPlot`[*list*]	generate a log-log plot from a list of data
`PolarPlot`[r, {t, t_{min}, t_{max}}]	generate a polar plot of the radius r as a function of angle t
`SphericalPlot3D`[r, {*theta*, *min*, *max*}, {*phi*, *min*, *max*}]	generate a three-dimensional spherical plot
`BarChart`[*list*]	plot a list of data as a bar chart
`ErrorListPlot`[{{x_1, y_1, dy_1}, …}]	generate a plot with error bars
`PieChart`[*list*]	plot a list of data as a pie chart

Some special plotting functions. The second group of functions are defined in standard *Mathematica* packages.

This generates a log-linear plot.

```
In[1]:= LogPlot[Exp[-x] + 4 Exp[-2 x], {x, 0, 6}]
```

Out[1]=



Here is a list of the first 10 primes.

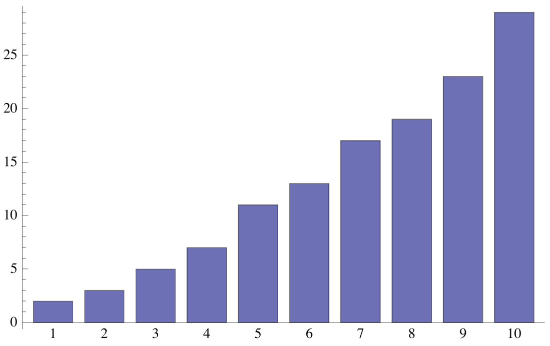
```
In[2]:= p = Table[Prime[n], {n, 10}]
Out[2]= {2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
```

Here is a bar chart of the primes.

```
In[3]:= Needs["BarCharts`"]

In[4]:= BarChart[p]
```

Out[4]=

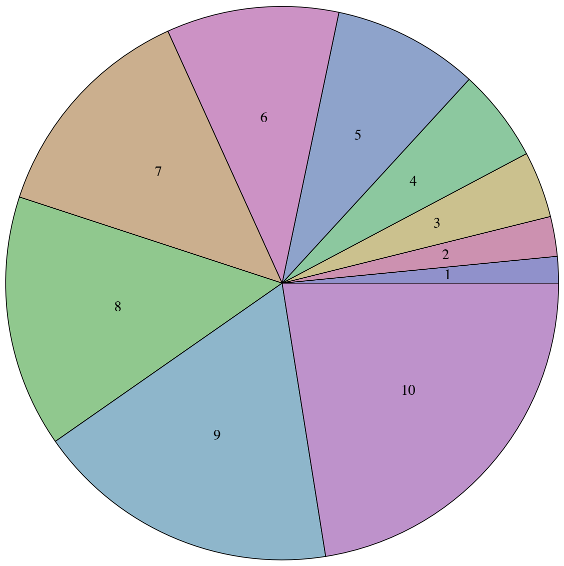


This gives a pie chart.

```
In[5]:= Needs["PieCharts`"]

In[6]:= PieChart[p]
```

Out[6]=



Sound

On most computer systems, *Mathematica* can produce not only graphics but also sound. *Mathematica* treats graphics and sound in a closely analogous way.

For example, just as you can use `Plot[`f`, {`x`, `x_{min}`, `x_{max}`}]` to plot a function, so also you can use `Play[`f`, {`t`, 0, `t_{max}`}]` to "play" a function. `Play` takes the function to define the waveform for a sound: the values of the function give the amplitude of the sound as a function of time.

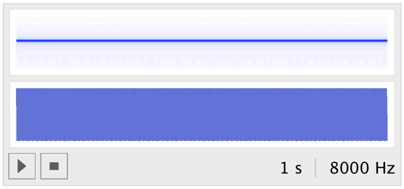
`Play[`f`,{`t`,0,`t_{max}`}]`	play a sound with amplitude f as a function of time t in seconds

Playing a function.

On a suitable computer system, this plays a pure tone with a frequency of 440 hertz for one second.

```
In[1]:= Play[Sin[2 Pi 440 t], {t, 0, 1}]
```

Out[1]=

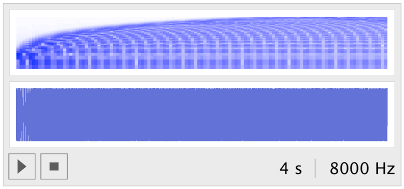


Sounds produced by `Play` can have any waveform. They do not, for example, have to consist of a collection of harmonic pieces. In general, the amplitude function you give to `Play` specifies the instantaneous signal associated with the sound. This signal is typically converted to a voltage, and ultimately to a displacement. Note that *amplitude* is sometimes defined to be the *peak* signal associated with a sound; in *Mathematica*, it is always the *instantaneous* signal as a function of time.

This plays a more complex sound.

```
In[2]:= Play[Sin[700 t + 25 t Sin[350 t]], {t, 0, 4}]
```

Out[2]=



`Play` is set up so that the time variable that appears in it is always measured in absolute seconds. When a sound is actually played, its amplitude is sampled a certain number of times every second. You can specify the sample rate by setting the option `SampleRate`.

`Play`[f, {t, 0, t_{max}}, `SampleRate` -> r]	play a sound, sampling it r times a second

Specifying the sample rate for a sound.

In general, the higher the sample rate, the better high-frequency components in the sound will be rendered. A sample rate of r typically allows frequencies up to $r/2$ hertz. The human auditory system can typically perceive sounds in the frequency range 20 to 22000 hertz (depending somewhat on age and sex). The fundamental frequencies for the 88 notes on a piano range from 27.5 to 4096 hertz.

The standard sample rate used for compact disc players is 44100. The effective sample rate in a typical telephone system is around 8000. On most computer systems, the default sample rate used by *Mathematica* is around 8000.

You can use `Play`[{f_1, f_2, …] to produce stereo sound. In general, *Mathematica* supports any number of sound channels.

`ListPlay`[{a_1, a_2, …}, `SampleRate` -> r]	play a sound with a sequence of amplitude levels

Playing sampled sounds.

The function `ListPlay` allows you simply to give a list of values which are taken to be sound amplitudes sampled at a certain rate.

When sounds are actually rendered by *Mathematica*, only a certain range of amplitudes is allowed. The option `PlayRange` in `Play` and `ListPlay` specifies how the amplitudes you give should be scaled to fit in the allowed range. The settings for this option are analogous to those for the `PlotRange` graphics option discussed in "Options for Graphics".

`PlayRange->Automatic`	use an internal procedure to scale amplitudes
`PlayRange->All`	scale so that all amplitudes fit in the allowed range
`PlayRange->{`a_{min}`,`a_{max}`}`	make amplitudes between a_{min} and a_{max} fit in the allowed range, and clip others

Specifying the scaling of sound amplitudes.

While it is often convenient to use the setting `PlayRange -> Automatic`, you should realize that `Play` may run significantly faster if you give an explicit `PlayRange` specification, so it does not have to derive one.

`EmitSound[`*snd*`]`	emit a sound when evaluated

Playing sounds programmatically.

A `Sound` object in output is typically formatted as a button which contains a visualization of the sound and which plays the sound when pressed. Sounds can be played without the need for user intervention or producing output by using `EmitSound`. In fact, the internal implementation of `Sound` buttons uses `EmitSound` when the button is pressed.

The internal structure of `Sound` objects is discussed in "The Representation of Sound".

The Structure of Graphics and Sound

The Structure of Graphics

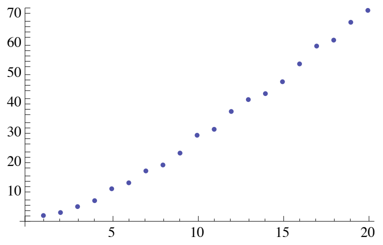
"Graphics and Sound" discusses how to use functions like `Plot` and `ListPlot` to plot graphs of functions and data. Here, we discuss how *Mathematica* represents such graphics, and how you can program *Mathematica* to create more complicated images.

The basic idea is that *Mathematica* represents all graphics in terms of a collection of *graphics primitives*. The primitives are objects like `Point`, `Line` and `Polygon`, that represent elements of a graphical image, as well as directives such as `RGBColor` and `Thickness`.

This generates a plot of a list of points.

```
In[1]:= ListPlot[Table[Prime[n], {n, 20}]]
```

Out[1]=



`InputForm` shows how *Mathematica* represents the graphics. Each point is represented as a coordinate in a `Point` graphics primitive. All the various graphics options used in this case are also given.

```
In[2]:= InputForm[%]
```

```
Out[2]//InputForm= Graphics[{{{}, {Hue[0.67, 0.6, 0.6], Point[{{1., 2.},
                        {2., 3.}, {3., 5.}, {4., 7.}, {5., 11.}, {6., 13.},
                        {7., 17.}, {8., 19.}, {9., 23.}, {10., 29.}, {11.,
                        31.}, {12., 37.}, {13., 41.}, {14., 43.}, {15.,
                        47.}, {16., 53.}, {17., 59.}, {18., 61.}, {19.,
                        67.}, {20., 71.}}]}, {}}},
                   {AspectRatio -> GoldenRatio^(-1), Axes -> True,
                    AxesOrigin -> {0, 0}, PlotRange ->
                     {{0., 20.}, {0., 71.}}, PlotRangeClipping -> True,
                    PlotRangePadding -> {Scaled[0.02], Scaled[0.02]}}]
```

Each complete piece of graphics in *Mathematica* is represented as a *graphics object*. There are several different kinds of graphics object, corresponding to different types of graphics. Each kind of graphics object has a definite head which identifies its type.

`Graphics` [*list*]	general two-dimensional graphics
`Graphics3D` [*list*]	general three-dimensional graphics

Graphics objects in *Mathematica*.

The functions like `Plot` and `ListPlot` discussed in "The Structure of Graphics and Sound" all work by building up *Mathematica* graphics objects, and then displaying them.

You can create other kinds of graphical images in *Mathematica* by building up your own graphics objects. Since graphics objects in *Mathematica* are just symbolic expressions, you can use all the standard *Mathematica* functions to manipulate them.

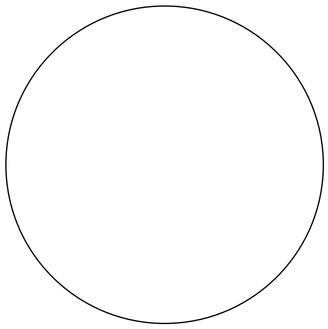
Graphics objects are automatically formatted by the *Mathematica* front end as graphics upon output. Graphics may also be printed as a side effect using the `Print` command.

The `Graphics` object is computed by *Mathematica*, but its output is suppressed by the semicolon.

```
In[3]:= Graphics[Circle[]];
        2 + 2
Out[3]= 4
```

A side effect output can be generated using the `Print` command. It has no `Out[]` label because it is a side effect.

```
In[4]:= Print[Graphics[Circle[]]];
        2 + 2
```



```
Out[4]= 4
```

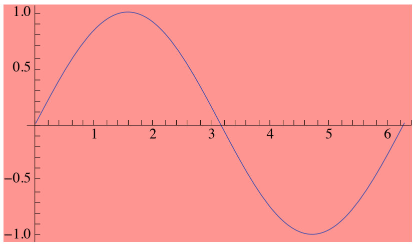
`Show[`g`, `*opts*`]`	display a graphics object with new options specified by *opts*
`Show[`g_1`,`g_2`,…]`	display several graphics objects combined using the options from g_1
`Show[`g_1`,`g_2`,…,`*opts*`]`	display several graphics objects with new options specified by *opts*

Displaying graphics objects.

`Show` can be used to change the options of an existing graphic or to combine multiple graphics.

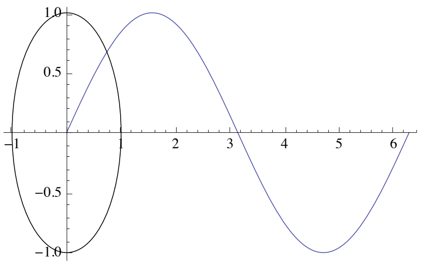
This uses `Show` to adjust the `Background` option of an existing graphic.

```
In[5]:= g1 = Plot[Sin[x], {x, 0, 2 Pi}];
        Show[g1, Background → Pink]
Out[8]=
```



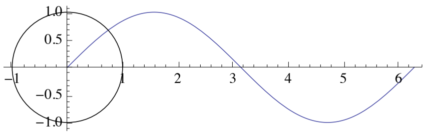
This uses `Show` to combine two graphics. The values used for `PlotRange` and other options are based upon those which were set for the first graphic.

```
In[6]:= Show[{g1, Graphics[Circle[]]}]
```

Out[9]=


Here, new options are specified for the entire graphic.

```
In[7]:= Show[{g1, Graphics[Circle[]]}, PlotRange → All, AspectRatio → Automatic]
```

Out[10]=


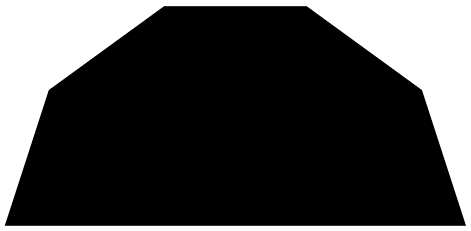
Graphics directives	Examples: `RGBColor`, `Thickness`
Graphics options	Examples: `PlotRange`, `Ticks`, `AspectRatio`, `ViewPoint`

Local and global ways to modify graphics.

Given a particular list of graphics primitives, *Mathematica* provides two basic mechanisms for modifying the final form of graphics you get. First, you can insert into the list of graphics primitives certain *graphics directives*, such as `RGBColor`, which modify the subsequent graphical elements in the list. In this way, you can specify how a particular set of graphical elements should be rendered.

This creates a two-dimensional graphics object that contains the `Polygon` graphics primitive.

```
In[8]:= poly = Polygon[Table[N[{Cos[n Pi / 5], Sin[n Pi / 5]}], {n, 0, 5}]];
        Graphics[poly]
```

Out[8]=


`InputForm` shows the complete graphics object.

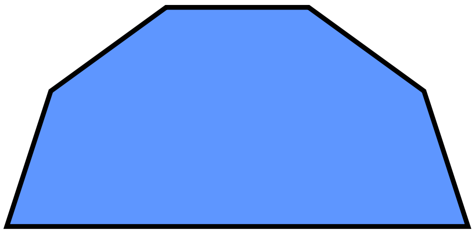
```
In[9]:= InputForm[%]

Out[9]//InputForm= Graphics[Polygon[{{1., 0.}, {0.8090169943749475,
                       0.5877852522924731}, {0.30901699437494745,
                       0.9510565162951535}, {-0.30901699437494745,
                       0.9510565162951535}, {-0.8090169943749475,
                       0.5877852522924731}, {-1., 0.}}]]
```

This takes the graphics primitive created above, and adds the graphics directives `RGBColor` and `EdgeForm`.

```
In[10]:= Graphics[{RGBColor[0.3, 0.5, 1], EdgeForm[Thickness[0.01]], poly}]
```

Out[10]=

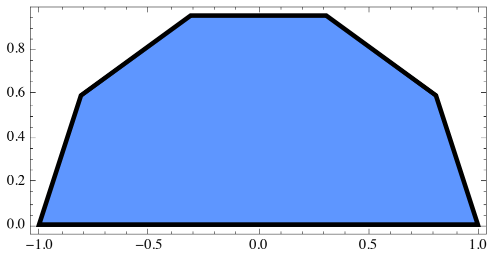


By inserting graphics directives, you can specify how particular graphical elements should be rendered. Often, however, you want to make global modifications to the way a whole graphics object is rendered. You can do this using *graphics options*.

By adding the graphics option `Frame` you can modify the overall appearance of the graphics.

```
In[11]:= Show[%, Frame -> True]
```

Out[11]=



`InputForm` shows that the option was introduced into the resulting `Graphics` object.

```
In[12]:= InputForm[%]

Out[12]//InputForm= Graphics[{RGBColor[0.3, 0.5, 1],
                      EdgeForm[Thickness[0.01]],
                      Polygon[{{1., 0.}, {0.8090169943749475,
                         0.5877852522924731}, {0.30901699437494745,
                         0.9510565162951535}, {-0.30901699437494745,
                         0.9510565162951535}, {-0.8090169943749475,
                         0.5877852522924731}, {-1., 0.}}]}, {Frame -> True}]
```

You can specify graphics options in `Show`. As a result, it is straightforward to take a single graphics object, and show it with many different choices of graphics options.

Notice however that `Show` always returns the graphics objects it has displayed. If you specify graphics options in `Show`, then these options are automatically inserted into the graphics objects that `Show` returns. As a result, if you call `Show` again on the same objects, the same graphics options will be used, unless you explicitly specify other ones. Note that in all cases new options you specify will overwrite ones already there.

`Options[`*g*`]`	give a list of all graphics options for a graphics object
`Options[`*g*`,`*opt*`]`	give the setting for a particular option

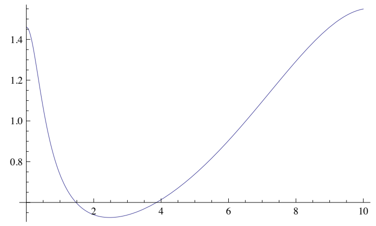
Finding the options for a graphics object.

Some graphics options can be used as options to visualization functions which generate graphics. Options which can take the right-hand side of `Automatic` are sometimes resolved into specific values by the visualization functions.

Here is a plot.

```
In[13]:= zplot = Plot[Abs[Zeta[1 / 2 + I x]], {x, 0, 10}, PlotRange → Automatic]
```

Out[13]=



Mathematica uses an internal algorithm to compute an explicit value for `PlotRange` in the resulting graphic.

```
In[14]:= Options[zplot, PlotRange]
Out[14]= {PlotRange → {{0., 10.}, {0.526253, 1.54919}}}
```

`FullGraphics[`*g*`]`	translate objects specified by graphics options into lists of explicit graphics primitives

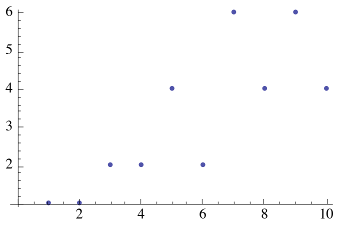
Finding the complete form of a piece of graphics.

When you use a graphics option such as `Axes`, the *Mathematica* front end automatically draws objects such as axes that you have requested. The objects are represented merely by the option values rather than by a specific list of graphics primitives. Sometimes, however, you may find it useful to represent these objects as the equivalent list of graphics primitives. The function `FullGraphics` gives the complete list of graphics primitives needed to generate a particular plot, without any options being used.

This plots a list of values.

```
In[15]:= ListPlot[Table[EulerPhi[n], {n, 10}]]
```

Out[15]=



`FullGraphics` yields a graphics object that includes graphics primitives representing axes and so on.

```
In[16]:= Short[InputForm[FullGraphics[%]], 6]
```

```
Out[16]//Short=
  Graphics[{{{{}, {Hue[0.67, 0.6, 0.6], Point[{{1., 1.}, {2., 1.}, {3., 2.}, {4., 2.}, {5.,
    4.}, {6., 2.}, {7., 6.}, {8., 4.}, {9., 6.}, {10., 4.}}]}, {}}}, {{GrayLevel[0.],
    AbsoluteThickness[0.25], Line[{{2., 1.}, {2., 1.0505635621484342}}]}, <<56>>}}]
```

Two-Dimensional Graphics Elements

`Point`[{x,y}]	point at position x, y
`Line`[{{x_1,y_1},{x_2,y_2},…}]	line through the points {x_1, y_1}, {x_2, y_2}, …
`Rectangle`[{x_{min},y_{min}},{x_{max},y_{max}}]	filled rectangle
`Polygon`[{{x_1,y_1},{x_2,y_2},…}]	filled polygon with the specified list of corners
`Circle`[{x,y},r]	circle with radius r centered at x, y
`Disk`[{x,y},r]	filled disk with radius r centered at x, y
`Raster`[{{a_{11},a_{12},…},{a_{21},…},…}]	rectangular array of gray levels between 0 and 1
`Text`[$expr$,{x,y}]	the text of $expr$, centered at x, y (see "Graphics Primitives for Text")

Basic two-dimensional graphics elements.

Here is a line primitive.

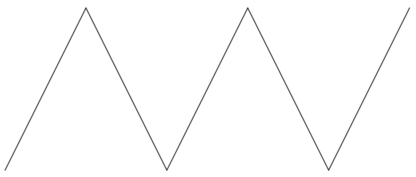
```
In[1]:= sawline = Line[Table[{n, (-1)^n}, {n, 6}]]

Out[1]= Line[{{1, -1}, {2, 1}, {3, -1}, {4, 1}, {5, -1}, {6, 1}}]
```

This shows the line as a two-dimensional graphics object.

```
In[2]:= sawgraph = Graphics[sawline]
```

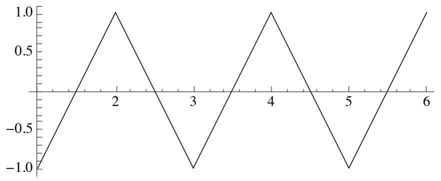
Out[2]=



This redisplays the line, with axes added.

```
In[3]:= Show[%, Axes -> True]
```

Out[3]=

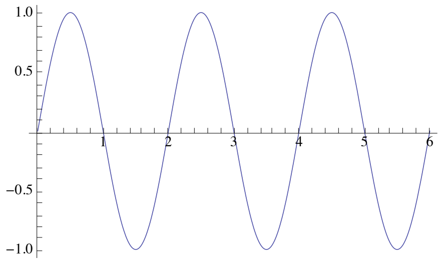


You can combine graphics objects that you have created explicitly from graphics primitives with ones that are produced by functions like `Plot`.

This produces an ordinary *Mathematica* plot.

```
In[4]:= Plot[Sin[Pi x], {x, 0, 6}]
```

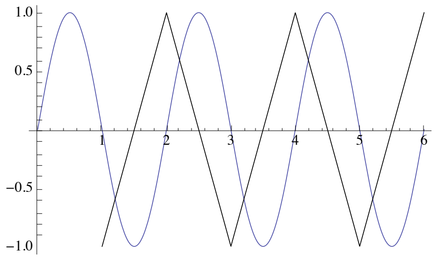
Out[4]=



This combines the plot with the sawtooth picture made above.

```
In[5]:= Show[%, sawgraph]
```

Out[5]=



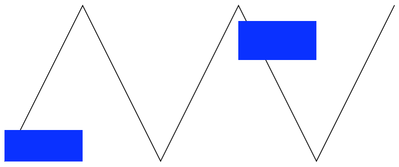
You can combine different graphical elements simply by giving them in a list. In two-dimensional graphics, *Mathematica* will render the elements in exactly the order you give them. Later elements are therefore effectively drawn on top of earlier ones.

Here are two blue `Rectangle` graphics elements.

```
In[6]:= {Blue, Rectangle[{1, -1}, {2, -0.6}], Rectangle[{4, .3}, {5, .8}]}
Out[6]= {RGBColor[0, 0, 1], Rectangle[{1, -1}, {2, -0.6}], Rectangle[{4, 0.3}, {5, 0.8}]}
```

This draws the rectangles on top of the line that was defined above.

```
In[7]:= Graphics[{sawline, %}]
Out[7]=
```



The `Polygon` graphics primitive takes a list of x, y coordinates, corresponding to the corners of a polygon. *Mathematica* joins the last corner with the first one, and then fills the resulting area.

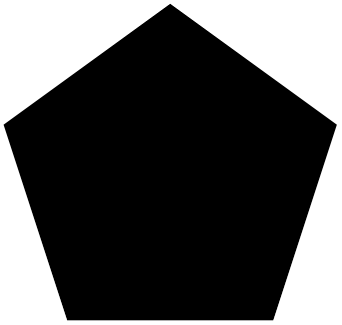
Here are the coordinates of the corners of a regular pentagon.

```
In[8]:= pentagon = Table[{Sin[2 Pi n / 5], Cos[2 Pi n / 5]}, {n, 5}]
```

Out[8]= $\left\{\left\{\sqrt{\frac{5}{8}+\frac{\sqrt{5}}{8}}, \frac{1}{4}\left(-1+\sqrt{5}\right)\right\}, \left\{\sqrt{\frac{5}{8}-\frac{\sqrt{5}}{8}}, \frac{1}{4}\left(-1-\sqrt{5}\right)\right\}, \left\{-\sqrt{\frac{5}{8}-\frac{\sqrt{5}}{8}}, \frac{1}{4}\left(-1-\sqrt{5}\right)\right\}, \left\{-\sqrt{\frac{5}{8}+\frac{\sqrt{5}}{8}}, \frac{1}{4}\left(-1+\sqrt{5}\right)\right\}, \{0, 1\}\right\}$

This displays the pentagon. With the default choice of aspect ratio, the pentagon looks somewhat squashed.

```
In[9]:= Graphics[Polygon[pentagon]]
Out[9]=
```



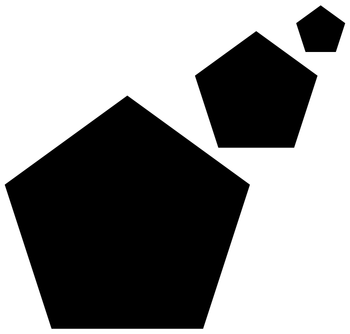
`Point[{`pt_1`,`pt_2`,…}]`	a multipoint consisting of points at pt_1, pt_2, …
`Line[{`$line_1$`,`$line_2$`,…}]`	a multiline consisting of lines $line_1$, $line_2$, …
`Polygon[{`$poly_1$`,`$poly_2$`,…}]`	a multipolygon consisting of polygons $poly_1$, $poly_2$, …

Primitives which can take multiple elements.

A large number of points can be represented by putting a list of coordinates inside of a single `Point` primitive. Similarly, a large number of lines or polygons can be represented as a list of coordinate lists. This representation is efficient and can generally be rendered more quickly by the *Mathematica* front end. Graphics directives such as `RGBColor` apply uniformly to the entire set of primitives.

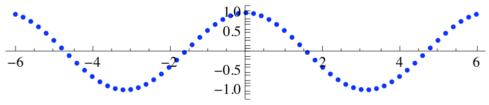
This creates a multipolygon based upon the set of coordinates defined previously.

```
In[10]:= Graphics[Polygon[{pentagon, 1 + .5 pentagon, 1.5 + .2 pentagon}]]
```

Out[10]=


Here is a multipoint which is colored blue.

```
In[11]:= Graphics[{Blue, Point[Table[{x, Cos[x]}, {x, -6, 6, .2}]]}, Axes → True]
```

Out[11]=


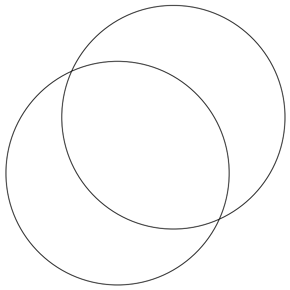
`Circle[{`x`,`y`},`r`]`	a circle with radius r centered at the point $\{x, y\}$
`Circle[{`x`,`y`},{`r_x`,`r_y`}]`	an ellipse with semi-axes r_x and r_y
`Circle[{`x`,`y`},`r`,{`$theta_1$`,`$theta_2$`}]`	a circular arc
`Circle[{`x`,`y`},{`r_x`,`r_y`},{`$theta_1$`,`$theta_2$`}]`	an elliptical arc
`Disk[{`x`,`y`},`r`]`, etc.	filled disks

Circles and disks.

This shows two circles with radius 2.

```
In[12]:= Graphics[{Circle[{0, 0}, 2], Circle[{1, 1}, 2]}]
```

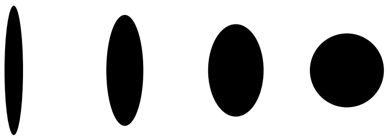
Out[12]=



This shows a sequence of disks with progressively larger semi-axes in the x direction, and progressively smaller ones in the y direction.

```
In[13]:= Graphics[Table[Disk[{3 n, 0}, {n / 4, 2 - n / 4}], {n, 4}]]
```

Out[13]=

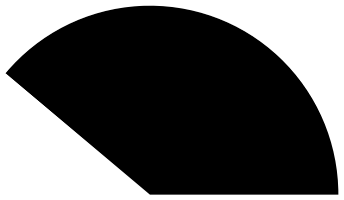


Mathematica allows you to generate arcs of circles, and segments of ellipses. In both cases, the objects are specified by starting and finishing angles. The angles are measured counterclockwise in radians with zero corresponding to the positive x direction.

This draws a 140° wedge centered at the origin.

```
In[14]:= Graphics[Disk[{0, 0}, 1, {0, 140 Degree}]]
```

Out[14]=



Raster[$\{\{a_{11}, a_{12}, \ldots\}, \{a_{21}, \ldots\}, \ldots\}$]	array of gray levels between 0 and 1
Raster[$\{\{\{a_{11}, o_{11}\}, \ldots\}, \ldots\}$]	array of gray levels with opacity between 0 and 1

`Raster[{{{`r_{11}`,`g_{11}`,`b_{11}`},…},…}]`	array of rgb values between 0 and 1
`Raster[{{{`r_{11}`,`g_{11}`,`b_{11}`,`o_{11}`},…},…}]`	array of rgb values with opacity between 0 and 1
`Raster[`*array*`,{{`x_{min}`,`y_{min}`},{`x_{max}`,`y_{max}`}},{`z_{min}`,`z_{max}`}]`	array of gray levels between z_{min} and z_{max} drawn in the rectangle defined by {x_{min}, y_{min}} and {x_{max}, y_{max}}

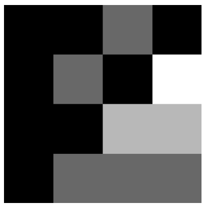
Raster-based graphics elements.

Here is a 4×4 array of values between 0 and 1.

```
In[15]:= modtab = Table[Mod[i, j] / 3, {i, 4}, {j, 4}] // N
Out[15]= {{0., 0.333333, 0.333333, 0.333333},
          {0., 0., 0.666667, 0.666667}, {0., 0.333333, 0., 1.}, {0., 0., 0.333333, 0.}}
```

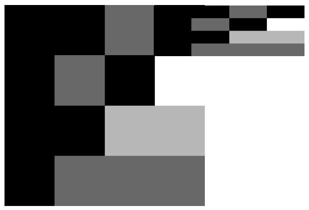
This uses the array of values as gray levels in a raster.

```
In[16]:= Graphics[Raster[modtab]]
```

Out[16]=


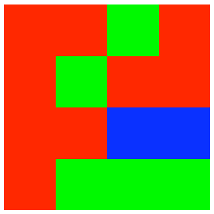
This shows two overlapping copies of the raster.

```
In[17]:= Graphics[{Raster[modtab, {{0, 0}, {2, 2}}], Raster[modtab, {{1.5, 1.5}, {3, 2}}]}]
```

Out[17]=


The `ColorFunction` option can be used to change the default way in which a `Raster` is colored.

```
In[18]:= Graphics[{Raster[modtab, ColorFunction → Hue]}]
```

Out[18]=


Graphics Directives and Options

When you set up a graphics object in *Mathematica*, you typically give a list of graphical elements. You can include in that list *graphics directives* which specify how subsequent elements in the list should be rendered.

In general, the graphical elements in a particular graphics object can be given in a collection of nested lists. When you insert graphics directives in this kind of structure, the rule is that a particular graphics directive affects all subsequent elements of the list it is in, together with all elements of sublists that may occur. The graphics directive does not, however, have any effect outside the list it is in.

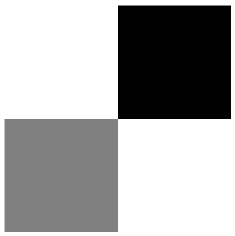
The first sublist contains the graphics directive `GrayLevel`.

```
In[1]:= {{GrayLevel[0.5], Rectangle[{0, 0}, {1, 1}]}, Rectangle[{1, 1}, {2, 2}]}
Out[1]= {{GrayLevel[0.5], Rectangle[{0, 0}, {1, 1}]}, Rectangle[{1, 1}, {2, 2}]}
```

Only the rectangle in the first sublist is affected by the `GrayLevel` directive.

```
In[2]:= Graphics[%]
```

Out[2]=



`GrayLevel[`i`]`	gray level between 0 (black) and 1 (white)
`RGBColor[`r`,`g`,`b`]`	color with specified red, green and blue components, each between 0 and 1
`Hue[`h`]`	color with hue h between 0 and 1
`Hue[`h`,`s`,`b`]`	color with specified hue, saturation and brightness, each between 0 and 1

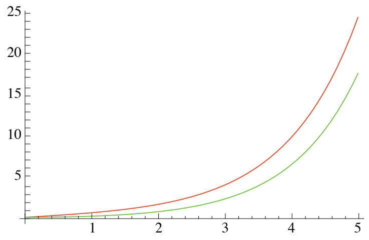
Basic *Mathematica* color specifications.

Mathematica accepts the names of many colors directly as color specifications. These color names, such as `Red`, `Gray`, `LightGreen` and `Purple`, are implemented as variables which evaluate to an `RGBColor` specification. The color names can be used interchangeably with color directives.

The first plot is colored with a color name, while the second one has a fine-tuned `RGBColor` specification.

```
In[3]:= Plot[{BesselI[1, x], BesselI[2, x]}, {x, 0, 5},
         PlotStyle -> {{Red}, {RGBColor[0.3, 0.7, 0.1]}}]
```

Out[3]=



The function `Hue[`h`]` provides a convenient way to specify a range of colors using just one parameter. As h varies from 0 to 1, `Hue[`h`]` runs through red, yellow, green, cyan, blue, magenta, and back to red again. `Hue[`h`, `s`, `b`]` allows you to specify not only the "hue", but also the "saturation" and "brightness" of a color. Taking the saturation to be equal to one gives the deepest colors; decreasing the saturation toward zero leads to progressively more "washed out" colors.

When you give a graphics directive such as `RGBColor`, it affects *all* subsequent graphical elements that appear in a particular list. *Mathematica* also supports various graphics directives which affect only specific types of graphical elements.

The graphics directive `PointSize[`d`]` specifies that all `Point` elements which appear in a graphics object should be drawn as circles with diameter d. In `PointSize`, the diameter d is measured as a fraction of the width of your whole plot.

Mathematica also provides the graphics directive `AbsolutePointSize[`d`]`, which allows you to specify the "absolute" diameter of points, measured in fixed units. The units are $\frac{1}{72}$ of an inch, approximately printer's points.

`PointSize[`d`]`	give all points a diameter d as a fraction of the width of the whole plot
`AbsolutePointSize[`d`]`	give all points a diameter d measured in absolute units

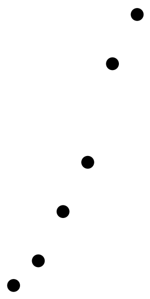
Graphics directives for points.

Here is a list of points.

```
In[4]:= Table[Point[{n, Prime[n]}], {n, 6}]
Out[4]= {Point[{1, 2}], Point[{2, 3}], Point[{3, 5}], Point[{4, 7}], Point[{5, 11}], Point[{6, 13}]}
```

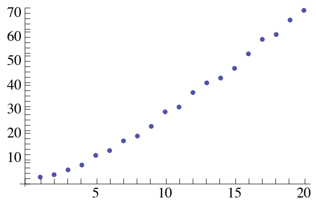
This makes each point have a diameter equal to one-tenth of the width of the plot.

```
In[5]:= Graphics[{PointSize[0.1], %}, PlotRange -> All]
Out[5]=
```



Here each point has size 3 in absolute units.

```
In[6]:= ListPlot[Table[Prime[n], {n, 20}], PlotStyle -> AbsolutePointSize[3]]
Out[6]=
```



`Thickness[w]`	give all lines a thickness w as a fraction of the width of the whole plot
`AbsoluteThickness[w]`	give all lines a thickness w measured in absolute units
`Dashing[{`w_1`,`w_2`,…}]`	show all lines as a sequence of dashed segments, with lengths w_1, w_2, …
`AbsoluteDashing[{`w_1`,`w_2`,…}]`	use absolute units to measure dashed segments

Graphics directives for lines.

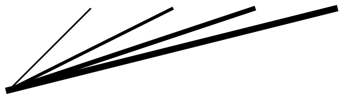
This generates a list of lines with different absolute thicknesses.

```
In[7]:= Table[{AbsoluteThickness[n], Line[{{0, 0}, {n, 1}}]}, {n, 4}]
Out[7]= {{AbsoluteThickness[1], Line[{{0, 0}, {1, 1}}]}, {AbsoluteThickness[2], Line[{{0, 0}, {2, 1}}]},
         {AbsoluteThickness[3], Line[{{0, 0}, {3, 1}}]}, {AbsoluteThickness[4], Line[{{0, 0}, {4, 1}}]}}
```

Here is a picture of the lines.

```
In[8]:= Graphics[%]
```

Out[8]=

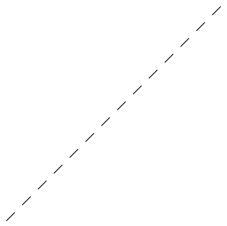


The `Dashing` graphics directive allows you to create lines with various kinds of dashing. The basic idea is to break lines into segments which are alternately drawn and omitted. By changing the lengths of the segments, you can get different line styles. `Dashing` allows you to specify a sequence of segment lengths. This sequence is repeated as many times as necessary in drawing the whole line.

This gives a dashed line with a succession of equal-length segments.

```
In[9]:= Graphics[{Dashing[{0.05, 0.05}], Line[{{-1, -1}, {1, 1}}]}]
```

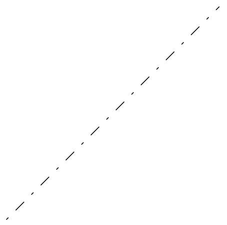
Out[9]=



This gives a dot-dashed line.

```
In[10]:= Graphics[{Dashing[{0.01, 0.05, 0.05, 0.05}], Line[{{-1, -1}, {1, 1}}]}]
```

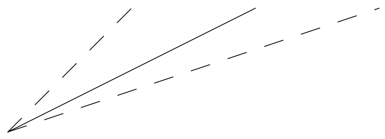
Out[10]=



`Dashing` can be turned off by specifying an empty list. Here, `Dashing` is turned off for only the second line.

```
In[11]:= Graphics[{Dashing[{0.05}], Line[{{0, 0}, {1, 1}}],
           {Dashing[{}], Line[{{0, 0}, {2, 1}}]},
           Line[{{0, 0}, {3, 1}}]}]
```

Out[11]=

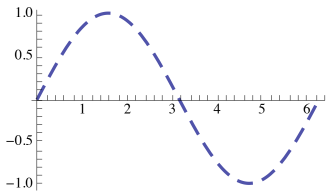


Graphics directives which require a numerical size specification can also accept values of `Tiny`, `Small`, `Medium`, and `Large`. For each directive, these values have been fine-tuned to produce an appearance which will seem appropriate to the human eye.

This specifies a large thickness with medium dashing.

```
In[12]:= Plot[Sin[x], {x, 0, 2 Pi}, PlotStyle → {{Dashing[{Medium}], Thickness[Large]}}]
```

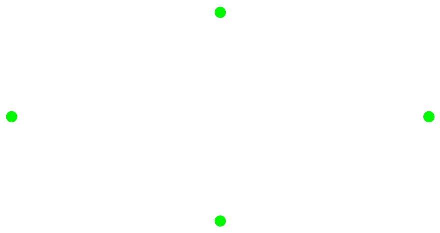
Out[12]=



This specifies that the entire multipoint should use large, green points.

```
In[13]:= Graphics[{PointSize[Large], Green, Point[{{0, 0}, {1, 0.5}, {2, 0}, {1, -.5}}]}]
```

Out[13]=



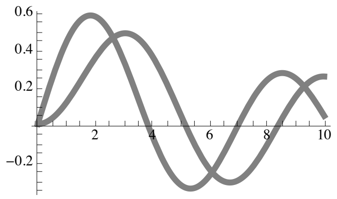
One way to use *Mathematica* graphics directives is to insert them directly into the lists of graphics primitives used by graphics objects. Sometimes, however, you want the graphics directives to be applied more globally, and for example to determine the overall "style" with which a particular type of graphical element should be rendered. There are typically graphics options which can be set to specify such styles in terms of lists of graphics directives.

`PlotStyle->`*style*	specify a style to be used for all curves in `Plot`
`PlotStyle->{{`$style_1$`},{`$style_2$`},…}`	specify styles to be used (cyclically) for a sequence of curves in `Plot`
`MeshStyle->`*style*	specify a style to be used for a mesh in density and surface graphics
`BoxStyle->`*style*	specify a style to be used for the bounding box in three-dimensional graphics

Some graphics options for specifying styles.

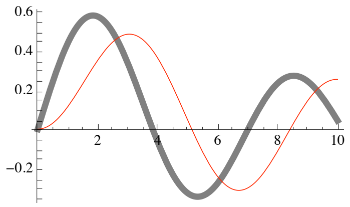
This generates a plot in which all curves are specified to use the same style.

In[14]:=
```
Plot[{BesselJ[1, x], BesselJ[2, x]},
 {x, 0, 10}, PlotStyle -> {{Thickness[0.02], Gray}}]
```

Out[14]=


A different `PlotStyle` expression can be used to give specific styles to each curve.

In[15]:=
```
Plot[{BesselJ[1, x], BesselJ[2, x]}, {x, 0, 10},
 PlotStyle -> {{Thickness[0.02], Gray}, {Red}}]
```

Out[15]=


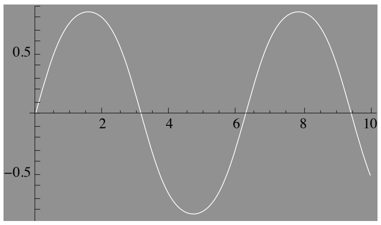
The various "style options" allow you to specify how particular graphical elements in a plot should be rendered. *Mathematica* also provides options that affect the rendering of the whole plot.

`Background->`*color*	specify the background color for a plot
`BaseStyle->`*color*	specify the base style for a plot, affecting elements not affected by `PlotStyle`
`Prolog->`*g*	give graphics to render before a plot is started
`Epilog->`*g*	give graphics to render after a plot is finished

Graphics options that affect whole plots.

This draws the plot in white on a gray background.

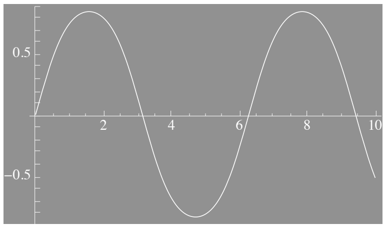
In[16]:=
```
Plot[Sin[Sin[x]], {x, 0, 10}, Background → Gray, PlotStyle → White]
```

Out[16]=


This makes the axes white as well.

```
In[17]:= Show[%, BaseStyle → White]
```

Out[17]=



Coordinate Systems for Two-Dimensional Graphics

When you set up a graphics object in *Mathematica*, you give coordinates for the various graphical elements that appear. When *Mathematica* renders the graphics object, it has to translate the original coordinates you gave into "display coordinates" which specify where each element should be placed in the final display area.

`PlotRange->`$\{\{x_{min}, x_{max}\}, \{y_{min}, y_{max}\}\}$	the range of original coordinates to include in the plot

Option which determines translation from original to display coordinates.

When *Mathematica* renders a graphics object, one of the first things it has to do is to work out what range of original x and y coordinates it should actually display. Any graphical elements that are outside this range will be clipped, and not shown.

The option `PlotRange` specifies the range of original coordinates to include. As discussed in "Options for Graphics", the default setting is `PlotRange -> Automatic`, which makes *Mathematica* try to choose a range which includes all "interesting" parts of a plot, while dropping "outliers". By setting `PlotRange -> All`, you can tell *Mathematica* to include everything. You can also give explicit ranges of coordinates to include.

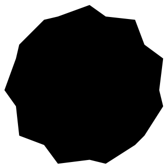
This sets up a polygonal object whose corners have coordinates between roughly ± 1.

```
In[1]:= obj = Polygon[Table[{Sin[n Pi / 10], Cos[n Pi / 10]} + 0.05 (-1) ^n, {n, 20}]];
```

In this case, the polygonal object fills almost the whole display area.

```
In[2]:= Graphics[obj]
```

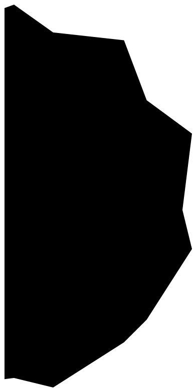
Out[2]=



Specifying an explicit `PlotRange` allows you to zoom in on a section of a graphic.

```
In[3]:= Graphics[obj, PlotRange → {{0, 1}, All}]
```

Out[3]=



`AspectRatio->`r	make the ratio of height to width for the display area equal to r
`AspectRatio->Automatic`	determine the shape of the display area from the original coordinate system

Specifying the shape of the display area.

What we have discussed so far is how *Mathematica* translates the original coordinates you specify into positions in the final display area. What remains to discuss, however, is what the final display area is like.

On most computer systems, there is a certain fixed region of screen or paper into which the *Mathematica* display area must fit. How it fits into this region is determined by its "shape" or aspect ratio. In general, the option `AspectRatio` specifies the ratio of height to width for the final display area.

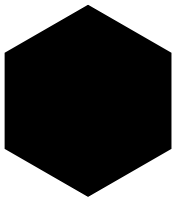
It is important to note that the setting of `AspectRatio` does not affect the meaning of the scaled or display coordinates. These coordinates always run from 0 to 1 across the display area. What `AspectRatio` does is to change the shape of this display area.

For two-dimensional graphics, `AspectRatio` is set by default to `Automatic`. This determines the aspect ratio from the original coordinate system used in the plot instead of setting it at a fixed value. One unit in the x direction in the original coordinate system corresponds to the same distance in the final display as one unit in the y direction. In this way, objects that you define in the original coordinate system are displayed with their "natural shape".

This generates a graphic object corresponding to a regular hexagon. With the default value of `AspectRatio -> Automatic`, the aspect ratio of the final display area is determined from the original coordinate system, and the hexagon is shown with its "natural shape".

```
In[4]:= Graphics[Polygon[Table[{Sin[n Pi / 3], Cos[n Pi / 3]}, {n, 6}]]]
```

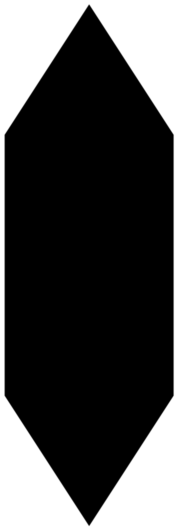
Out[4]=



This renders the hexagon in a display area whose height is three times its width.

```
In[5]:= Show[%, AspectRatio -> 3]
```

Out[5]=



Sometimes, you may find it convenient to specify the display coordinates for a graphical element directly. You can do this by using scaled coordinates `Scaled[{sx, sy}]` rather than `{x, y}`. The scaled coordinates are defined to run from 0 to 1 in x and y, with the origin taken to be at the lower-left corner of the plot range.

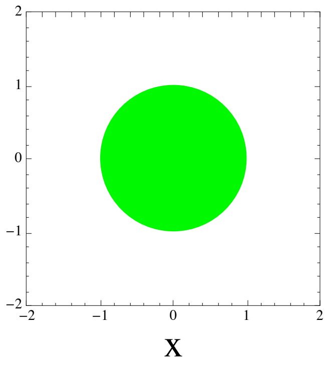
`{x,y}`	original coordinates
`Scaled[{sx,sy}]`	coordinates scaled to the plot range
`ImageScaled[{sx,sy}]`	coordinates scaled to the display area

Coordinate systems for two-dimensional graphics.

The display area is significantly larger than the plot range due to the frame label.

```
In[6]:= g = Graphics[{Green, Disk[]}, PlotRange → 2,
          Frame → True, FrameLabel → {Style["x", Large]}]
```

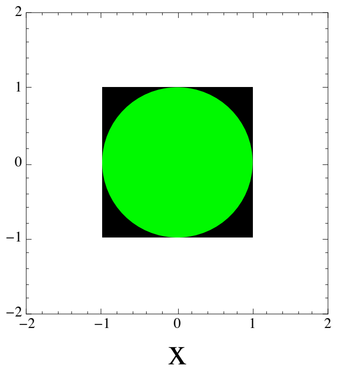
Out[6]=



Using `Scaled` coordinates, the rectangle falls at the origin, which is at the center of the specified plot range.

```
In[7]:= Show[g, Prolog → {Rectangle[Scaled[{0.25, 0.25}], Scaled[{0.75, 0.75}]]}]
```

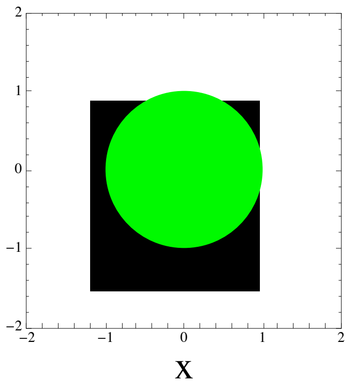
Out[7]=



Using `ImageScaled` coordinates, the rectangle falls at exactly the center of the graphic, which does not coincide with the center of the plot range.

```
In[8]:= Show[g, Prolog → {Rectangle[ImageScaled[{0.25, 0.25}], ImageScaled[{0.75, 0.75}]]}]
```

Out[8]=



When you use `{x, y}`, `Scaled[{sx, sy}]` or `ImageScaled[{sx, sy}]`, you are specifying position either completely in original coordinates, or completely in scaled coordinates. Sometimes, however, you may need to use a combination of these coordinate systems. For example, if you want to draw a line at a particular point whose length is a definite fraction of the width of the plot, you will have to use original coordinates to specify the basic position of the line, and scaled coordinates to specify its length.

You can use `Scaled[{dsx, dsy}, {x, y}]` to specify a position using a mixture of original and scaled coordinates. In this case, `{x, y}` gives a position in original coordinates, and `{dsx, dsy}` gives the offset from the position in scaled coordinates.

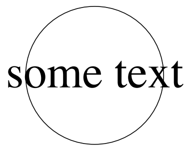
`Circle[{x,y},Scaled[sx]]`	a circle whose radius is scaled to the width of the plot range
`Disk[{x,y},Scaled[sx]]`	a disk whose radius is scaled to the width of the plot range
`FontSize->Scaled[sx]`	specification for a font size scaled to the width of the plot range

Some places where **`Scaled`** can be used with a single argument.

Both the radius of the circle and the size of the font are specified in `Scaled` values.

```
In[9]:= Graphics[
         {Circle[{0, 0}, Scaled[0.3]], FontSize → Scaled[0.2], Text["some text", {0, 0}]}]
```

Out[9]=

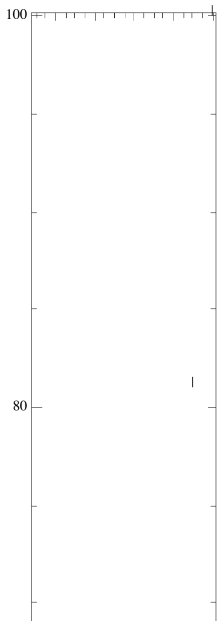


`Scaled[{sdx,sdy},{x,y}]`	scaled offset from original coordinates
`ImageScaled[{sdx,sdy},{x,y}]`	image scaled offset from original coordinates
`Offset[{adx,ady},{x,y}]`	absolute offset from original coordinates
`Offset[{adx,ady},Scaled[{sx,sy}]]`	absolute offset from scaled coordinates
`Offset[{adx,ady}, ImageScaled[{sx,sy}]]`	absolute offset from image scaled coordinates

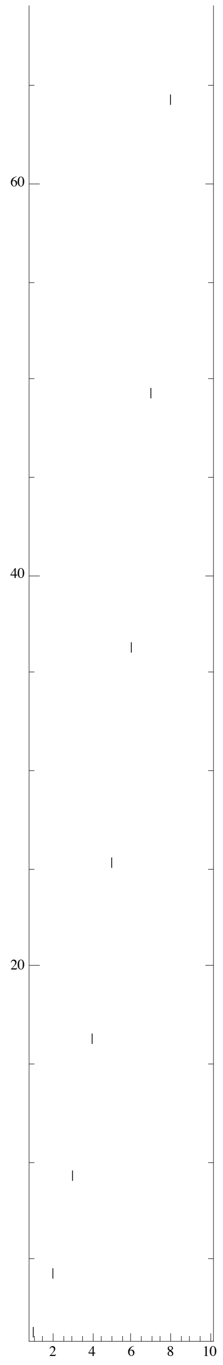
Positions specified as offsets.

Each line drawn here has an absolute length of 6 printer's points.

```
In[10]:= Graphics[Table[Line[{{x, x^2}, Offset[{0, 6}, {x, x^2}]}], {x, 10}], Frame -> True]
```



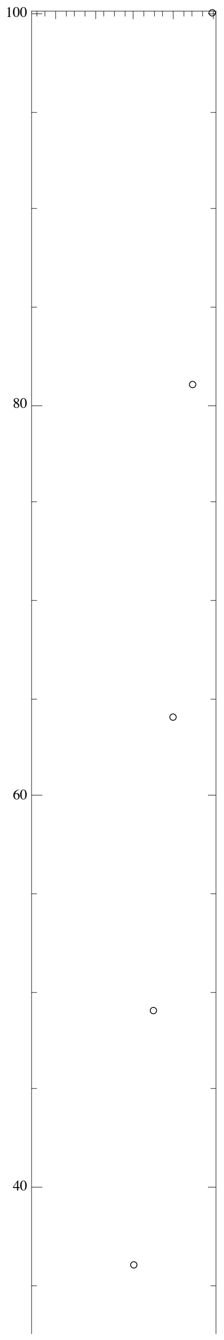
Out[10]=

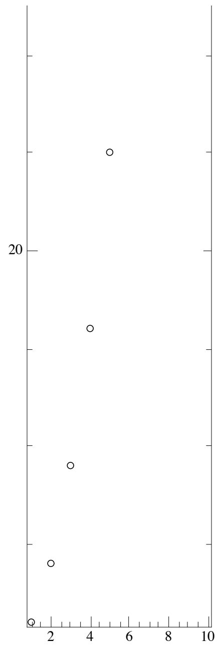


You can also use `Offset` inside `Circle` with just one argument to create a circle with a certain absolute radius.

```
In[11]:= Graphics[Table[Circle[{x, x^2}, Offset[{2, 2}]], {x, 10}], Frame -> True]
```

Out[11]=





In most kinds of graphics, you typically want the relative positions of different objects to adjust automatically when you change the coordinates or the overall size of your plot. But sometimes you may instead want the offset from one object to another to be constrained to remain fixed. This can be the case, for example, when you are making a collection of plots in which you want certain features to remain consistent, even though the different plots have different forms.

`Offset[{`*adx*`,` *ady*`},` *position*`]` allows you to specify the position of an object by giving an absolute offset from a position that is specified in original or scaled coordinates. The units for the offset are printer's points, equal to $\frac{1}{72}$ of an inch.

When you give text in a plot, the size of the font that is used is also specified in printer's points. Therefore, a 10-point font, for example, has letters whose basic height is 10 printer's points. You can use `Offset` to move text around in a plot, and to create plotting symbols or icons which match the size of the text.

Using scaled coordinates, you can specify the sizes of graphical elements as fractions of the size of the display area. You cannot, however, tell *Mathematica* the actual physical size at which a particular graphical element should be rendered. Of course, this size ultimately depends on the details of your graphics output device, and cannot be determined for certain within *Mathemat-*

ica. Nevertheless, graphics directives such as `AbsoluteThickness` discussed in "Graphics Directives and Options" do allow you to indicate “absolute sizes” to use for particular graphical elements. The sizes you request in this way will be respected by most, but not all, output devices. (For example, if you optically project an image, it is neither possible nor desirable to maintain the same absolute size for a graphical element within it.)

Labeling Two-Dimensional Graphics

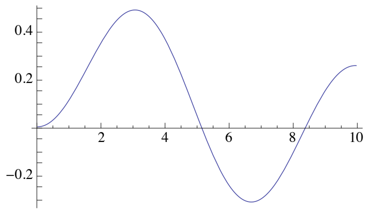
`Axes->True`	give a pair of axes
`GridLines->Automatic`	draw grid lines on the plot
`Frame->True`	put axes on a frame around the plot
`PlotLabel->"`*text*`"`	give an overall label for the plot

Ways to label two-dimensional plots.

Here is a plot, using the default `Axes -> True`.

```
In[1]:= bp = Plot[BesselJ[2, x], {x, 0, 10}]
```

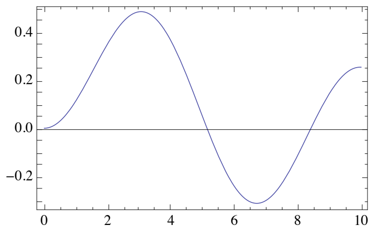
Out[1]=



Setting `Frame -> True` generates a frame with axes, and removes tick marks from the ordinary axes.

```
In[2]:= Show[bp, Frame -> True]
```

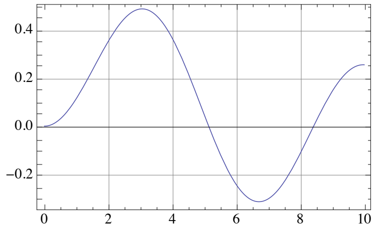
Out[2]=



This includes grid lines, which are shown in light gray.

```
In[3]:= Show[%, GridLines -> Automatic]
```

Out[3]=



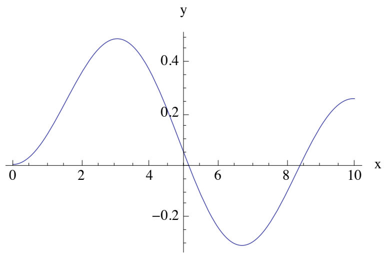
`Axes->False`	draw no axes
`Axes->True`	draw both x and y axes
`Axes->{False,True}`	draw a y axis but no x axis
`AxesOrigin->Automatic`	choose the crossing point for the axes automatically
`AxesOrigin->{`*x*`,`*y*`}`	specify the crossing point
`AxesStyle->`*style*	specify the style for axes
`AxesStyle->{`*xstyle*`,`*ystyle*`}`	specify individual styles for axes
`AxesLabel->None`	give no axis labels
`AxesLabel->`*ylabel*	put a label on the y axis
`AxesLabel->{`*xlabel*`,`*ylabel*`}`	put labels on both x and y axes

Options for axes.

This makes the axes cross at the point `{5, 0}`, and puts a label on each axis.

```
In[4]:= Show[bp, AxesOrigin -> {5, 0}, AxesLabel -> {"x", "y"}]
```

Out[4]=



`Ticks->None`	draw no tick marks
`Ticks->Automatic`	place tick marks automatically
`Ticks->{`*xticks*`,`*yticks*`}`	tick mark specifications for each axis

Settings for the `Ticks` option.

With the default setting `Ticks -> Automatic`, *Mathematica* creates a certain number of major and minor tick marks, and places them on axes at positions which yield the minimum number of decimal digits in the tick labels. In some cases, however, you may want to specify the positions and properties of tick marks explicitly. You will need to do this, for example, if you want to have tick marks at multiples of π, or if you want to put a nonlinear scale on an axis.

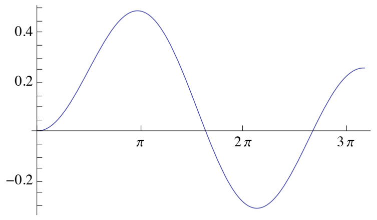
`None`	draw no tick marks
`Automatic`	place tick marks automatically
$\{x_1, x_2, \ldots\}$	draw tick marks at the specified positions
$\{\{x_1, label_1\}, \{x_2, label_2\}, \ldots\}$	draw tick marks with the specified labels
$\{\{x_1, label_1, len_1\}, \ldots\}$	draw tick marks with the specified scaled lengths
$\{\{x_1, label_1, \{plen_1, mlen_1\}\}, \ldots\}$	draw tick marks with the specified lengths in the positive and negative directions
$\{\{x_1, label_1, len_1, style_1\}, \ldots\}$	draw tick marks with the specified styles
func	a function to be applied to x_{min}, x_{max} to get the tick mark option

Tick mark options for each axis.

This gives tick marks at specified positions on the x axis, and chooses the tick marks automatically on the y axis.

```
In[5]:= Show[bp, Ticks -> {{0, Pi, 2 Pi, 3 Pi}, Automatic}]
```

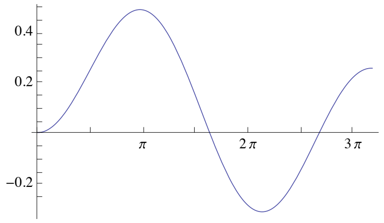
Out[5]=



This adds tick marks with no labels at multiples of $\pi/2$.

```
In[6]:= Show[bp,
         Ticks -> {{0, {Pi / 2, ""}, Pi, {3 Pi / 2, ""}, 2 Pi, {5 Pi / 2, ""}, 3 Pi}, Automatic}]
```

Out[6]=



Particularly when you want to create complicated tick mark specifications, it is often convenient to define a "tick mark function" which creates the appropriate tick mark specification given the minimum and maximum values on a particular axis.

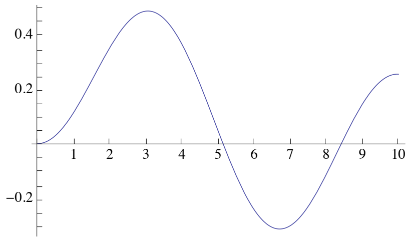
This defines a function which gives a list of tick mark positions with a spacing of 1.

```
In[7]:= units[xmin_, xmax_] := Range[Floor[xmin], Floor[xmax], 1]
```

This uses the `units` function to specify tick marks for the x axis.

```
In[8]:= Show[bp, Ticks -> {units, Automatic}]
```

Out[8]=



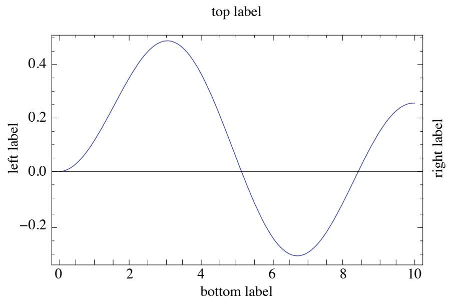
`Frame->False`	draw no frame
`Frame->True`	draw a frame around the plot
`FrameStyle->`*style*	specify a style for the frame
`FrameStyle->{{`*left*`,`*right*`},{`*bottom*`,`*top*`}}`	specify styles for each edge of the frame
`FrameLabel->None`	give no frame labels
`FrameLabel->{{`*left*`,`*right*`},{`*bottom*`,`*top*`}}`	put labels on edges of the frame
`RotateLabel->False`	do not rotate text in labels
`FrameTicks->None`	draw no tick marks on frame edges
`FrameTicks->Automatic`	position tick marks automatically
`FrameTicks->{{`*left*`,`*right*`},{`*bottom*`,`*top*`}}`	specify tick marks for frame edges

Options for frame axes.

The `Axes` option allows you to draw a single pair of axes in a plot. Sometimes, however, you may instead want to show the scales for a plot on a frame, typically drawn around the whole plot. The option `Frame` allows you effectively to draw four axes, corresponding to the four edges of the frame around a plot.

This draws frame axes, and labels each of them.

```
In[9]:= Show[bp, Frame -> True,
         FrameLabel -> {{"left label", "right label"}, {"bottom label", "top label"}}]
```

Out[9]=


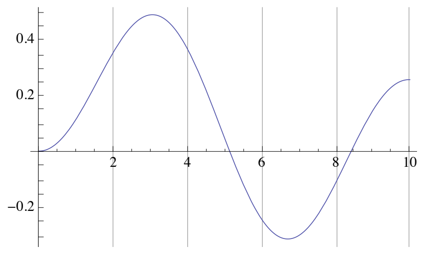
`GridLines->None`	draw no grid lines
`GridLines->Automatic`	position grid lines automatically
`GridLines->{`*xgrid*`,`*ygrid*`}`	specify grid lines in analogy with tick marks

Options for grid lines.

Grid lines in *Mathematica* work very much like tick marks. As with tick marks, you can specify explicit positions for grid lines. There is no label or length to specify for grid lines. However, you can specify a style.

This generates x but not y grid lines.

```
In[10]:= Show[bp, GridLines -> {Automatic, None}]
```

Out[10]=


Insetting Objects in Graphics

"Redrawing and Combining Plots" describes how you can make regular arrays of plots using `GraphicsGrid`. Using the `Inset` graphics primitive, however, you can combine and superimpose plots in any way.

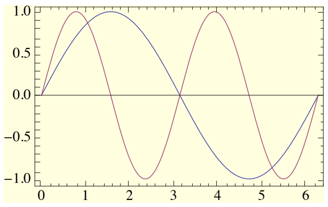
`Inset`[*obj*, *pos*]	specifies that the inset should be placed at position *pos* in the graphic
`Inset`[*obj*, *pos*, *opos*, *size*]	render an object with a given *size* so that point *opos* in *obj* is positioned at point *pos* in the containing graphic
`Inset`[*obj*, *pos*, *opos*, *size*, *dirs*]	specifies that the axes of the inset should be oriented in directions *dirs*

Creating an inset.

Here is a plot.

```
In[1]:= p1 = Plot[{Sin[x], Sin[2 x]}, {x, 0, 2 π},
          ImageSize → 200, Frame → True, Background → LightYellow]
```

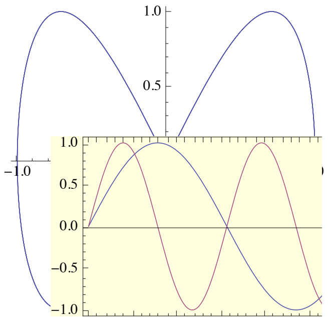
Out[1]=



This creates a plot within a parametric plot.

```
In[2]:= ParametricPlot[{Sin[x], Sin[2 x]}, {x, 0, 4 π}, Epilog → Inset[p1, {.3, -.5}]]
```

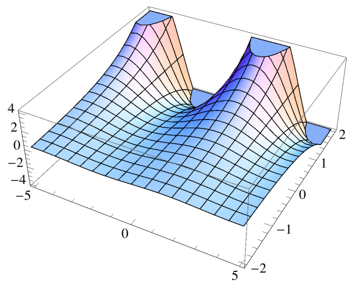
Out[2]=



Here is a three-dimensional plot.

```
In[3]:= p3 = Plot3D[Sin[x] Exp[y], {x, -5, 5}, {y, -2, 2}]
```

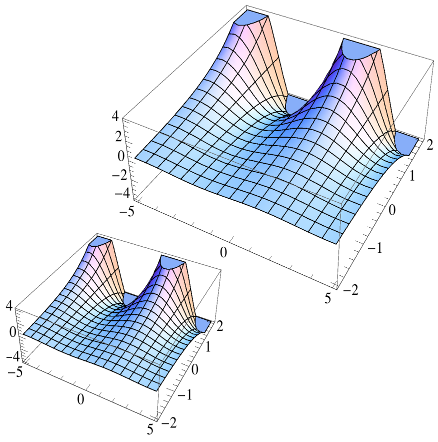
Out[3]=



This creates a two-dimensional graphics object that contains two differently sized copies of the three-dimensional plot.

```
In[4]:= Graphics[{Inset[p3, -{1, 1}, Center, {2, 2}],
          Inset[p3, {.5, .5}, Center, {3, 3}]}, PlotRange → 2]
```

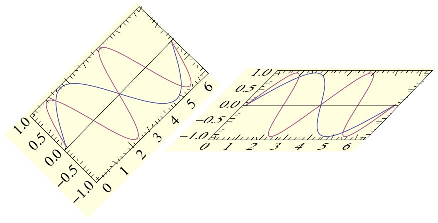
Out[4]=



Here are rotated and skewed plots inset in a graphic.

```
In[5]:= Graphics[{
          Inset[p1, {1, 0}, Center, {1, 1}, {1, 1}],
          Inset[p1, {2, 0}, Center, {1, 1}, {{1, 0}, {1, 1}}]
         }]
```

Out[5]=



Mathematica can render plots, arbitrary 2D or 3D graphics, cells, and text within an `Inset`. Notice that in general the display area for graphics objects will be sized so as to touch at least one pair of edges of the `Inset`.

Density and Contour Plots

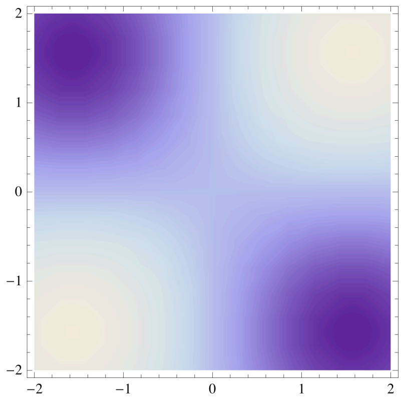
`DensityPlot[`f`,{`x`,`x_{min}`,`x_{max}`},{`y`,`y_{min}`,`y_{max}`}]`	make a density plot of f
`ContourPlot[`f`,{`x`,`x_{min}`,`x_{max}`},{`y`,`y_{min}`,`y_{max}`}]`	make a contour plot of f as a function of x and y

Density and contour plots.

This gives a density plot of $\sin(x)\sin(y)$. Lighter regions show higher values of the function.

```
In[1]:= DensityPlot[Sin[x] Sin[y], {x, -2, 2}, {y, -2, 2}]
```

Out[1]=



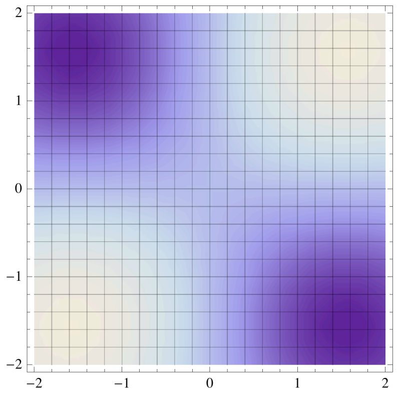
option name	*default value*	
`ColorFunction`	`Automatic`	what colors to use for shading; `Hue` uses a sequence of hues
`Mesh`	`None`	whether to draw a mesh
`PlotPoints`	`Automatic`	number of initial sample points in each direction
`MaxRecursion`	`Automatic`	the maximum number of recursive subdivision steps to do

Some options for `DensityPlot`.

You can include a mesh like this.

```
In[2]:= DensityPlot[Sin[x] Sin[y], {x, -2, 2}, {y, -2, 2}, Mesh → 19]
```

Out[2]=

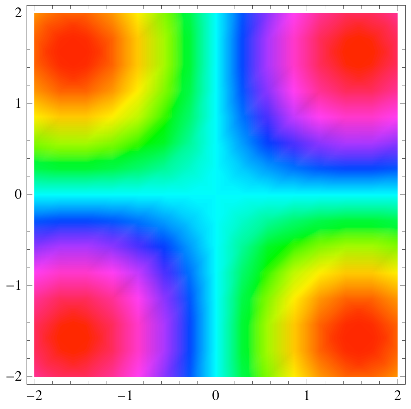


In a density plot, the color of each point represents the value at that point of the function being plotted. By default, the color ranges from black to white through intermediate shades of blue as the value of the function increases. In general, however, you can specify other “color maps” for the relation between the value at a point and its color. The option `ColorFunction` allows you to specify a function which is applied to the function value to find the color at any point. The color function may return any *Mathematica* color directive, such as `GrayLevel`, `Hue` or `RGBColor`. A common setting to use is `ColorFunction -> Hue`.

This uses different hues to represent different values.

```
In[3]:= DensityPlot[Sin[x] Sin[y], {x, -2, 2}, {y, -2, 2}, ColorFunction → Hue]
```

Out[3]=



A significant resource for customized color functions is the `ColorData` function. `ColorData` provides many customized sets of colors which can be used directly by `ColorFunction`.

This shows a list of the gradients which can be accessed using `ColorData`.

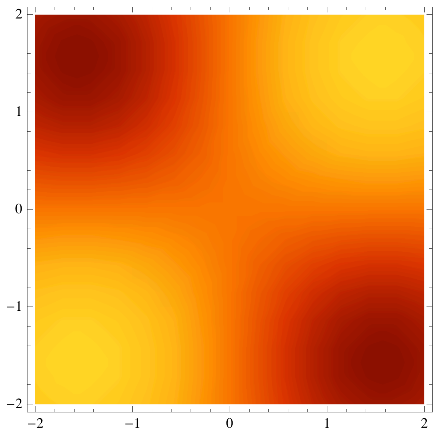
```
In[4]:= ColorData["Gradients"]

Out[4]= {DarkRainbow, Rainbow, Pastel, Aquamarine, BrassTones, BrownCyanTones, CherryTones, CoffeeTones,
         FuchsiaTones, GrayTones, GrayYellowTones, GreenPinkTones, PigeonTones, RedBlueTones,
         RustTones, SiennaTones, ValentineTones, AlpineColors, ArmyColors, AtlanticColors,
         AuroraColors, AvocadoColors, BeachColors, CandyColors, CMYKColors, DeepSeaColors, FallColors,
         FruitPunchColors, IslandColors, LakeColors, MintColors, NeonColors, PearlColors, PlumColors,
         RoseColors, SolarColors, SouthwestColors, StarryNightColors, SunsetColors, ThermometerColors,
         WatermelonColors, RedGreenSplit, DarkTerrain, GreenBrownTerrain, LightTerrain,
         SandyTerrain, BlueGreenYellow, LightTemperatureMap, TemperatureMap, BrightBands, DarkBands}
```

This `DensityPlot` is identical to the one above, but uses the `"SolarColors"` gradient.

```
In[5]:= DensityPlot[Sin[x] Sin[y], {x, -2, 2},
         {y, -2, 2}, ColorFunction → ColorData["SolarColors"]]
```

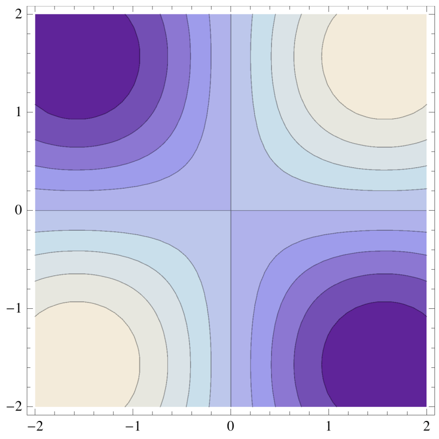
Out[5]=



This gives a contour plot of the function.

```
In[6]:= ContourPlot[Sin[x] Sin[y], {x, -2, 2}, {y, -2, 2}]
```

Out[6]=



A contour plot gives you essentially a "topographic map" of a function. The contours join points on the surface that have the same height. The default is to have contours corresponding to a sequence of equally spaced z values. Contour plots produced by *Mathematica* are by default shaded, in such a way that regions with higher z values are lighter.

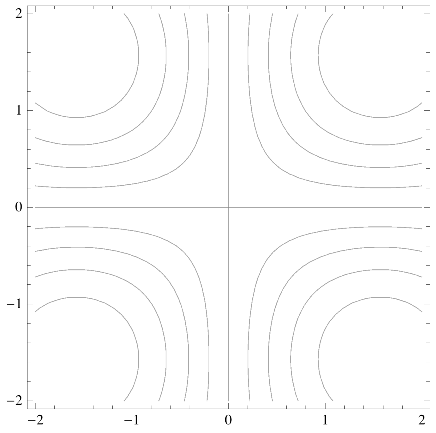
option name	*default value*	
`ColorFunction`	`Automatic`	what colors to use for shading; `Hue` uses a sequence of hues
`Contours`	`Automatic`	the total number of contours, or the list of z values for contours
`PlotRange`	`{Full,Full,Automatic}`	the range of values to be included; you can specify $\{z_{min}, z_{max}\}$, `All` or `Automatic`, or a list $\{xrange, yrange, zrange\}$
`ContourShading`	`Automatic`	how to shade the regions; `None` leaves the regions blank, or a list of colors can be provided
`PlotPoints`	`Automatic`	number of initial sample points in each direction
`MaxRecursion`	`Automatic`	the maximum number of recursive subdivi-sion steps to do

Some options for `ContourPlot`.

This shows the plot with no shading.

```
In[7]:= ContourPlot[Sin[x] Sin[y], {x, -2, 2}, {y, -2, 2}, ContourShading → None]
```

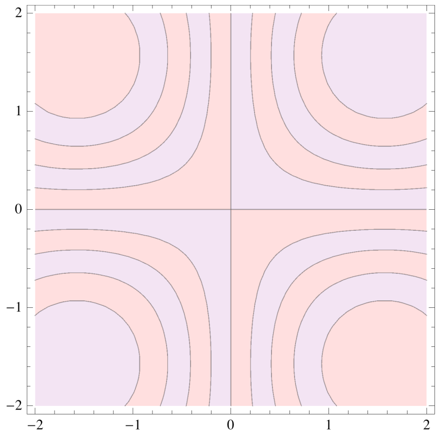
Out[7]=



This cycles the colors used for contour regions between light red and light purple.

```
In[8]:= ContourPlot[Sin[x] Sin[y], {x, -2, 2},
         {y, -2, 2}, ContourShading → {LightRed, LightPurple}]
```

Out[8]=



Both `DensityPlot` and `ContourPlot` use an adaptive algorithm that subdivides parts of the plot region to obtain more sample points for a smoother representation of the function you are plotting. Because the number of sample points is always finite, however, it is possible that features of your function will sometimes be missed. When necessary, you can increase the number of sample points by increasing the values of the `PlotPoints` and `MaxRecursion` options.

One point to notice is that whereas a curve generated by `Plot` may be inaccurate if your function varies too quickly in a particular region, the shape of contours generated by `ContourPlot` can be inaccurate if your function varies too slowly. A rapidly varying function gives a regular pattern of contours, but a function that is almost flat can give irregular contours. You can typically overcome this by increasing the value of `PlotPoints` or `MaxRecursion`.

Three-Dimensional Graphics Primitives

One of the most powerful aspects of graphics in *Mathematica* is the availability of three-dimensional as well as two-dimensional graphics primitives. By combining three-dimensional graphics primitives, you can represent and render three-dimensional objects in *Mathematica*.

`Point[{`x`,`y`,`z`}]`	point with coordinates x, y, z
`Line[{{`x_1`,`y_1`,`z_1`},{`x_2`,`y_2`,`z_2`},`…`}]`	line through the points $\{x_1, y_1, z_1\}$, $\{x_2, y_2, z_2\}$, …
`Polygon[{{`x_1`,`y_1`,`z_1`},{`x_2`,`y_2`,`z_2`},`…`}]`	filled polygon with the specified list of corners
`Cuboid[{`x_{min}`,`y_{min}`,`z_{min}`},{`x_{max}`,`y_{max}`,`z_{max}`}]`	cuboid
`Text[`*expr*`,{`x`,`y`,`z`}]`	text at position $\{x, y, z\}$ (see "Graphics Primitives for Text")

Three-dimensional graphics elements.

Every time you evaluate `rcoord`, it generates a random coordinate in three dimensions.

```
In[1]:= rcoord := RandomReal[1., {3}]
```

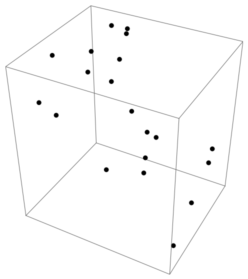
This generates a list of 20 random points in three-dimensional space.

```
In[2]:= pts = Table[Point[rcoord], {20}];
```

Here is a plot of the points.

```
In[3]:= Graphics3D[pts]
```

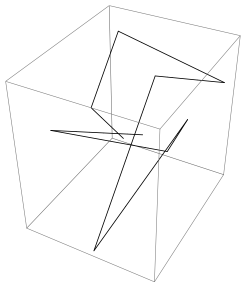
Out[3]=



This gives a plot showing a line through 10 random points in three dimensions.

```
In[4]:= Graphics3D[Line[Table[rcoord, {10}]]]
```

Out[4]=



If you give a list of graphics elements in two dimensions, *Mathematica* simply draws each element in turn, with later elements obscuring earlier ones. In three dimensions, however, *Mathematica* collects together all the graphics elements you specify, then displays them as three-dimensional objects, with the ones in front in three-dimensional space obscuring those behind.

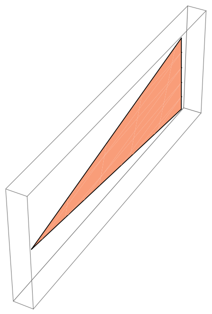
Every time you evaluate `rantri`, it generates a random triangle in three-dimensional space.

```
In[5]:= rantri := Polygon[Table[rcoord, {3}]]
```

This draws a single random triangle.

```
In[6]:= Graphics3D[rantri]
```

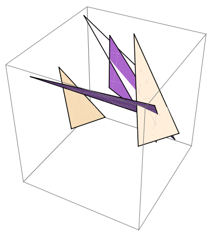
Out[6]=



This draws a collection of 5 random triangles. The triangles in front obscure those behind.

```
In[7]:= Graphics3D[Table[rantri, {5}]]
```

Out[7]=



By creating an appropriate list of polygons, you can build up any three-dimensional object in *Mathematica*. Thus, for example, all the surfaces produced by `ParametricPlot3D` are represented essentially as lists of polygons.

`Point[{`pt_1`,`pt_2`,…}]`	a multipoint consisting of points at pt_1, pt_2, …
`Line[{`$line_1$`,`$line_2$`,…}]`	a multiline consisting of lines $line_1$, $line_2$, …
`Polygon[{`$poly_1$`,`$poly_2$`,…}]`	a multipolygon consisting of polygons $poly_1$, $poly_2$, …

Primitives which can take multiple elements.

As with the two-dimensional primitives, some three-dimensional graphics primitives have multi-coordinate forms which are a more efficient representation. When dealing with a very large number of primitives, using these multi-coordinate forms where possible can both reduce the memory footprint of the resulting graphic and make it render much more quickly.

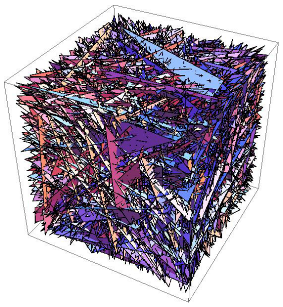
`rantricoords` defines merely the coordinates of a random triangle.

```
In[8]:= rantricoords := Table[rcoord, {3}]
```

Using the multi-coordinate form of `Polygon`, this efficiently represents a very large number of triangles.

```
In[9]:= Graphics3D[Polygon[Table[rantricoords, {10 000}]]]
```

Out[9]=



Mathematica allows polygons in three dimensions to have any number of vertices in any configuration. Depending upon the locations of the vertices, the resulting polygons may be non-coplanar or nonconvex. When rendering non-coplanar polygons, *Mathematica* will break the polygon into triangles, which are planar by definition, before rendering it.

The non-coplanar polygon is broken up into triangles. The interior edge joining the triangles is not outlined like the outer edges of the `Polygon` primitive.

```
In[10]:= Graphics3D[{Polygon[{{0, 0, 0}, {0, 0, 1}, {1, 1, 0}, {1, 0, 1}}]}]
```

Out[10]=